

Java Programming Language

Student Guide

SL-275-SE6 Rev G

D67426TC10

Edition 1.0

July 2010

D68073

ORACLE®

Copyright © 2007, 2010, Oracle and/or its affiliates. All rights reserved.

Disclaimer

This document contains proprietary information, is provided under a license agreement containing restrictions on use and disclosure, and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except as expressly permitted in your license agreement or allowed by law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Sun Microsystems, Inc. Disclaimer

This training manual may include references to materials, offerings, or products that were previously offered by Sun Microsystems, Inc. Certain materials, offerings, services, or products may no longer be offered or provided. Oracle and its affiliates cannot be held responsible for any such references should they appear in the text provided.

Restricted Rights Notice

If this documentation is delivered to the U.S. Government or anyone using the documentation on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

Table of Contents

關於本課程.....	Preface-xv
課程目標	Preface-xv
課程概述	Preface-xvii
課程學習地圖	Preface-xviii
各章節概述	Preface-xix
未涵蓋的主題	Preface-xxi
你準備齊全了嗎?	Preface-xxii
自我介紹	Preface-xxiii
如何使用課程教材	Preface-xxiv
通則 (Conventions).....	Preface-xxv
圖示	Preface-xxv
印刷通則	Preface-xxvi
額外的通則	Preface-xxvii
執行您的第一個 Java 程式	1-1
單元重點	1-1
其他資源	1-2
了解 JDK 軟體開發工具	1-3
了解 JDK 軟體對於開發 JAVA 應用程式所提供的支援... ..	1-4
JDK 佈署機制	1-11
Java 虛擬機器 (JVM)	1-12
Java TM 執行環境 (JRE TM)	1-14
了解 Java 程式的載入與執行.....	1-15
類別載入器 (Class Loader).....	1-16
位元碼檢驗器	1-16
開發一個簡單的 Java 程式.....	1-17
TestGreeting 程式	1-18
Greeting 類別	1-20
編譯與執行 TestGreeting	1-21
解決編譯時期的問題	1-22

建立 Java 應用程式	2-1
單元重點	2-1
其他資源	2-2
Java 應用程式的組成	2-3
檢視自定類別的建立過程：實作概念	2-6
建立類別	2-6
了解應用程式的主類別（Main Classes）	2-11
比較 Java 應用程式的靜態與動態觀點	2-12
表態觀點	2-12
動態觀點	2-13
建立，使用及移除物件	2-18
建立類別：欄位與建構子語法	3-1
單元重點	3-1
其它資源	3-2
類別宣告簡介	3-3
決定新類別的套件（package）名稱	3-3
宣告新類別使用的外部類別	3-4
宣告類別欄位	3-5
宣告類別建構子	3-6
宣告類別方法	3-7
宣告欄位：基本資料型別	3-9
宣告欄位：類別欄位	3-11
Java 標準類別函式庫	3-11
宣告欄位：識別名稱（Identifiers）的格式	3-15
宣告欄位：初始化欄位	3-16
使用預設值將欄位初始化	3-17
明確指定欄位的初始值	3-18
使用建構子將欄位初始化	3-19
注解，空白和關鍵字	3-24
注解	3-24
空白	3-25
Java 程式語言的關鍵字	3-25
目錄結構與套件	3-27
開發時期的目錄結構	3-27
編譯時使用 -d 選項	3-28
佈署	3-29
宣告類別：類別方法（Methods）的語法	4-1
單元重點	4-1
其他資源	4-2
了解類別方法	4-3
了解類別與類別方法間的關係	4-4
了解方法介面（method interface）與方法本體間的關係	4-5
了解欄位，參數，區域變數與方法本體的關係	4-6
了解方法本體	4-8
了解運算式（Expressions）	4-9

了解簡單運算式	4-10
運算式：進階議題	4-15
簡單運算式中數值型態的升級（Promotion）轉換	4-15
型別轉換	4-16
基本資料型別的 Autoboxing	4-17
執行複合運算式	4-18
敘述句（Statements）	4-19
區塊敘述句	4-21
分支敘述句	4-22
簡單的 if 敘述句	4-22
簡單的 if,else 敘述句	4-22
複雜的 if,else 敘述句	4-23
switch 敘述句	4-24
迴圈敘述句	4-26
for 迴圈	4-26
while 迴圈	4-27
do/while 迴圈	4-27
特殊的迴圈流程控制機制	4-28
類別方法的進階設計議題	4-30
在類別方法中使用變動參數	4-31
類別方法的進階設計議題：按值傳遞	4-32
類別方法的進階設計議題：this 參考	4-34
建構類別：使用封裝（Encapsulation）機制	5-1
單元重點	5-1
其他資源	5-2
了解封裝概念	5-3
使用 Java 技術進行封裝	5-6
使用 static 關鍵字	5-9
類別專屬欄位	5-9
類別專屬方法	5-10
static 初始化區塊	5-12
靜態引入（Static Imports）	5-13
建立陣列	6-1
單元重點	6-1
其他資源	6-2
宣告陣列	6-3
建立陣列	6-4
由物件參考所組成的陣列	6-5
將陣列初始化	6-6
多維度陣列	6-7
陣列邊界（Array Bounds）	6-8
使用加強版的 for 迴圈	6-8
陣列長度是無法改變的	6-9
使用繼承（Inheritance）機制建立類別	7-1

單元重點	7-1
其他資源	7-2
了解繼承概念	7-3
了解繼承的優點	7-5
使用 Java 技術實作繼承機制	7-6
步驟 1: 選擇父類別	7-7
步驟 2: 父類別哪些特性會被繼承	7-8
步驟 3: 宣告子類別	7-9
步驟 4: 加入子類別特有的屬性和方法	7-9
步驟 5: 需要時, 覆寫父類別方法	7-10
步驟 6: 加入所需的建構子	7-13
多型 (Polymorphism)	7-19
虛擬方法調用 (Virtual Method Invocation)	7-19
異質 (Heterogeneous) 集合	7-21
多型參數	7-21
Instanceof 運算子	7-23
強制轉型物件參考	7-24
Object 類別	7-25
Equals 方法	7-25
toString 方法	7-28
final 關鍵字	7-29
Final 類別	7-29
Final 方法	7-29
Final 變數	7-29
使用例外 (Exception) 類別和 Assertionsf 進行錯誤處理	8-1
單元重點	8-1
其他資源	8-2
例外 (Exceptions) 和 Assertions	8-3
例外	8-3
例外的範例	8-4
try-catch 敘述句	8-6
呼叫堆疊 (Call Stack) 機制	8-8
Finally 子句	8-9
例外的分類	8-10
常見的例外	8-12
處理或宣告例外的規則	8-13
方法覆寫和例外	8-15
產生自訂例外	8-16
丟出使用自訂例外	8-17
處理使用自訂例外	8-18
Assertions	8-19
建議的 Assertion 使用方式	8-19
控制執行時期 Assertion 的執行	8-22
宣告和使用特殊的類別型式	9-1
單元重點	9-1
其他資源	9-2

抽象方法 (Abstract Methods) 和抽象類別 (Abstract Classes)	9-3
一個抽象類別的範例	9-5
介面 (Interfaces)	9-7
Flyer 範例	9-7
多重介面的範例	9-12
介面宣告和使用規則	9-13
巢狀類別 (Nested Classes)	9-14
了解巢狀類別語法	9-15
內部類別 (Inner Classes)	9-16
Static 巢狀類別	9-19
內部介面, 列舉型別和標注 (Annotations)	9-19
列舉型別	9-20
Switch 敘述句	9-20
For 迴圈	9-21
具有欄位, 方法和建構子的列舉型別	9-22
使用泛型 (Generics) 和集合框架 (CollectionsFramework) ..	10-1
單元重點	10-1
其他資源	10-2
集合 API	10-3
集合實作	10-5
一個 Set 的範例	10-5
一個 List 的範例	10-6
Map 介面	10-8
一個 Map 的範例	10-9
舊式集合類別	10-10
集合排序	10-11
Comparable 介面	10-11
Comparator 介面	10-13
泛型 (Generics)	10-16
泛型 Set 範例	10-17
泛型 Map 範例	10-18
泛型: 了解型別參數	10-19
萬用型別參數	10-21
泛型: 重構已存在的非泛型程式	10-24
迭代器 (Iterators)	10-26
加強版的 for 迴圈	10-28
資料輸出輸入	11-1
單元重點	11-1
其他資源	11-2
命令列參數	11-3
系統屬性	11-4
Properties 類別	11-5
輸出入資料流的基礎理論	11-6
資料流中的資料	11-6

位元組資料流 (Byte Streams)	11-7
InputStream 類別的方法	11-7
OutputStream 類別的方法	11-8
字元資料流 (Character Streams)	11-9
Reader 類別的方法	11-9
Writer 類別的方法	11-10
節點資料流 (Node Streams)	11-11
一個簡單的範例	11-11
緩衝記憶體資料流 (Buffered Streams)	11-13
輸出入資料流的串接	11-14
處理資料流 (Processing Streams)	11-15
基本的位元組串流類別	11-16
FileInputStream 以及 FileOutputStream	
類別	11-17
BufferedInputStream 和 BufferedOutputStream 類別	11-17
PipedInputStream 和 PipedOutputStream	
類別	11-17
DataInputStream 和 DataOutputStream	
類別	11-17
ObjectInputStream 以及 ObjectOutputStream	
類別	11-18
序列化	11-21
物件序列化及物件圖 (Object Graphs)	11-21
寫入及讀取物件資料流	11-22
基本的字元資料流類別	11-25
InputStreamReader 以及 OutputStreamWriter 相關方法	11-26
位元組及字元間的轉換	11-26
使用其他字元組的編碼	11-26
FileReader 和 FileWriter 類別	11-27
BufferedReader 和 BufferedWriter 類別	11-27
StringReader 和 StringWriter 類別	11-27
PipedReader 和 PipedWriter 類別	11-27
主控台 (Console) 輸出入以及檔案輸出	12-1
學習目標	12-1
其他資源	12-2
主控台輸出入	12-3
將資料寫至標準輸出	12-3
從標準輸入讀取資料	12-4
簡易的格式化輸出	12-6
簡易格式化輸入	12-7
檔案以及檔案輸出入	12-8
建立一個新的檔案物件	12-8
檔案測試以及工具	12-9
檔案串流輸出入	12-10
使用網路技術實作多層架構應用程式	13-1

學習目標	13-1
其他資源	13-2
網路	13-3
Sockets	13-3
建立連線 (Connection)	13-3
定址連線	13-5
通訊埠號	13-5
Java 網路模型	13-6
迷你 TCP/IP 伺服器	13-7
迷你 TCP/IP 客戶端	13-8
URL 類別	13-9
實作多執行緒應用程式	14-1
學習目標	14-1
其他資源	14-2
執行緒	14-3
建立執行緒	14-4
啟動執行緒	14-5
執行緒排程	14-6
終結執行緒	14-7
執行緒的基本控制	14-9
測試執行緒	14-9
存取執行緒的優先順序	14-9
暫時停止執行緒	14-9
另外一個建立執行緒的方法	14-12
選擇建立執行緒的方法	14-13
使用 synchronized 關鍵字	14-14
問題	14-14
物件鎖定旗標 (Object Lock Flag)	14-16
釋放鎖定旗標	14-18
使用 synchronized 機制	14-19
執行緒狀態	14-20
死結 (Deadlock)	14-20
執行緒互動	14-21
情境	14-21
問題	14-21
解決方案	14-21
Wait 以及 notify 方法	14-21
執行緒狀態	14-23
同步監控 (Monitor) 模型	14-24
綜合應用	14-25
Producer 執行緒	14-26
Consumer 執行緒	14-26
SyncStack 類別	14-28
SyncTest 範例	14-31
應用 SwingAPI 建立 Java 圖形使用者介面	15-1

目標	15-1
其他資源	15-2
可插接式 look-and-feel	15-5
Swing 架構.....	15-6
Swing 套件.....	15-8
檢視 Java 技術 GUI 的構成	15-10
Swing 程式容器.....	15-12
頂層程式容器	15-12
多用途程式容器	15-14
特殊用途程式容器	15-15
Swing 元件.....	15-18
Swing 元件階層架構.....	15-19
按鈕 (Buttons)	15-20
文字元件	15-21
不可編輯資訊的顯示元件	15-23
選單	15-24
格式化的顯示元件	15-25
其他基本控制項	15-26
Swing 元件屬性.....	15-28
共通元件屬性	15-28
元件專有屬性	15-29
版面配置管理程式.....	15-30
BorderLayout 版面配置管理程式	15-30
FlowLayout 版面配置管理程式	15-31
BoxLayout 版面配置管理程式	15-32
CardLayout 版面配置管理程式	15-33
GridLayout 版面配置管理程式	15-34
GridBagLayout 版面配置管理程式	15-34
GroupLayout 版面配置管理程式.....	15-35
建構 GUI 應用程式	15-36
使用程式建構	15-36
處理 GUI 產生的事件 (event)	16-1
目標	16-1
其他資源	16-2
何謂事件?	16-3
Java SE 事件模型	16-4
委派模型	16-4
監聽程式範例	16-5
GUI 的行為	16-7
事件分類	16-7
複雜的範例	16-9
多重監聽程式	16-12
開發事件監聽程式.....	16-13
事件配接器 (adapters)	16-13
使用內部類別 (inner classes) 進行事件處理.....	16-14

使用匿名類別進行事件處理	16-15
Swing 的並行性 (Concurrency)	16-17

前言

關於本課程

課程目標

藉由完成本次課程，你將學會：

- 設計 Java™ 技術應用程式、善用 Java 語言物件導向 (object-oriented) 的特色，例如：封裝 (encapsulation)、繼承 (inheritance)、與多型 (polymorphism)。
- 從命令列 (command line) 執行 Java 技術應用程式。
- 使用 Java 資料型別 (data types) 與表示式 (expressions)。
- 使用 Java 流程控制構件 (constructs)。
- 使用陣列 (arrays) 與其他資料集合。
- 利用例外 (exception) 處理，實作錯誤處理技術。
- 利用 Java 技術之 GUI 元件：面板 (panels)、按鈕 (buttons)、標籤 (labels)、文字欄位 (text fields)、與文字區域 (text area)，建立事件驅動 (event-driven) 的圖形化使用者介面 (graphical user interface, GUI)。
- 實作輸入 / 輸出 (I/O) 功能、讀寫資料與文字檔案。
- 設計多執行緒 (multithreaded) 的程式。
- 設計簡易的 Transmission Control Protocol/Internet Protocol (TCP/IP) 網路用戶端 (client)、透過 sockets 與伺服器通訊。

Java™ 程式語言課程提供了進階 **Java** 應用程式中有關物件導向程式設計的學問及技巧。在本課程中，你將學習 **Java** 程式語言的語法、物件導向的概念、以及 **Java** 執行時期 (runtime) 環境裡更複雜的特色，例如：GUI 支援、多執行緒、及網路等等。本課程涵蓋幫助你準備 **Sun Certified Programmer for the Java™ Platform (SCJP)** 考試的必要知識。關於該考試的資訊，可參考網址：<http://www.sun.com/training/certification/java/>

不過，僅憑 **SL-275** 並不能使你立刻通過該項考試。你得花上數個月的時間，藉由建立真正的程式以磨練這些技巧，也得複習考試科目，並研究本課程中未提及的領域。**SCJP** 考試科目同樣可在上面列出的網址找到。

課程概述

本課程首先說明 Java 執行時期環境及 Java 程式語言之語法。隨著物件導向應用到 Java 程式語言，再繼續解釋它的概念。課程進行的過程中，會逐漸討論 Java 平臺其他進階的特點。

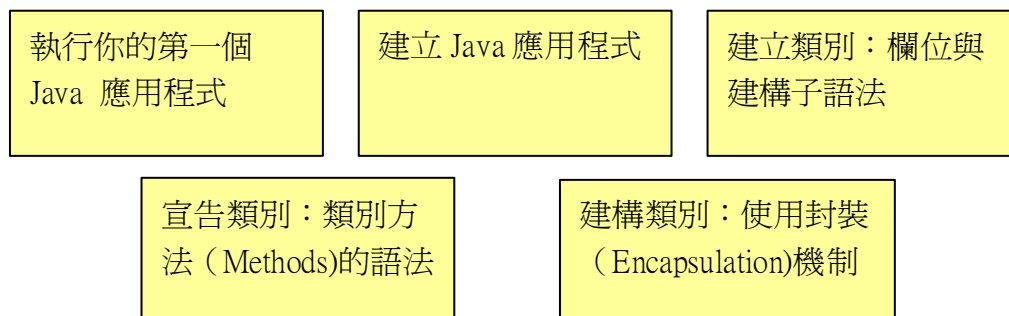
本課程的學生應熟悉使用 Java 程式語言或其他語言，來實作基礎的程式設計。因本課程是 SL-110：Java 程式語言基礎 (Fundamentals of the Java™ Programming Language) 的後續課程。

由於 Java 程式語言與作業系統各自分開，它所產生的 GUI 因此會依程式碼執行時所在的作業系統而改變。本次課程中，程式碼範例均執行於 Solaris? Operating System (Solaris OS) 以及 Microsoft Windows 作業環境；連帶的，手冊裡的圖片也有 Motif 與 Windows 兩種 GUI。當然，課程的內容適用所有 Java 作業系統平臺。

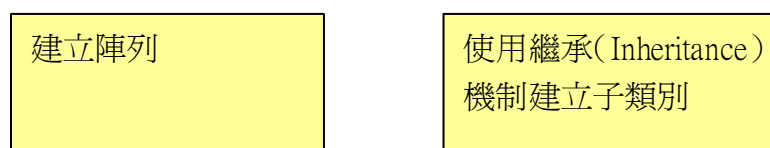
課程學習地圖

以下的課程學習地圖讓你察看哪些是你已學會的部份，以及接下去你將學習哪些部份。

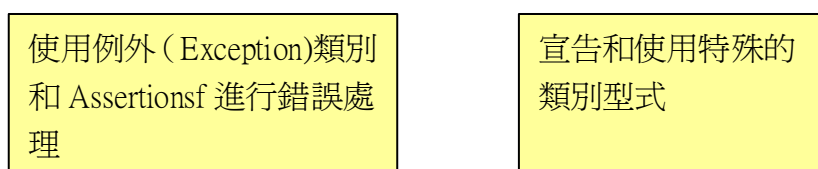
Java 程式語言基礎



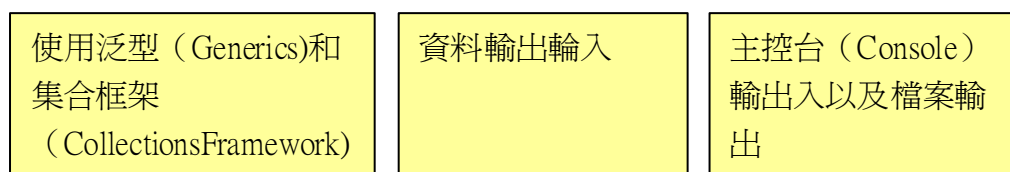
更多的物件導向程式設計



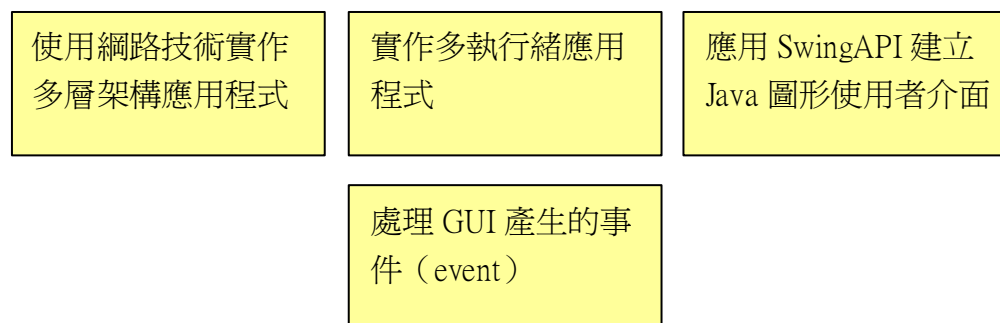
建立應用程式



開發圖形使用者介面



進階 Java 程式設計



各章節概述

- 章節 1 —— 「執行你的第一個 Java 應用程式」
提供一個 Java 程式語言大致上的概觀及它的主要特色，並介紹一個簡單的 Java 應用程式。
- 章節 2 —— 「建立 Java 應用程式」
檢視 Java 技術應用程式的構成。檢視的觀念包括：類別 (class)、實例 (instances)、及參照 (reference)。
- 章節 3 —— 「建立類別：欄位與建構子語法」
著重在運用欄位來對映物件的狀態。討論內容包含：宣告欄位的語法、使用預設值或指定值進行初始化、以及使用建構子。
- 章節 4 —— 「宣告類別：類別方法 (Methods) 的語法」
著重在運用方法來實作各種行為。解釋方法的識別標誌 (signature)、方法的主體，此外也說明表示式、含括運算子、以及 Java 程式控制的語法。
- 章節 5 —— 「建構類別：使用封裝 (Encapsulation) 機制」
討論封裝的概念，與使用 Java 程式語言實作封裝。檢視 package 敘述的用法、存取修飾字 (access modifiers)、與 static 關鍵字。
- 章節 6 —— 「建立陣列」
敘述 Java 陣列如何宣告、建立、初始化、及複製。
- 章節 7 —— 「使用繼承 (Inheritance) 機制建立子類別」
詳述以下觀念：利用繼承來建立特製的型別 (types)、以及實作多型 (polymorphism)。此外也包含 final 關鍵字的用法。
- 「使用例外 (Exception) 類別和 Assertions 進行錯誤處理」
例外提供你在執行時期讓錯誤自動入甕的機制。本章節探討預設及使用者自訂的例外、以及判定的用法。
- 章節 9 —— 「宣告和使用特殊的類別型式」
討論列舉型別 (enumerated types)、介面 (interfaces)、巢狀類別 (nested class) 的宣告及用法。
- 章節 10 —— 「使用泛型 (Generics) 和集合框架 (CollectionsFramework)」
檢視集合架構與 Java 程式語言中泛型的用法。
- 章節 11 —— 「資料輸出輸入」
敘述可用來讀寫資料與文字的類別。同時也討論物件的序列化 (serialization)。

- 章節 12 —— 「主控台 (Console) 輸出入以及檔案輸出」
介紹實作大型、以文字為基礎的應用程式時有用的原則，如控制臺與檔案 I / O。
- 章節 13 —— 「使用網路技術實作多層架構應用程式」
介紹 Java 網路程式套件，並示範 TCP / IP 主從式 (client-server) 模型。
- 章節 14 —— 「實作多執行緒應用程式」
執行緒是一項複雜的議題；本模組解釋執行緒與 Java 程式語言相關的部份，並透過一個易懂的範例介紹執行緒間的溝通與同步 (synchronization)。
- 章節 15 —— 「應用 Swing API 建立 Java 圖形使用者介面」
本模組討論各式各樣的 Swing API GUI 元素。
- 章節 16 —— 「處理 GUI 產生的事件 (event)」
僅在框架 (frame) 裡建立 GUI 元件的版面配置 (layout) 尚嫌不足。你須撰寫程式碼以處理發生的事件，例如點擊按鈕或鍵入字元。本模組示範如何撰寫 GUI 事件處理程式。

未涵蓋的主題

本課程不討論以下在 Sun 教學服務的課程中可學到的主題：

- 物件導向分析與設計 —— 討論於 OO-226：*應用 UML 進行物件導向應用程式分析與設計 (Object-Oriented Application Analysis and Design Using UML)*。
- 一般程式設計概念 —— 討論於 SL-110：*Java™ 程式語言基礎 (Fundamentals of the Java? Programming Language)*。

請參考 Sun 教學服務目錄，以獲得更具體的資訊或報名註冊。

你準備齊全了嗎？

為了能順利學習本課程，請先確定下列條件，你的答案都是肯定的。

在參與本課程之前，你應先完成 **SL-110：Java™** 程式語言基礎，或是曾經：

- 以 **C** 或 **C++** 建立並編譯程式
- 使用文字編輯器建立並編輯文字檔案
- 使用 **World Wide Web (WWW)** 網路瀏覽器，例如 **Netscape Navigator™**

自我介紹

課程已介紹完畢，現在請向其他學生及講師介紹自己，至少包含下列項目：

- 姓名
- 公司組織
- 職稱、業務、及工作職責
- 與本課程所論及的主題相關之經驗
- 參加本課程之原因
- 對課程的期待

如何使用課程教材

為了讓你能成功的學習本課程，這些課程教材包含了由以下元件所組成的學習模組：

- 課程目標 —— 在本課程結束並完成全部的單元目標後，你應能達成課程目標。
- 單元目標 —— 在完成部份的教學內容後，你應能達成單元目標。單元目標能輔助課程目標及其他更高階的單元目標。
- 授課 —— 講師會針對模組的單元目標講解更多資訊。這些資訊幫助你學習必要的知識與技巧，以順利進行活動。
- 活動 —— 活動有許多不同的形式，例如：練習、自我測驗、討論、以及實機示範。活動能協助你掌握單元目標。
- 視聽教具 —— 講師可能運用許多視聽教具來傳達概念。常用的視聽教具有圖片、動畫、影片等等。

通則 (conventions)

本課程於各個不同的訓練單元與其他學習資源中，採用以下通則。

圖示



額外資源 -- 表示對於該模組中敘述之主題，提供其他額外資訊做為參考。



討論 -- 建議你在這個時候，對目前主題進行小組或課堂討論



註釋 -- 表示幫助學生的額外資訊，無論得知與否，對他們此刻正亟於理解的課題，均不造成影響。即便在沒有這項資訊的情況下，學生也應能理解概念或完成作業。舉例來說，如：關鍵字 (**keyword**) 捷徑、次要的系統微調，均是註釋用的資訊。

印刷通則

Courier 字型會用在指令的名稱、檔案、目錄、程式碼以及目前電腦螢幕上的輸出；例如：

使用 `ls -al` 列出所有檔案。

`system% You have mail`

Courier 字型也被用在程式結構上，比方說類別 (**class**) 名稱、方法 (**method**)、關鍵字 (**keyword**)；例如：

`getServletInfo` 方法用來取得著作資訊。

`java.awt.Dialog` 類別包含 `Dialog` 建構子 (**constructor**)。

Courier bold 字型則用在你鍵入的字元或數字；例如：

欲列出本目錄下的檔案，則鍵入：

`# ls`

Courier bold 字型也用在正文敘述中所指涉的每一行程式碼；例如：

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
```

注意 `javax.servlet` 介面 (**interface**) 被匯入來容許存取它的 `life cycle` 方法 (第 2 行)。

Courier italics 字型用於變數與指令列中，暫代實際名稱或實際數字的文字；例如：

要刪除檔案，使用 `rm filename` 這個指令。

Courier italic bold 字型表示需要學生自行輸入內容值的變數；例如：

鍵入 `chmod a+rx filename` 賦予全域、群組、使用者擁有讀、寫、執行的權限。

Palatino italics 字型用在書名、新字詞、術語、或是欲強調的字詞；例如：

閱讀 *User's Guide* 第 6 章。

這些被稱為 **class** 選項。

額外的通則

Java 程式語言範例會用到以下額外的通則：

- 方法名稱的後面不接括號，除非要顯示正式或實際的參數列表；例如：
“dolt 方法...” 指的是所有叫做 **dolt** 的方法。
“dolt() 方法...” 指的是一個不帶參數的 **dolt** 方法。
- 程式中的斷行只會出現在有分隔符號（逗號）、連接符號（運算子）、或是空白間隔的地方。被截斷的程式碼，會在起始行的下方，空四格後對齊。
- 假設在 **Solaris™ Operating System (Solaris OS)** 使用的指令與在 **Microsoft Windows** 平臺使用的指令不同，兩者均會列出；例如：
假設在 **Solaris OS** 環境下工作：
`$CD SERVER_ROOT/BIN`
假設在 **Microsoft Windows** 環境下工作：
`C:\>CD SERVER_ROOT\BIN`

執行您的第一個 Java 程式

單元重點

完成這個單元後，您將能夠：

- 了解 JDK™ 是什麼
- 了解 Java 程式的載入與執行過程
- 開發一個簡單的 Java 程式

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Gosling, Joy, Bracha, and Steele. *The Java Language Specification, Third Edition*. Prentice-Hall. 2005. [請參考：<http://java.sun.com/docs/books/jls/>]. 存取日期：May 1, 2007.

了解 JDK 軟體開發工具

JDK 軟體開發工具具有下列三項功能：

- 開發 Java 應用程式
- 佈署 Java 應用程式
- 執行 Java 應用程式

圖 1-1 展現了 JDK 軟體開發工具的結構概要，並列出 JDK 元件與相對應開發活動之間的關係。



圖 1-1 Java SE JDK 概觀

JDK 軟體開發工具由下列幾個部份組成：

- Java 程式語言
- 工具及工具的 API
- 佈署機制
- Java 平台標準版 (Java SE) 的函式庫類別
- Java 平台的虛擬機器 (Java Virtual Machine(JVM™))



注意 -- 嚴格來說 Java 程式語言並非 JDK 軟體開發工具的一部分。JDK 軟體開發工具基本上就是提供並支援以 Java 語言為基礎的應用程式開發，因此在討論 JDK 軟體開發工具時通常會一併討論 Java 程式語言。

要使用 JDK，必須先依照您的作業系統下載不同版本的 JDK 軟體開發工具及說明文件。您可以經由下列的連結下載 JDK：

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

了解 JDK 軟體對於開發 **JAVA** 應用程式所提供的支援

JDK 提供了下列三項機制來幫助您開發 **Java** 應用程式：

- **Java** 程式語言
- **JDK** 工具
- **JDK** 函式庫

Java 程式語言

Java 程式語言的用途廣泛且具備平行處理機制及強型別 (**Strongly typed**)，是一個以類別為基礎的物件導向語言。**Java** 程式語言由 **Java** 語言規格書所定義。在 **Java** 中基本的建構單元稱為類別 (**class**)。**Java** 語言規格書中明定了類別及所有 **Java** 語法的規範。程式 1-1 為 **Java** 類別程式範例。

程式 1-1 **Java** 類別程式範例

```
1  package trader;
2  import java.io.Serializable;
3  public class Stock implements Serializable {
4      private String symbol;
5      private float price;
6
7      public Stock(String symbol, float price){
8          this.symbol = symbol;
9          this.price = price;
10     }
11
12     // Methods to return the private values of this object
13     public String getSymbol() {
14         return symbol;
15     }
16
17     public float getPrice() {
18         return price;
19     }
20
21     public void setPrice(float newPrice) {
22         price = newPrice;
23     }
24
25     public String toString() {
26         return "Stock: " + symbol + " " + price;
27     }
```

JDK 工具以及其 API

圖 1-2 為 Java 標準版 JDK 的工具分類。

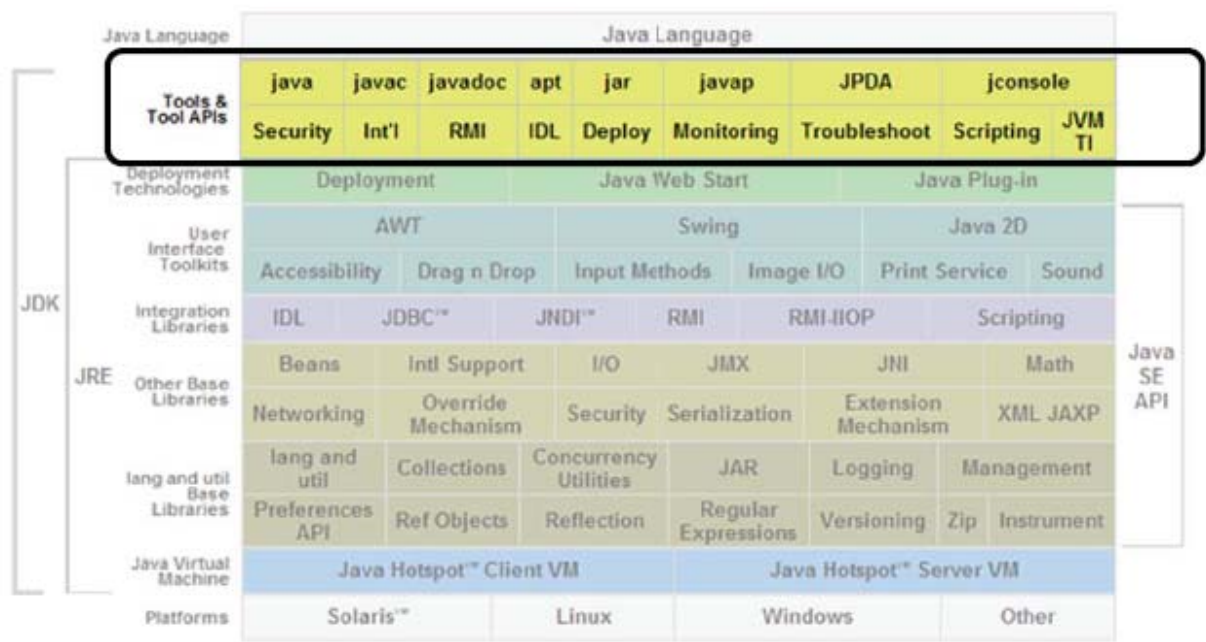


圖 1-2 特別標示出來的部份是 Java 標準版的 JDK 及其工具

JDK 工具用來協助 Java 程式的開發及佈署。JDK 工具可分為兩大類：

- 基本工具

基本工具可以協助開發人員寫作、建構 (build) 及執行 Java 程式。表 1-1 列出在開始接觸 Java 時您會經常使用的基本工具。

表 1-1 簡單列舉 Java 標準版基本工具

工具名稱	功能
javac	Java 語言的編譯器
java	執行 Java 程式
jdb	Java 除錯器
javadoc	API 文件產生器
jar	Java 封裝檔 (JAR) 產生器以及管理工具

- 進階工具

進階工具包含幾個部份。每個部份都可以協助您善用特定的技術來開發 **Java** 程式。表 1-2 舉出進階工具的分類。

表 1-2 進階工具分類

工具分類	說明
安全工具	協助產生具有安全考量的程式
國際化工具	幫助程式區域化
遠端呼叫工具	產生跨網路的程式
分散式物件架構 (CORBA) 工具	產生以 (CORBA) 技術為基礎的網路程式
Java 佈署工具	提供壓縮與解壓工具以協助 Java 程式的佈署
Java Plug-in 工具	Java Plug-in 技術使用工具
Java web start 工具	Java web start 技術使用工具
Java 監控與管理 (JMX) 工具	與 JMX 技術連結使用工具
Java web services 工具	支援 web services 的開發
實驗性工具	這些工具僅供測試實驗用，在未來新版 JDK 中並不一定會繼續包含這些實驗性的工具



注意 --JDK 軟體開發工具本質上是一組函式庫，並且包含了相關說明文件，您可以藉由這些資源撰寫程式。這些 **API** 不在本節的討論範圍，詳細資訊請您參考 **JDK** 文件的 **Java Platform Debugger Architecture (JPDA)** 章節中。

JDK 函式庫

JDK 函式庫由預先訂定的類別所構成。這些類別依其功能而分類。在 Java 中這些函式庫稱為套件 (package)。圖 1-3 標示出 Java 標準版的函式庫。

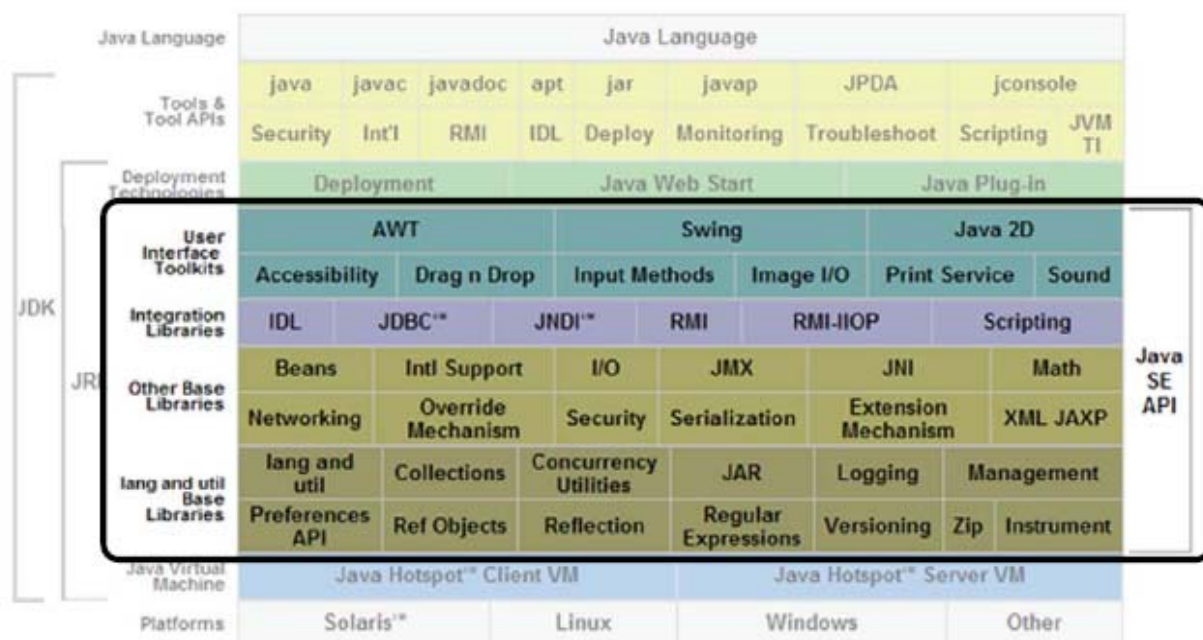


圖 1-3 Java 標準版的函式庫

函式庫可分為下列幾個主要類別：

- 使用者介面工具組
使用者介面工具組由一系列的使用者圖形介面函式庫所組成。這些函式庫提供 **GUI** 元件（按鈕，標籤，頁籤等）並支援拖曳，音效，可存取性，輸入，列印等功能。
- 整合函式庫
整合函式庫提供各式的網路技術並支援分散式應用程式。這些網路技術包含 **RMI**, **CORBA**, **Java™** 資料庫連結 (**JDBC™**)，**JNDI™API**。
- 其它基礎函式庫
這是一組基礎的函式庫。這類函式庫包含安全、國際化、**Java** 管理延伸介面 (**Java Management Extension, JMX**) 以及其他類似的函式庫。
- 語言和工具的基本函式庫
這類函式庫用來建立應用程式的核心部份。**lang** 及 **util** 屬於核心函式庫。在這組函式庫中也包含了支援同時存取（支援多執行緒應用程式）、記錄、管理等其他功能。

表 1-3 列舉幾個標準函式庫中常用的類別。

表 1-3 標準函式庫中常用的類別

函式庫名稱	函式庫中的類別	目的
java.lang	Enum, Float, String, Object	Java 語言的基本類別
java.util	ArrayList, Calendar, Date	工具類別
java.io	File, Reader, Writer	輸出輸入類別
java.math	BigDecimal, BigInteger	數學精算類別
java.text	DateFormat, Collator	文字格式處理類別
javax.crypto	Cipher, KeyGenerator	資訊加密類別
java.net	Socket, URL, InetAddress	網路應用類別
java.sql	ResultSet, Date, Timestamp	結構化查詢語言 (SQL) 支援類別。
javax.swing	JFrame, JPanel	圖形使用者介面類別
javax.xml.parsers	DocumentBuilder, SAXParser	可延伸標籤語言 (XML) 支援類別

JDK 函式庫文件 (亦稱之為 Java API 文件) 的格式為 HTML 檔案。整個文件的排版非常有層次性。在主頁面列出所有套件的連結，當您選擇某個套件時則會下方列出套件中的所有類別。點選類別後則顯示該類別的相關資訊。圖 1-4 為類別的說明頁面。

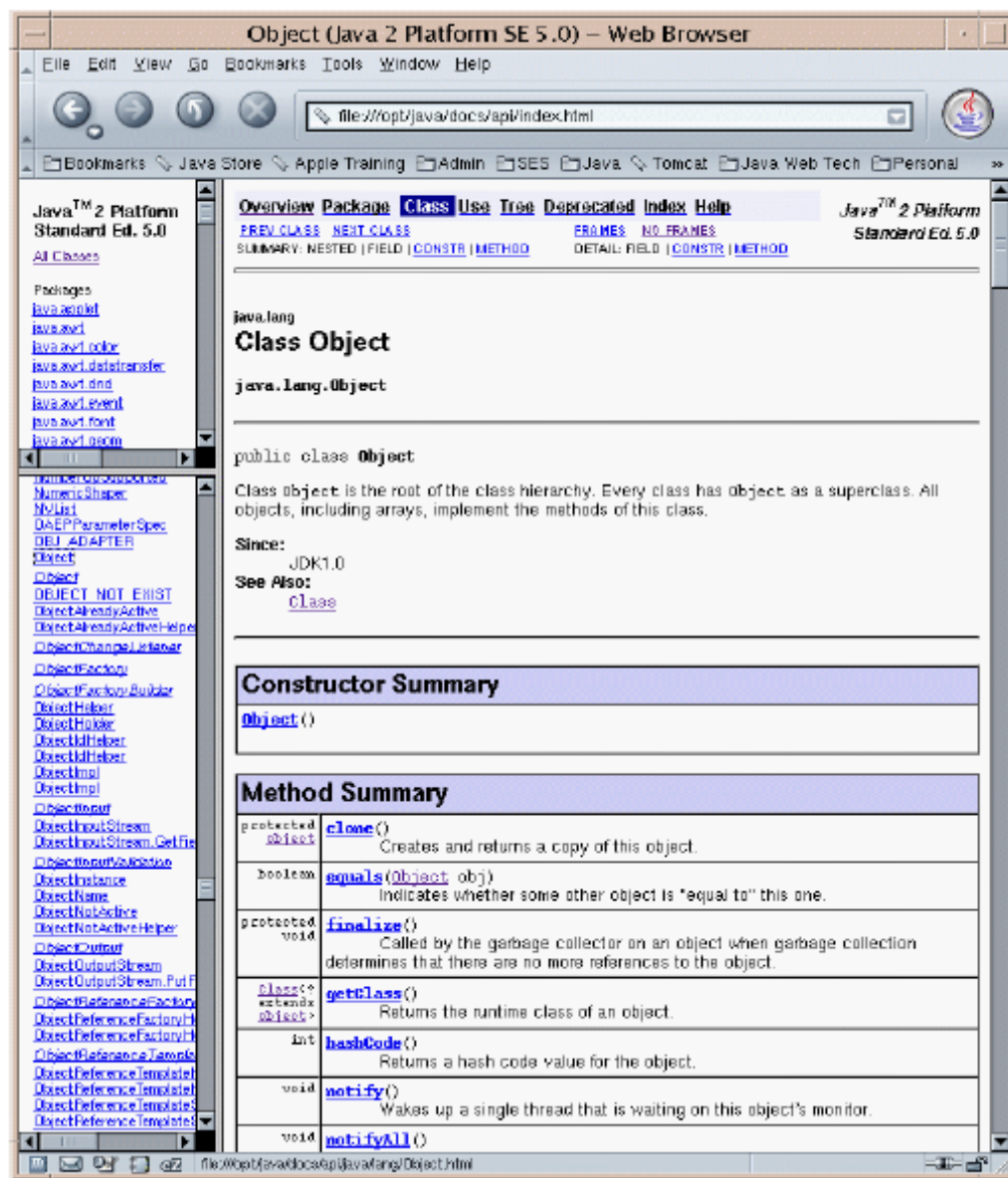


圖 1-4 Java API 文件

類別技術文件包含下列幾個部份：

- 類別階層架構
- 類別說明及其用途
- 屬性
- 建構子
- 方法
- 屬性說明
- 建構子及其參數說明
- 方法及其參數說明

JDK 佈署機制

圖 1-5 標示 JDK 軟體開發工具的佈署技術



圖 1-5 Java 標準版 JDK 的佈署機制

JDK 軟體開發工具支援桌上型應用程式用戶端的佈署。其提供下列 2 種佈署機制：

- **Java Plug-in**

Java Plug-in 技術使用在佈署網頁上執行的 Java Applet。支援的瀏覽器有 Internet Explorer Mozilla 以及 Netscape Navigator。



注意 --Java Applet 只能於瀏覽器上執行。

- **Java Web Start**

Java Web Start 技術透過Java Network Launching Protocol (JNLP)協定來佈署桌上型應用程式。

Java 虛擬機器 (JVM)

本章節透過問答方式來介紹說明 JVM(Java Virtual Machine)。

- 什麼是 JVM?

JVM 由編譯過後的 Java 類別組成，其目的在執行 Java 程式。Java 類別編譯後成為位元碼 (byte code)。JVM 載入位元碼加以解譯執行。圖 1-6 顯示應用程式，JVM，操作系統，硬體平台間的關係。



圖 1-6 JVM，作業系統，應用程式

- JVM 對平台有依賴性嗎？

JVM 於不同平台上有其特別版本。例如要在 Solaris 系統上執行 Java 程式必須要有 Solaris 適用的 JVM，要在 Windows 系統上執行則必須有適用於 Windows 系統的 JVM。

- Java 程式對平台有依賴性嗎？

Java 語言所開發的程式對平台沒有依賴性。也就是說可以在支援 JVM 的不同平台上執行。

- 什麼是 Java HotSpot™ 虛擬機器？

第一代的 JVM 為直譯式位元碼。最新版本的 JVM 則是會在執行前先編譯位元碼。這種動態式編譯的功能可提高執行效率。JVM 編譯器會動態的識別重複執行的程式碼稱之為熱區 (hotspots)，並對此部分的位元碼進行最佳化。因此稱之為 HotSpot JVM。

- 什麼是 Java HotSpot™ 用戶端虛擬機器？

對於程式用戶端的執行平台上 JDK 開發工具提供一 Java HotSpot™ 用戶端虛擬機器。這個虛擬機器可以縮短程式起始時間並減少記憶體的使用。

- 什麼是 Java HotSpot™ 伺服器端虛擬機器？

對於所有的平台 JDK 開發工具提供 Java HotSpot 伺服器端虛擬機器。
伺服器端虛擬機器其目的在於加速程式執行速度。

圖 1-7 標示 JDK 開發工具所支援的平台以及 JVM

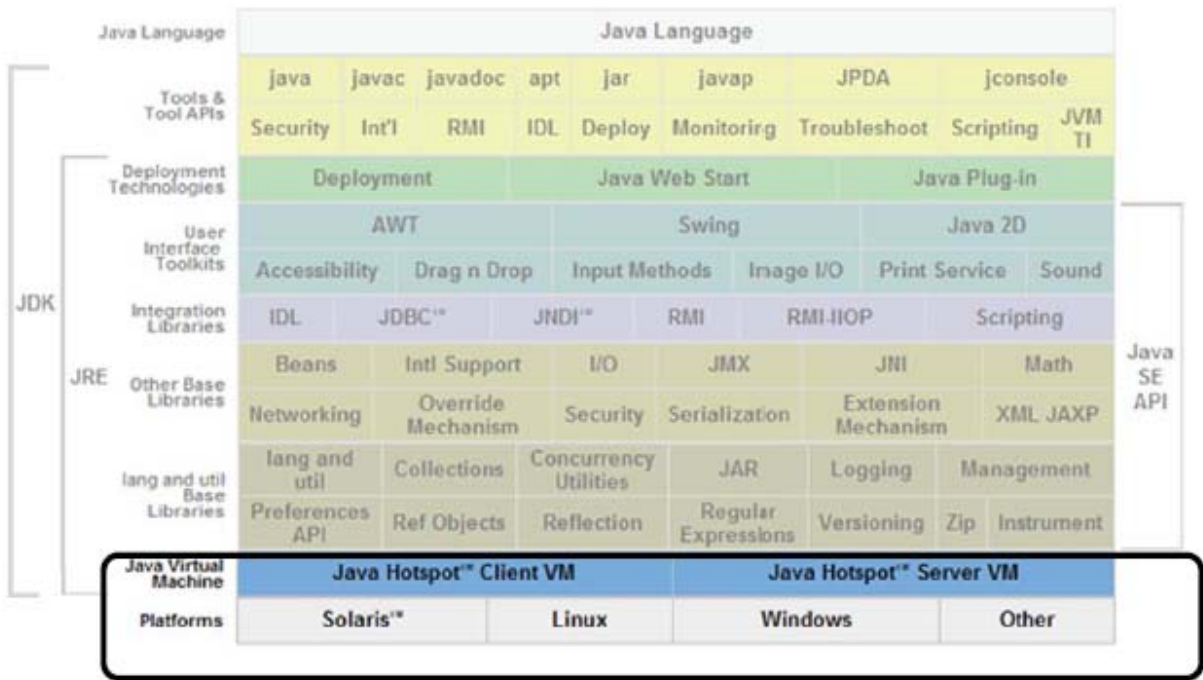


圖 1-7 JDK 開發工具支援的平台以及 JVM

Yu-Wei SHEN (SHEN_Yu-Wei@yahoo.com.tw) has a non-transferable license to use this Student Guide

Java™ 執行環境 (JRE™)

JRE™ 是 JDK 的子集合。其包含了 JDK 中執行 Java 程式所需的元件，但未包含佈署的元件。

圖 1-8 顯示 JRE 與 JDK 共同的元件。

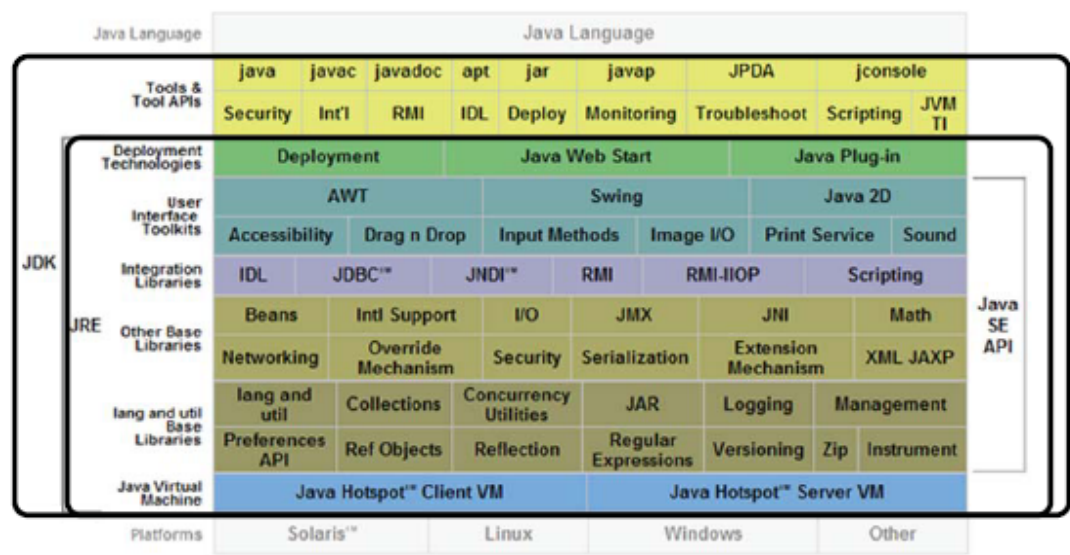


圖 1-8 JDK 與 JRE 的比較

了解 Java 程式的載入與執行

圖 1-9 顯示 JVM 載入與執行 Java 程式。

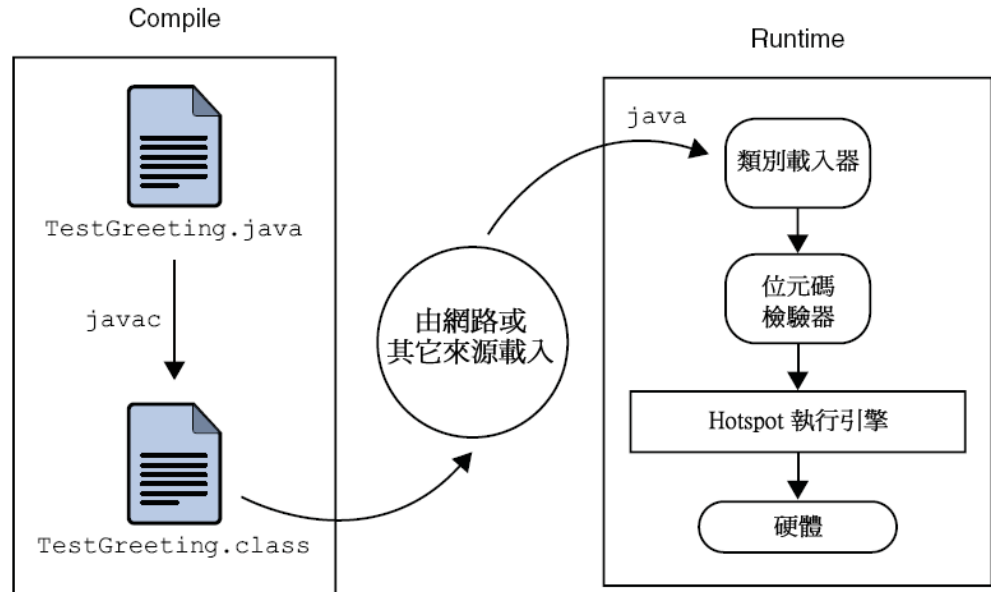


圖 1-9 JRE 的運作方式

Java 原始檔經編譯後由文字檔成為位元碼檔，並儲存成以 **.class** 為副檔名的檔案。

在執行時，Java HotSpot 執行引擎會將位元碼檔載入、檢查並執行。若是 Applets 則是下載位元碼檔，然後由瀏覽器內嵌的 JVM 執行。

以下會對 JVM 的三個主要任務做進一步詳細的討論：

- 載入程式 - 由類別載入器負責
- 檢驗程式 - 由位元碼檢驗器負責
- 執行程式 - 由執行引擎負責

類別載入器 (Class Loader)

程式執行時類別載入器會將所需的類別載入。類別載入器以空間命名的方式來區別類別是由本端系統載入還是經由網路遠端作為安全的考量。本端系統的類別會先被載入，這樣一來可以防止木馬程式的入侵。

在載入所有類別後，可執行檔的記憶體配置就已經決定了。這時候特定記憶體位址會被指定給特定的符號參考，且會產生一個對照表。因為記憶體的配置發生於執行時期，因此 **Java** 直譯器會限制未經授權的取用加以保護受限的程式碼。

位元碼檢驗器

Java 程式在執行前會通過多個檢驗。**JVM** 會檢驗位元碼的格式是否有錯，是否違反物件取用權限或是更改物件型態。

注意 -- 所有經由網路載入的類別也會被位元碼檢驗器檢驗。



檢驗流程

位元碼檢驗器會對程式碼作四種檢驗，以確保程式符合 **JVM** 的規範不會破壞系統的完整性。假設檢驗器完成所有的檢驗且沒有錯誤訊息，則可以確認下列事項：

- 檔案格式符合 **JVM** 的規範
- 沒有違反物件取用權限
- 程式不會產生運算時堆疊的 **Overflow** 或 **Underflow**
- 程式中的所有參數型別都是正確的
- 沒有錯誤的資料轉換，例如整數型態 (**int**) 的資料被轉換為物件

開發一個簡單的 Java 程式

就像其它程式語言一樣，您可以使用 **Java** 程式語言撰寫應用程式。程式 1-2 和 程式 1-3 列出經典範例 **Hello World**。

程式 1-2 TestGreeting.java 程式

```
1  //
2  //"Hello World" 程式範例
3  //
4  public class TestGreeting {
5      public static void main (String[] args) {
6          Greeting hello = new Greeting();
7          hello.greet();
8      }
9  }
```

程式 1-3 Greeting.java 類別

```
1  public class Greeting {
2      public void greet() {
3          System.out.println("hi");
4      }
5  }
```

TestGreeting 程式

以下將會說明 TestGreeting 程式。

程式 1-4 為範例程式的 1 至 3 行程式碼

程式 1-4 1 至 3 行

```
1  //  
2  // Sample "Hello World" application  
3  //
```

1 至 3 行為程式中的註解，使用 `//` 表示。

程式 1-5 第 4 行

```
4  public class TestGreeting {
```

第 4 行宣告類別的名稱。原始檔經編譯後將會與產生類別名稱同名的 `.class` 檔。假設您未告訴編譯器產出的目錄，則會使用原始檔所在的目錄在這個例子中編譯器會產生 `TestGreeting.class`。

程式 1-6 第 5 行

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

第 5 行為程式開始執行的起點。必須告訴 Java 直譯器由哪裡執行否則會執行失敗。

其它程式語言，像是 C 和 C++ 也都是用 `main()` 宣告程式執行的起始點。在這個小節會做簡單介紹，詳細內容會於課程中說明。

假如程式透過命令列傳入參數，這些參數會以字串陣列 `args` 傳給 `main()` 這個方法。這個例子中並未使用任何參數。

以下說明第 5 行中的每個元素：

- **public** - 代表 `main()` 方法可以被任意呼叫，包括 Java 直譯器。
- **static** - 這個關鍵字告訴編譯器 可以呼叫 TestGreeting 中的 `main()` 方法。 **static** 方法不需要產生實例即可執行。
- **void** - 這個關鍵字指出 `main()` 方法不回傳任何值。這個特性非常重要，因為 Java 程式語言對於型別的檢驗非常小心，以確保回傳的型別與宣告的是一樣。

- `String args[]` - 代表在 `main` 方法中宣告一個字串陣列參數。當 `main` 被呼叫時命令列的參數會帶給 `args`。例如：

```
java TestGreeting args[0] args[1] . . .
```

程式 1-7 第 6 行

```
6      Greeting hello = new Greeting();
```

第 6 行展示如何產生一個物件，然後指定給變數 `hello`。`new Greeting` 的語法告訴 Java 直譯器要建構一個 `Greeting` 類別的新物件。

程式 1-8 第 7 行

```
7      hello.greet();
```

第 7 行展示呼叫物件的方法。方法的實作在 `Greeting.java` 的第 3 至 5 行。

程式 1-9 第 8 和 9 行

```
8      }  
9  }
```

第 8 和 9 行各有一個大括弧，各別表示 `main()` 方法和 `TestGreeting` 類別的結束。

Greeting 類別

以下說明 **Greeting** 類別。

程式 1-10 第 1 行

```
1 public class Greeting {
```

第 1 行宣告 **Greeting** 類別。

程式 1-11 第 2 至 4 行

```
2 public void greet() {  
3     System.out.println("hi");  
4 }
```

第 2 至 4 行展示方法的宣告。**greet** 方法宣告為 **public**，因此 **TestGreeting** 程式可以使用它。因為 **greet** 方法沒有回傳值，所以宣告回傳 **void**。**greet** 方法將訊息傳送到標準輸出資料流。其所呼叫的 **println()** 方法便是用來將訊息傳送到標準輸出資料流。

程式 1-12 第 5 行

```
5 }
```

第 5 行的大括弧為 **Greeting** 類別的結束。

編譯與執行 TestGreeting 程式

產生 TestGreeting.java 原始檔後，執行下列編譯命令：

```
javac TestGreeting.java
```

假如編譯器沒有回傳任何訊息，則會於同一個目錄下產生 TestGreeting.class。因為 TestGreeting 類別會使用到 Greeting 類別因此 Greeting.java 也自動編譯成 Greeting.class。要執行 TestGreeting 程式，需要使用 Java 直譯器。這些可執行的 Java 工具（javac, java, javadoc 等等）放置於 bin 目錄下。

```
java TestGreeting
```



注意 -- 您必須將 java_root/bin 加入 PATH 環境變數如此系統才找的到 java 和 javac 命令。其中 java_root 為 JDK 安裝的目錄。

解決編譯時期的問題

以下描述編譯程式時可能遭遇到的問題：

編譯時期錯誤

以下列舉出幾個編譯時期常見的錯誤以及顯示的訊息。這些訊息會因為 JDK 版本而有所差異。

- `javac: Command not found`
PATH 環境變數沒有適當的設定，無法找到 `javac` 命令。`javac` 位於 JDK 安裝目錄的 `bin` 目錄下。
- `Greeting.java:4:cannot resolve symbol`
symbol : method println (java.lang.String)
location: class java.io.PrintStream
`System.out.println("hi");`
 ^
`println` 方法名稱錯誤。
- 類別與檔案命名
假如 `.java` 檔為 `public class`，此時檔案的名稱必須和類別相同。例如：上一個例子中的類別
`public class TestGreeting`
則其程式原始檔必需命名為 `TestGreeting.java`。假如儲存為 `TestGree.java` 則編譯時會有下列錯誤訊息：
`TestGreet.java:4: Public class TestGreeting must be defined in a file called "TestGreeting.java".`
- 類別的數量
在一個程式原始檔中只能有一個頂層的 `non-static`、`public` 類別，而且必須和程式原始檔同名。假如有多個 `public` 類別時，編譯時會產生與上述相同的錯誤訊息。

執行時期錯誤

有些錯誤會在執行 `java TestGreeting` 時產生：

- **Can't find class TestGreeting**

這樣的訊息通常表示寫錯了命令列中的執行檔案名稱 `filename.class`。
Java 程式語言認為大小的字元並不相同。

舉例來說，

```
public class TestGreet {
```

產生了 `TestGreet.class` 檔案，非執行環境所預期的
(`TestGreeting.class`)。

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

這個訊息表示要執行的 Java 程式沒有 **static main** 的方法。也許程式
中有 `main` 方法但沒有宣告為 **static**，或是參數宣告錯誤例如：

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

這個例子中，`args` 是字串變數而非字串陣列變數。

```
public static void main() {
```

這個例子中，`main` 方法沒有參數。

圖 1-10 顯示如何編譯 Java 程式然後於 JVM 中執行。目前 JVM 有多個版本，以支援不同硬體與平台。

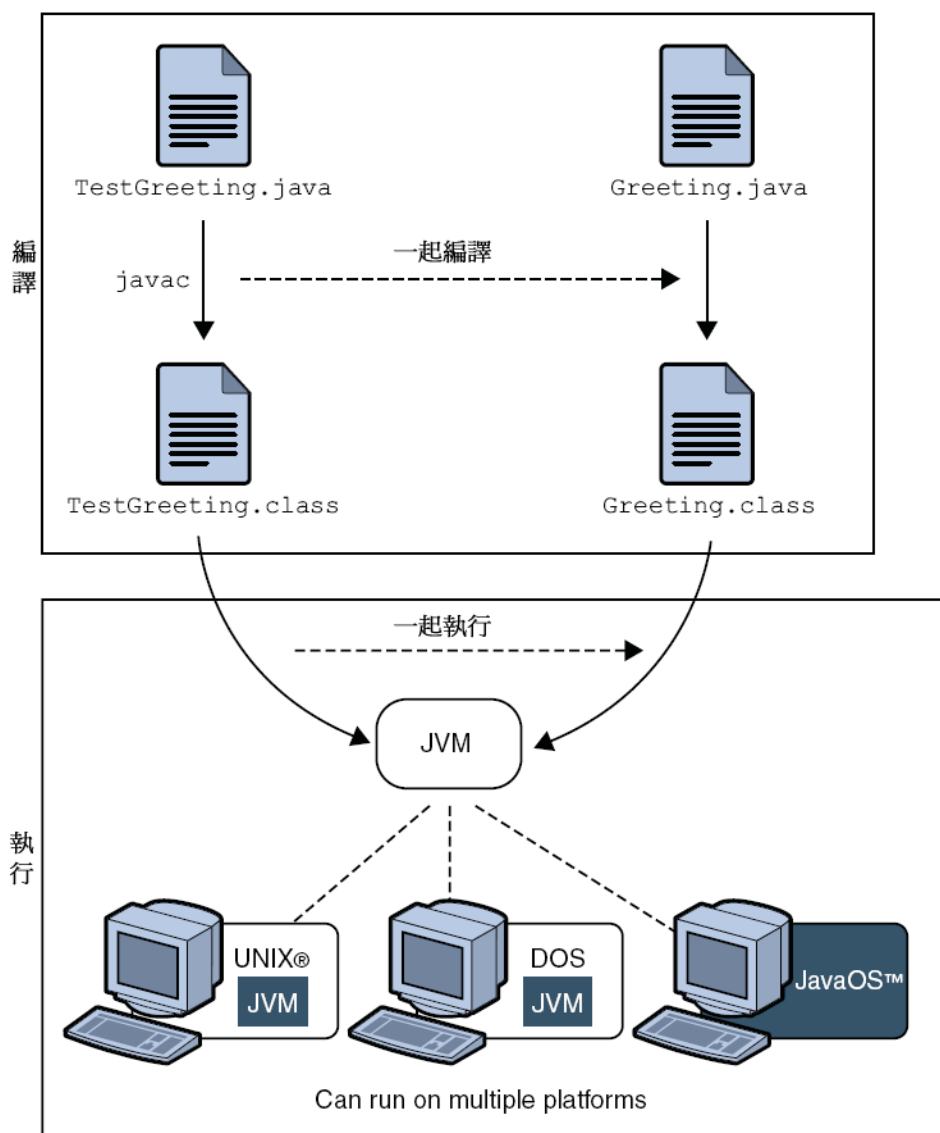


圖 1-10 Java 執行環境

章節 2

建立 Java 應用程式

單元重點

完成這個單元之後，您將能夠：

- 了解 Java 應用程式的組成
- 了解自定類別
- Java 應用程式靜態與動態觀點的差異
- 建構，使用及移除物件

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.

Java 應用程式的組成

一個房子是由多個房間以及一個（至少一個）玄關所組成。同樣的，一個 **Java** 應用程式是由多個類別以及一個被指定為主要的類別所組成。這個主要類別提供應用程式的進入點。圖 2-1 為房子設計與 **Java** 應用程式設計的比較。

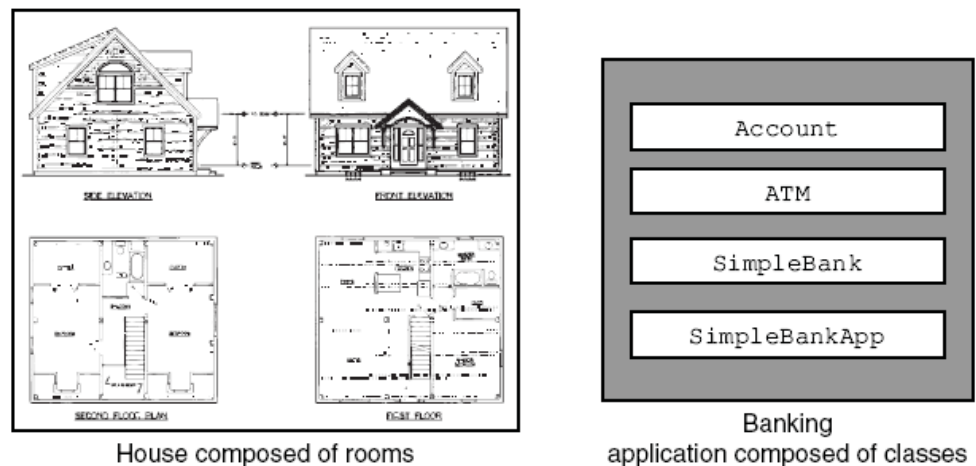


圖 2-1 應用程式組成概念展示圖

所有的房子都需要某種程度上的裝潢修改。你可以藉著改變房間來修改。大部分的案例，會依房間的需求選擇一些標準規格的产品來裝飾，變化數量或是擺設。例如裝飾廚房時會選擇櫥櫃，工作台，電器設備，以及其他的小器具。

圖 2-2 為一簡單的銀行應用程式自定類別。

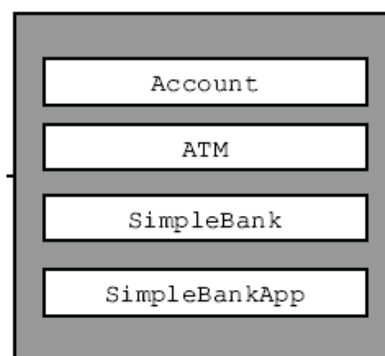


圖 2-2 銀行應用程式：自定類別

建立銀行應用程式的自定類別有：

- Account

Account 類別用來表示銀行帳戶。當應用程式需要產生一新的帳戶，就會透過 **Account** 類別產生一個 **Account** 物件（**Account** 實例）。

- **ATM**

ATM 類別用來表示自動提款機。在目前這個簡化過後的範例應用程式中只會產生單一的 **ATM** 物件。 **ATM** 實例提供了 **GUI** 介面 供銀行客戶申請帳戶（**Account** 實例）和使用（提款，存款或查詢存款金額）。

- **SimpleBank**

SimpleBank 類別用來表示簡化過後的銀行。 **SimpleBank** 類別定義了一組欄位和方法代表這個簡化版銀行的主要屬性以及其作業。

- **SimpleBankApp**

SimpleBankApp 類別是整個應用程式的進入點。

如同前面所述，銀行應用程式是由一組自定類別所組成。這些類別會使用標準 JDK 程式庫或是其他專案所建立的工具類別。圖 2-3 顯示了銀行應用程式所用到的幾個 JDK 類別。

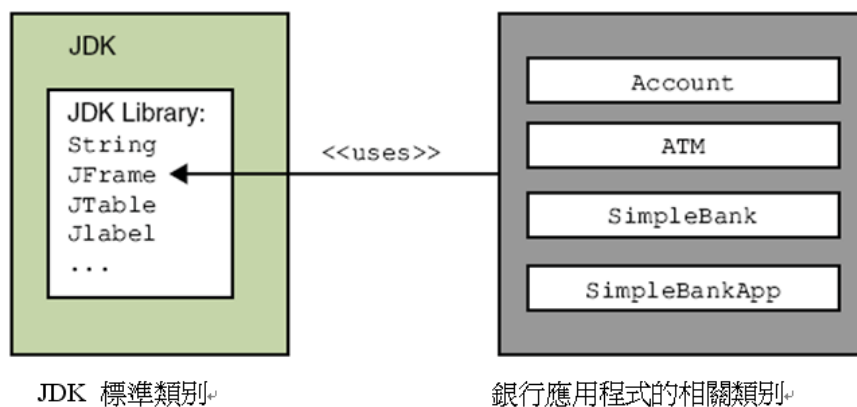


圖 2-3 銀行應用程式：自定與標準類別

檢視自定類別的建立過程：實作概念

以下會介紹建立類別的過程及概念。內容包含兩個部份：

- 類別的建立
- 應用程式的主類別

建立類別

以下透過問答方式介紹設計和宣告 **Java** 類別的概念。

- 類別代表什麼？
Java 類別代表一個型別的定義 (**class** 型別)。例如 **Account** 類別代表 (或是定義) 一銀行帳戶。同樣地，**JDK** 函式庫中 **Date** 類別代表 (或是定義) 一日期。
- 類別的實例代表什麼？
類別的實例代表類別的虛擬實體。例如，**Account** 類別的實例代表一虛擬的銀行帳戶實體。同樣地，**Date** 類別實例表示一虛擬的日期。每一個虛擬的實例包含狀態值。例如我的銀行帳戶狀態值與你的銀行帳戶狀態值有所不同，因而可以區別是你的帳戶還是我的帳戶。
- 什麼是物件？
物件同時具有狀態及行為。物件又稱為「類別實例」。
- 類別 (定義) 中具有哪些資訊？
類別具有兩種資訊：
 - 型別屬性
每個型別會有一組非必要的屬性。例如，**Account** 類別的屬性為 **customer** 和 **balance**。屬性是類別中的欄位，使用名稱和資料型別來宣告。表 2-1 為 **Account** 類別的屬性。

表 2-1 **Account** 類別的屬性

屬性名稱	屬性資料型別	註解	欄位宣告
customer	String	唯一的客戶帳號	String customer;
balance	double	目前客戶的存款總額	double balance;

- 型別操作 (operations)

操作是定義類別的方法 (methods)，例如：表 2-2 為 **Account** 的類別方法。

表 2-2 **Account** 類別的方法

宣告方法	註解
<code>double getBalance() { // ... }</code>	回傳目前客戶的存款總額
<code>void deposit(double sum) { // ... }</code>	將 sum 與存款總額相加
<code>void withdraw(double sum){ // ... }</code>	將存款總額扣除 sum
<code>String getCustomer(){ // ... }</code>	回傳客戶帳號
<code>String getDetails(){ // ... }</code>	回傳帳號詳細資料

- 什麼是統一塑模語言 (UML) 類別圖？

UML 類別圖定義一組記號及圖示來表示物件導向技術中的一些抽象概念，如類別，物件等等。以下將簡述幾個 **UML** 的記號：簡略類別圖，詳細類別圖以及物件圖。

一個類別圖代表一個類別。圖 2-4 為 **Account** 類別的 **UML** 詳細類別圖。

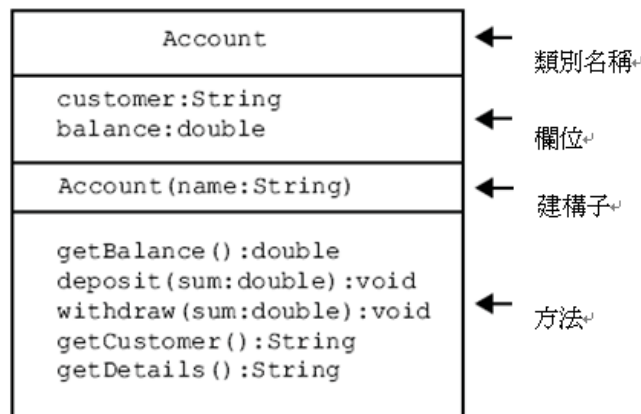


圖 2-4 **Account** 類別的 **UML** 詳細類別圖



注意 — 建構子和方法類似。兩者不同的地方為 **JVM** 只有於產生類別實例時才會呼叫建構子。通常建構子會包含將欄位初始化的程式。

圖 2-5 為 Account 類別的 UML 簡略類別圖。

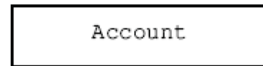


圖 2-5 Account 類別的 UML 簡略類別圖

圖 2-6 為 Account 類別實例的 UML 物件圖。物件圖表示類別的實例。

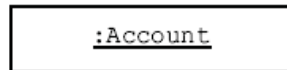


圖 2-6 Account 類別 UML 物件圖

- 什麼是 String 和 double 資料型別？

Account 類別使用到下列 Java 的資料型別：

- String 類別

String 類別定義在 java.lang 套件中。java.lang 為標準 JDK 函式庫的其中一個套件。

每一個 Account 物件含有一個 String 物件代表客戶的 ID。圖 2-7 顯示 Account 和 String 類別的關係。

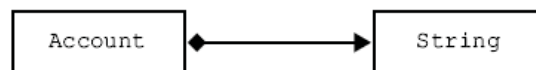


圖 2-7 UML 類別圖顯示 Account 和 String 的關係

- double 基本型別 (primitive)

double 資料型別不是一種類別。它是 Java 語言規範中 8 個基本資料型別的其中一個。8 種資料型別分別為：boolean, char, byte, short, int, long, float 以及 double。

- Java 類別檔案格式為何？

Java 類別有兩種檔案格式：

- 原始檔

原始檔為可閱讀的類別檔案格式。原始檔的副檔名為 java。例如：Account.java 和 SimpleBank.java。

- 類別檔

類別檔格式是 JVM 可解析的格式。類別檔的副檔名為 class。例如：Account.class 和 SimpleBank.class。

- 如何產生 Java 原始檔？

可以使用文字編輯器產生原始檔。程式 2-1 為 Java 原始檔的範例。

程式 2-1 Account 類別：原始檔內容

```
1  public class Account{
2      String customer;
3      double balance;
4
5      /** 產生新的 Account 實例 */
6      public Account(String name) {
7          customer = name;
8      }
9
10     public double getBalance() {
11         return balance;
12     }
13
14     public void deposit(double sum) {
15         if (sum > 0) {
16             balance = balance + sum;
17         }
18     }
19
20     public void withdraw(double sum) {
21         if ( sum <= balance && sum > 0) {
22             balance = balance - sum;
23         }
24     }
25
26     public String getCustomer() {
27         return customer;
28     }
29
30     public String getDetails() {
31         return "Type: " + "Account\n"
32             + "Customer: " + customer + "\n"
33             + "Balance: " + balance;
34     }
35 }
```

- 如何產生 Java .class 檔？

可以使用 Java 編譯器的 `javac` 命令編譯原始檔而產生 `.class` 檔。例如：圖 2-8 為 `javac` 命令的範例。.

```
javac Account.java
```

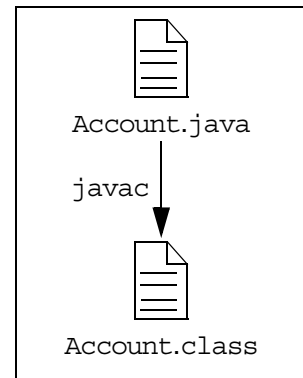


圖 2-8 使用 `javac` 命令

`javac` 命令工具列是 JDK 內建的工具之一。

了解應用程式的主類別 (Main Classes)

以下透過問答方式介紹 **Java** 應用程式中主類別的設計與實作要點。

- 應用程式中主類別的功能是什麼？

每個 **Java** 應用程式都需要一個主類別，它是程式的進入點，也可稱之為應用程式的啟始類別。

- 成為主類別的基本條件是什麼？

主類別必須有 **main** 這個方法，如下所示：

```
public static void main(String[] args) { // 程式由此開始 }
```

main 方法的實作內容則由應用程式決定。但原則上 **main** 方法會先產生應用程式所需的物件，例如，程式 2-2 顯示了應用程式的主要類別。

程式 2-2 銀行應用程式的主要類別

```
1 public class SimpleBankApp {
2     public static void main(String[] args) {
3         SimpleBank bank = new SimpleBank();
4         ATM atm = new ATM(bank);
5     }
6 }
```

- **main** 方法只能存在於主要類別嗎？

Main 方法主在提供 **Java** 應用程式的啟動器 (**java.exe**) 一個進入點。事實上任何類別都可以有 **main** 方法。通常在下列狀況下會加上 **main** 方法：

- 為了執行測試程式來測試其他類別。
- 開始應用程式。

- 開始執行應用程式的命令是什麼？

要開始執行應用程式你必須使用 **java** 這個命令工具，例如，由下列命令開始執行銀行應用程式：

```
java SimpleBankApp
```

這個命令會執行 **SimpleBankApp** 類別的 **main** 方法。

比較 Java 應用程式的靜態與動態觀點

Java 應用程式是物件導向的程式。物件導向的程式有兩個觀點：

- 靜態觀點
- 動態觀點

以下會介紹 Java 應用程式靜態與動態觀點。

靜態觀點

物件導向應用程式的靜態觀點指的是一群組成應用程式的類別。您可以將靜態觀點想成是當應用程式的類別都還儲存在檔案系統上，尚未執行時。圖 2-9 展示銀行應用程式靜態概念觀點。

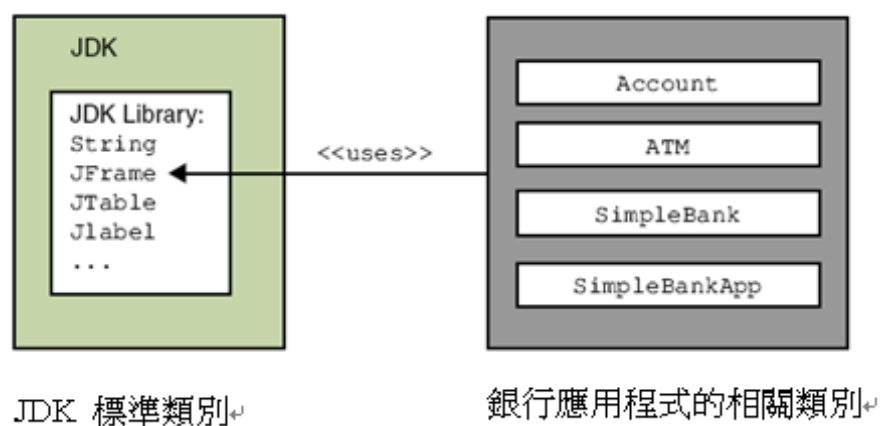


圖 2-9 銀行應用程式：靜態概念的觀點

身為一個 Java 程式撰寫者，主要的工作就是利用 JDK 提供的函式庫來產生應用程式所需的自定類別。開發過程中，我們將注意力放在類別設計上。

注意 ——Java 應用程式儲存成檔案時並不會互相組合連結，而是各自獨立成不同的檔案。



動態觀點

Java 應用程式的動態觀點主要是在執行時期時才能顯現。在執行時期，我們的焦點會放在類別的實體 - 物件上。特別是物件的建立，參考與使用。程式 2-3 為銀行應用程式主類別的 `main` 方法。

程式 2-3 銀行應用程式主類別

```

1 public class SimpleBankApp {
2     public static void main(String[] args) {
3         SimpleBank bank = new SimpleBank();
4         ATM atm = new ATM(bank);
5     }
6 }

```

1. 由下列命令列開始執行銀行應用程式：

java SimpleBankApp

上述的命令會執行 `SimpleBankApp` 類別的 `main` 方法並依序觸發下列幾個事件：.

2. JVM 會將執行 `SimpleBankApp` 所需的類別載入。
3. 執行到第 3 行會產生 `SimpleBank` 類別的實例。

```

3 SimpleBank bank = new SimpleBank();

```

圖 2-10 顯示了這個步驟所建立的物件。

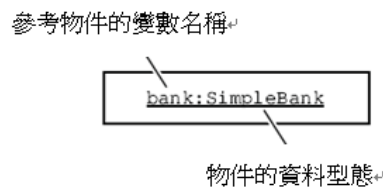


圖 2-10 UML 類別圖顯示 `SimpleBank` 物件

請注意下列的說明：

- SimpleBank 物件是由下列的程式所產生：

```
new SimpleBank()
```

要產生物件時，類別建構子必須與 **new** 關鍵字一起使用。JVM 會依據建構子的實作來初始物件的欄位值。建構子和類別同名稱。例如，SimpleBank 的建構子為 SimpleBank()。

- 下列的程式碼會宣告一個 SimpleBank 類別的物件參考，其名為 bank：

```
SimpleBank bank
```

物件參考包含名稱及其型別。在這個例子中參考的名稱為 **bank**，型別為 SimpleBank。物件參考其本身不是物件。但它會指向其所參考的物件。為了幫助分辨物件和參考，請聽聽下面的比喻。假如你有一個兒子，你兒子會叫你 **dad**。你的鄰居會叫你 **buddy**。你爸爸會叫你為兒子。對應起來你就是物件，但是有不同的名稱。緊跟著的派值 (assign) 運算子會將新產生的物件指派給 **bank**。

```
SimpleBank bank = new SimpleBank();
```

- 下列的程式碼會產生 ATM 類別的實例。

4 ATM atm = new ATM(bank);

圖 2-11 顯示了這個步驟產生的物件結果。



圖 2-11 UML 類別圖展示 ATM 和 SimpleBank 物件

請注意下列的說明：

- ATM 實例經由下列程式碼產生：

```
new ATM(bank)
```

這個例子中，ATM 的建構子接收 SimpleBank 物件的參考，也就是 **bank**。

- 下列程式碼宣告一個 ATM 的參考：

```
ATM atm
```

緊跟著的派值運算子會將新產生的 ATM 物件指派給 **atm**。

```
ATM atm = new ATM(bank);
```

5. 執行 ATM 類別的建構子（未列出程式碼）會產生很多物件。其間最重要的是模擬自動提款機的圖形介面物件。圖 2-12 顯示了銀行應用程式 ATM 的使用者介面

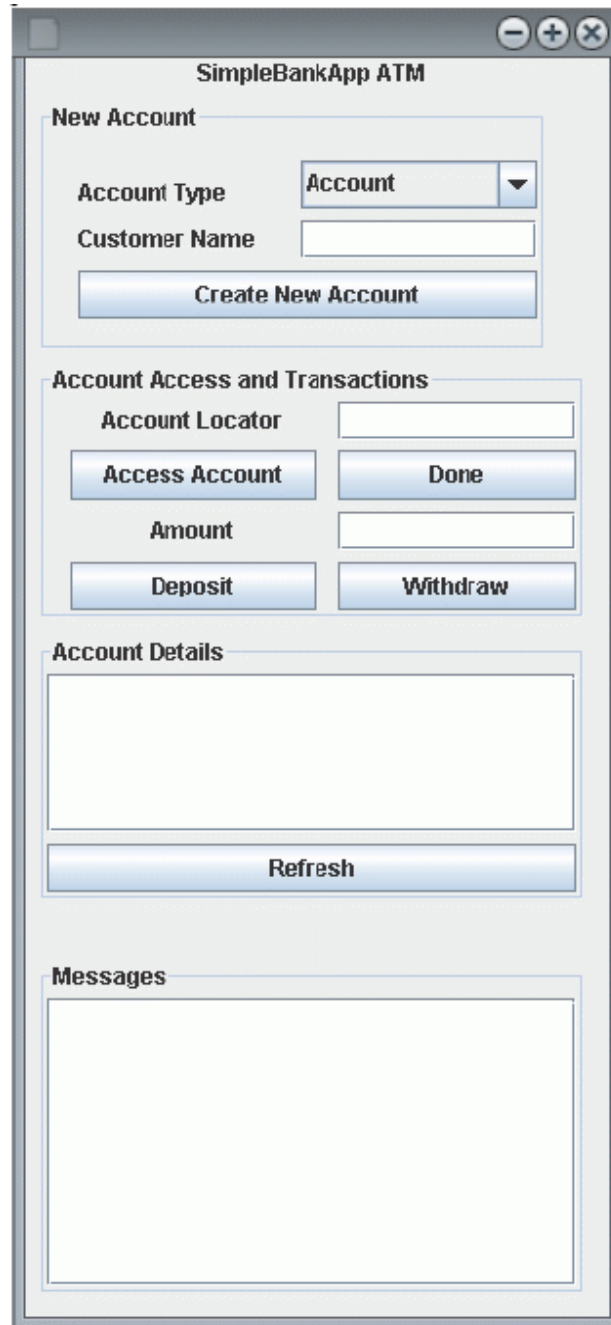


圖 2-12 銀行應用程式 ATM 使用者介面

6. 使用 ATM 介面，銀行應用程式的使用者可進行下列操作：
- 建立新帳戶

建立新帳戶的服務是由 **SimpleBank** 的實例所提供。建立新帳戶會產生一個 **Account** 實例。 **SimpleBank** 實例用來管理帳戶物件。



注意 -- 在一般情況下，建立新帳戶是由銀行行員來做而非 **ATM**。

- 存取已存在的帳戶資訊

透過輸入客戶的資訊，使用者可經由 **SimpleBank** 實例取得帳戶資料。

- 對帳戶操作

Account 實例具有下列的操作：

- 存款

使用者可以存款。這個服務是由 **Account** 實例的 **deposit** 方法提供。例如，要存入 50 元到目前的帳戶，輸入 50 到 **amount** 欄位然後按存款按鈕。按下存款按鈕後，類似下列的程式會被執行：

```
account.deposit(50.0);
```

account 變數則是參考至目前要存款的帳戶。請檢視程式 2-1 中 **Account** 類別所宣告的 **deposit** 方法。

- 提款

使用者可以提款。這個服務是由 **Account** 實例 **withdraw** 方法提供。假設輸入 10 到 **amount** 欄位，按下提款按鈕後，類似下列的程式會被執行：

```
account.withdraw(10.0);
```

其中 **account** 變數則是參考至目前要提款的帳戶。請檢視程式 2-1 中 **Account** 類別所宣告的 **withdraw** 方法。

- 取得帳號詳細資訊

使用者可以取得帳戶詳細資訊。這個服務是由 **Account** 實例的 **getDetails** 方法提供。使用者按下重新整理的按鈕後，類似下列的程式會被執行：

```
String details = account.getDetails();
```

其中 **account** 變數則是參考至目前取得之帳戶。 **ATM** 的使用者介面會顯示帳戶的詳細資訊。請檢視程式 2-1 中 **Account** 類別所宣告的 **getDetails** 方法。

- 儲存帳戶資訊

這個服務是由 **SimpleBank** 實例自動執行。

建立，使用及移除物件

執行 **Java** 應用程式過程中，應用程式會建立物件，移除物件，與物件互動。以下透過問答方式說明 **Java** 應用程式對於物件的管理。以下會以程式 2-1 中的 **Account** 類別為例來討論。圖 2-13 為 **Account** 類別的 UML 類別圖。

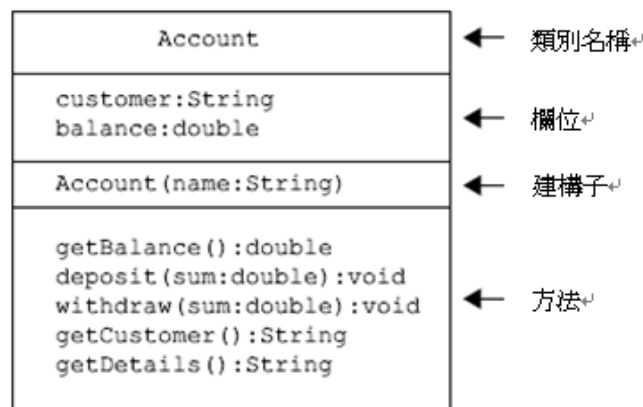


圖 2-13 Account 類別的 UML 類別圖

- 在類別圖上哪些資訊與物件結構有關？

雖然 **Java** 物件記憶體使用大小取決於實作本身，但 **JVM** 會分配足夠的記憶體存放類別的欄位資料。這個例子中 **Account** 類別的 **balance** 欄位是 **double**，**customer** 是 **String** 類別。**JVM** 分配足夠的記憶體存放 **double** 的值。至於 **customer** 欄位，**JVM** 分配記憶體存放參考值。簡單來說參考值就是會指向 **String** 物件的位址。圖 2-14 為記憶體中的 **Account** 物件。

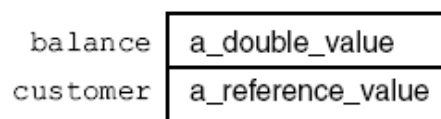


圖 2-14 Account 物件：記憶體使用狀況

- 類別圖上哪些資訊與物件的建構有關？

類別圖顯示 **Account** 類別宣告了一個建構子 **Account(name:String)**。

建構子會建立指定的物件，並初始化欄位，回傳新建立物件的參考。你必須指派這個物件的參考，否則將無法使用這個物件，最後會被從記憶體移除（垃圾回收）。程式 2-4 顯示物件的產生以及指向該物件的參考。

程式 2-4 建構子使用範例

```
Account myAcct; // 宣告參考的變數
myAcct = new Account("Me");
```

程式 2-4 顯示使用 **new** 關鍵字以及 **Account** 類別建構子產生 **Account** 物件。圖 2-15 顯示程式 2-4 執行後記憶體的結果。

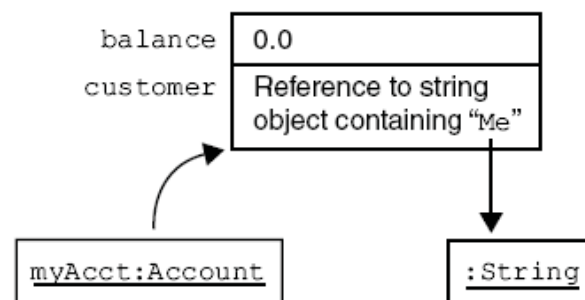


圖 2-15 被參考的 **Account** 物件

- 類別圖上哪些資訊與物件的互動性有關？

Account類別圖顯示**Account**類別宣告兩個欄位 (**balance** 和**customer**) 以及五個方法 (**getBalance**, **deposit**,**withdraw**, **getCustomer** 和 **getDetails**)。

要跟物件互動你必須：

- 物件的參考
- 物件可存取的成員

Account 類別圖顯示 **Account** 類別宣告兩個欄位 (**balance** 和 **customer**) 以及五個方法 (**getBalance**, **deposit**,**withdraw**, **getCustomer** 和 **getDetails**)。

程式 2-5 展示使用 ‘.’ 運算子與物件互動

程式 2-5 與物件互動的範例

```
1 Account myAcct; // 宣告參考變數
2 myAcct = new Account("Me"); // 產生物件
3 myAcct.balance = 200.0; // 指定欄位值
4 myAcct.deposit(300.0); // 呼叫方法
5 myAcct.withdraw(100.00); // 呼叫方法
6 double myBalance; // 宣告基本型態變數
```

```
7 myBalance = myAcct.getBalance();
```

- 是否可以使用多個參考值參考同一個物件？
可以使用多個參考值參考同一個物件，如程式 2-6 所示。

程式 2-6 使用多個參考值

```
1 Account myAcct; // 宣告參考變數
2 myAcct = new Account("Me"); // 產生物件
3 myAcct.deposit(200); // 使用 myAcct 的 deposit 方法
4 Account myChildAcct;
5 myChildAcct = myAcct;
6 myChildAcct.withdraw(300.0);
```

- 參考值一旦被指派，可以再指派物件嗎？
可以再指派參考值指向另一同類別的物件。如程式 2-7 所示。

程式 2-7 再指派參考值

```
1 Account myAcct; // 宣告參考變數
2 myAcct = new Account("Me"); // 產生物件
3 myAcct.deposit(200); // 使用 myAcct 的 deposit 方法
4 Account myChildAcct;
5 myChildAcct = myAcct;
6 myChildAcct.withdraw(300.0);
7 myChildAcct = new Account("Child");
8 myAcct = myChildAcct;
```

- 物件一旦建立後要如何移除？

Java 提供記憶體回收機制，移除未被使用的物件。只要物件不再被參考到（物件沒有任何參考變數指向它），那麼它就是被回收的候選者。這些物件會在掃除的階段將由記憶體移除。例如，表 2-3 顯示了程式 2-7 中變數 `myAcct` 和 `myChildAcct` 所參考物件的歷史紀錄。

表 2-3 程式 2-7 物件參考狀況

行數	參考變數	參考物件
1	<code>myAcct</code>	沒有
2	<code>myAcct</code>	<code>Account("Me")</code>
4	<code>myAcct</code> <code>myChildAcct</code>	<code>Account("Me")</code> 沒有
5	<code>myAcct</code> <code>myChildAcct</code>	<code>Account("Me")</code> <code>Account("Me")</code>

表 2-3 程式 2-7 物件參考狀況

行數	參考變數	參考物件
7	myAcct myChildAcct	Account ("Me") Account ("Child")
8	myAcct myChildAcct	Account ("Child") Account ("Child")

執行程式 2-7 第 8 行程式後，Account 物件 Account ("Me") 會成為記憶體回收的候選者。記憶體回收器以背景程式的模式週期性地被執行。執行的頻率依 JVM 版本而異。你可以對應用程式設定記憶體回收參數。然而，目前版本的 JVM 已經內建動態最佳化的設計。記憶體回收機制的設定我們不在本章節討論。

建立類別：欄位與建構子語法

單元重點

完成這個單元後，您將能夠：

- 了解如何宣告類別
- 宣告物件欄位，並透過預設值、明確設值及構構子等機制來初始化物件欄位
- 了解程式中的註解，空白值和關鍵字
- 了解目錄結構與套件

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O’ Reilly Media. 2005.

類別宣告簡介

類別宣告旨在定義一個新型別。以下說明自訂 **Java** 類別的步驟：

1. 決定新類別的套件名稱
2. 宣告新類別使用到的外部類別
3. 宣告類別的欄位
4. 宣告建構子（如果有的話）與並將欄位值初始化
5. 宣告類別的方法

決定新類別的套件 (package) 名稱

套件是 **Java** 用來組織類別的機制。套件提供了命名空間給其所包含的類別。屬於同一個套件的類別會儲存於同一個目錄。因此一群相關的類別會編整於同一個套件。程式 3-1 是宣告套件名稱的範例。

程式 3-1 宣告類別的套件名稱

```
1 package finance; // package statement
2
3 class Stock {
4     // Internals of the class declarations not shown for clarity reason
5 }
```



注意 — 在開發產品時，應使用 **URL** 的命名慣例來定義套件名稱。例如：
package com.mycompany.myproject.finance;

程式 3-1 中 **Stock** 類別定義於 **finance** 套件下。請您注意下列與 **package** 敘述有關的資訊：

- 宣告類別套件所用的 **package** 敘述（程式 3-1 第一行）必須是除了註解之外，第一個出現的敘述。
- 假如沒有 **package** 敘述，類別就歸屬於預設套件 (**default package**)。為了避免類別名稱重複出現，最好還是使用套件。
- 程式 3-1 中 **Stock** 的類別全稱為 **finance.Stock**。



注意 — 3-3 頁程式 3-1 中緊跟在 `//` 後的文字為註解。

宣告新類別使用的外部類別

外部類別指的是任何與新類別位於不同套件的類別。使用 `import` 敘述來宣告所引用的類別。例如，程式 3-2 第 3 行，`Stock` 類別引用了 `java.util` 套件下的 `Date` 類別。

程式 3-2 宣告引用類別範例

```
1 package finance;
2
3 import java.util.Date;
4 // 如果需要可以繼續加入其它的 import 敘述
5
6 class Stock {
7     // stock 類別的實作
8 }
```

注意下列與 `import` 敘述相關的資訊：

- `import` 敘述宣告的地方位於套件宣告和類別宣告之間。
 - 若要引用的類別位於同一個套件下則不需使用 `import` 宣告。不同套件下的類別則必須使用 `import`。
- `import` 敘述的替代方案是使用類別全稱。例如，程式 3-2 中若不使用 `import` 敘述，則出現 `Date` 類別的地方要修改為使用完整的類別名稱 `java.util.Date`。
- `java` 提供引用某個套件下所有類別的語法。例如，如果 `Stock` 類別使用到 `java.util` 套件下多個類別，則可以使用下列敘述引用類別：
`import java.util.*;`
 - 若是用到 `java.lang` 套件下的類別則不需宣告，因為這個套件已被自動引用。

宣告類別欄位

程式 3-3 展示 **Stock** 類別宣告了 3 個欄位。每個欄位代表 **Stock** 資料型別物件的屬性（或狀態）。在方法或建構子之外宣告的變數就是宣告類別的欄位。

程式 3-3 類別中宣告欄位的範例

```

1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6      // Field declarations
7      String symbol;
8      double price;
9      Date date;
10
11     // 建構子的宣告
12     // 方法的宣告
13
14 }
```

注意下列與欄位宣告相關的資訊：

- 下列的語法為宣告欄位最簡單的一種：

data-type identifier;

例如：

double price;

欄位的資料型別可以為 **Java** 基本資料型態；如 **double**，或是類別；如 **Date**。**Java** 資料型別會在「宣告欄位：類別欄位」有詳細的描述。識別子在「宣告欄位：變數識別名稱的格式」中有詳細說明。

- 下列語法可將欄位初始化：

data_type identifier = initial_value;

例如：

double price = 25.50;

- 下列語法可以同時宣告多個相同資料型別的欄位：

data_type identifier1, identifier2, identifier3;

例如：

Date birthDay, anniversary;

宣告類別建構子

建構子可以動態指定物件欄位的初始值。程式 3-4 是 **Stock** 類別的建構子。在單一類別中可以有多個建構子。

程式 3-4 宣告類別的建構子

```
1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6      // 宣告欄位
7      String symbol;
8      double price;
9      Date date;
10
11     // 宣告建構子
12     Stock(String stockSymbol) {
13         symbol = stockSymbol;
14     }
15
16     // 宣告方法
17
18 }
```

本節僅討論建構子的宣告位置。在「使用建構子將欄位初始化」一節中會有更深入的說明。

宣告類別方法

方法實作了一個類別的行為。程式 3-5 是在 **Stock** 類別中宣告方法的方式。一個類別可以包含多個方法。

程式 3-5 類別方法的宣告範例

```
1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6      // 宣告欄位
7      String symbol;
8      double price;
9      Date date;
10
11     Stock(String stockSymbol) {
12         symbol = stockSymbol;
13     }
14
15     void updatePrice (Date updateTime, double newPrice) {
16         date = updateTime;
17         price = newPrice;
18     }
19
20     double getSymbol() {
21         return symbol;
22     }
23
24     double getPrice() {
25         return price;
26     }
27
28     Date getDate() {
29         return date;
30     }
31 }
```

我們在此僅說明可以宣告方法的位置，下節會對方法宣告的方式做更進一步的討論。

宣告欄位：基本資料型別

Java 程式語言有很多內建的資料型別，可分為兩大類：

- 基本型別 - 相對於物件而言它是簡單的值。
- 類別型別 - 用於複雜型別，包括自定型別。類別是物件建構的樣板。

Java 程式語言定義了 8 種基本資料型別，可分為 4 類：

- 邏輯 - **boolean**(布林值)
邏輯的值是用 **boolean** 表示，**boolean** 有兩種值：**true** 和 **false**。
- 文字 - **char**
單一字元使用 **char** 表示，一個 **char** 使用 16 位元，無正負號的萬國碼 (unsigned Unicode) 來表示。
- 整數 - **byte**, **short**, **int**, 和 **long**
Java 程式語言有 4 種整數型別，分別使用 **byte**, **short**, **int**, 和 **long** 等關鍵字。
- 浮點數 - **double** 和 **float**
浮點數型別有兩種，分別是 **float** 和 **double**，浮點數的表示方式依照 **IEEE 754** 制定的規範，如表 3-1 所示。該格式是跨平台的。表 3-1 為 8 種基本型別的比較表。

表 3-1 基本資料型別比較表

基本資料型別	分類	整數或浮點數長度	範圍
boolean	邏輯	不適用	true , false
char	文字	16 bits	0 to $2^{16}-1$
byte	整數	8 bits	-2^7 to $2^7 - 1$
short	整數	16 bits	-2^{15} to $2^{15} - 1$
int	整數	32 bits	-2^{31} to $2^{31} - 1$
long	整數	64 bits	-2^{63} to $2^{63} - 1$
float	浮點數	32 bits	
double	浮點數	64 bits	

基本資料型別的 literals

表 3-2 列出了宣告基本資料型別 literals 的語法。.

表 3-2 基本資料型別 literals 的語法

基本資料型別	語法與範例	備註
boolean	true, false	boolean 只能使用 true 或 false.
char	'a' '\t' '\u????'	字元 a.. tab 字元. 特定的 Unicode 字元，???? 代表為 4 個十六進位值。 例如： ' \u03A6' 為希臘字的 phi [?].
byte	5	必須在 Int 數值範圍內 .
short	300	必須在 Int 數值範圍內
int	2 077 0xBAAC	十進位的 2. 0 開頭表八進位 0x 開頭表十六進位 所有整數值預設為 Int 型別
long	2L 077L 0xBAACL	L 或 l 表十進位的長整數值 2 0 開頭表八進位 0x 開頭表十六進位
float	52.56f 32.0F	使用 f 或是 F 表示 float 型態
double	100.25 100.25d 100.25D	浮點數預設型態為 double，除非直接指定。可用 d 或 D 代表 double.

宣告欄位：類別欄位

類別型別 (**Class types**) 用來組成較複雜的資料型別，包含自訂類別也屬於類別型別。類別型別是建構物件的樣板，有下列幾個來源：

- **Java** 標準類別函式庫 - 包含在 **JDK** 開發工具中。每次新版的 **JDK** 都會有新的類別加入。
- 商業版或是開放原始碼函式庫 - 由廠商或是開放原始碼的社群獲得。
- 公司或使用者自行寫作的類別 - 由個人於參加專案或是您所屬公司、組織中所自行開發的類別。

Java 標準類別函式庫

本節簡單介紹幾個標準函式庫中的類別。表 3-3 列出了幾個常用的類別。

表 3-3 標準函式庫中的類別

套件名稱	類別範例	目的
java.lang	Enum, Float, String, Object	基本類別
java.util	ArrayList, Calendar, Date	工具類別
java.io	File, Reader, Writer	支援輸出輸入
java.math	BigDecimal, BigInteger	支援高精準度的數學運算
java.text	DateFormat, Collator	支援文字處理與格式化
javax.crypto	Cipher, KeyGenerator	支援加解密
java.net	Socket, URL, InetAddress	網路程式類別
java.sql	ResultSet, Date, Timestamp	支援結構化查詢語言 (SQL)
javax.swing	JFrame, JPanel	支援圖形介面 (GUI)
javax.xml.parsers	DocumentBuilder, SAXParser	支援延伸標籤語言 (XML)

以下會將焦點集中在 **String** 類別以及包覆 (wrapper) 類別 (每一個基本型別都具有一個對等的包覆類別)。

String 類別

String 不是基本型別而是類別，它代表一連串的字元 (**char**)。這些字元都是使用萬國碼 (**Unicode**)。表 3-2 中使用在 **char** 型別的反斜線記號也適用於 **String**。

String 的 **literal** 為使用兩個雙引號所包含的字串：

```
"The quick brown fox jumps over the lazy dog."
```

下列為 **String** 型別變數的宣告和初始化範例：

```
// 宣告並將 String 初始化
String greeting = "Good Morning !! \n";
String errorMessage = "Record Not Found !";
```

```
// 宣告 2 個 String 型別變數
String str1, str2;
```

即使像上面這樣，只宣告 **String** 的 **literal**，就會導致系統建立一個新的 **String** 物件，您也可以使用 **new** 關鍵字來產生 **String** 物件。例如：

```
// 使用 new 關鍵字產生 String 物件
String s1 = new String("Hello");
```

包覆 (Wrapper) 類別

由於執行效能的原因，**Java** 程式語言並未將基本資料型別視為物件。例如：布林值，字元等都被視為基本型別。**Java** 程式語言也提供包覆類別將基本資料型別的資料包覆起來，如此一來就可以透過物件的方式來處理基本型別中的資料。**Java.lang** 套件中包含了每個 **Java** 基本型別的包覆類別。每個包覆類別的物件都包含有一基本型態的值（請參閱表 3-4。）

注意 — 這些包覆類別都是 **immutable** 物件，這表示一旦指定了包覆類別的值，就無法再改變。



表 3-4 包覆類別

基本資料型別	包覆類別
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float

表 3-4 包覆類別

基本資料型別	包覆類別
double	Double

您可以利用適當的建構子將所要包覆的值做為建構子參數傳入包覆物件中。
如程式 3-6 所示：

程式 3-6 使用包覆類別範例

```
int pInt = 420;  
Integer wInt = new Integer(pInt); // 稱為 boxing  
int p2 = wInt.intValue(); // 稱為 unboxing
```

包覆類別提供很多相關的方法來處理資料轉換的問題，例如：

```
int x = Integer.valueOf(str).intValue();  
或：  
int x = Integer.parseInt(str);
```

宣告欄位：識別名稱 (Identifiers) 的格式

Java 程式語言中識別名稱指的是變數，方法，類別等的名稱。識別名稱可以包含英文字母，下底線 (`_`) 或是以 `$` 號為開頭，後面可以接數字。識別名稱有大小寫之分，但沒有長度限制。

下列為有效的識別名稱範例：

- `identifier`
- `userName`
- `user_name`
- `_sys_var1`
- `$change` // 合法但不建議

Java 使用 16 位元的萬國碼，而非 **ASCII**，所以字母並不只是 `a - z` 和 `A - Z` 而已。

使用非 **ASCII** 字元作為識別名稱時，請注意下列事項：

- 萬國碼支援多國語言，有些字元看起來相同的但實際上是不同字元
- 大部分的作業系統不支援萬國碼的字元檔名，所以類別名稱最好只用 **ASCII**

Java 語言的關鍵字不可以成為識別名稱，但是可以為其名稱的一部分。例如：`thisOne` 是有效的識別名稱，而 `this` 不是。因為 `this` 是 **Java** 的關鍵字。



注意 — 識別名稱通常不會包含 `$` 號。雖然有些語言廣泛的使用它，但出現在程式中會有些奇怪。因此最好避免使用，除非有特別的使用慣例或其他原因。

宣告欄位：初始化欄位

本節將說明指派初始值給新產生物件欄位（屬性）的程序。例如：程式 3-7 **Stock** 類別具有 3 個欄位。本節將會說明有哪些機制可以在產生物件後，指定欄位的初始值。

程式 3-7 **Stock** 類別欄位初始化的範例

```
1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6      // 欄位宣告
7      String symbol;
8      double price;
9      Date date = new Date();;
10
11     // 建構子宣告
12     Stock(String stockSymbol) {
13         symbol = stockSymbol;
14     }
15
16     // 方法宣告
17
18 }
```

Java 提供下列將欄位值初始化的機制：

- 使用預設值
- 明確指定欄位的值
- 使用建構子

接下來會對以上三種方式提供進一步的說明。

使用預設值將欄位初始化

請看程式 3-8 的 **Stock** 類別。此類別沒有建構子。所有欄位宣告時都未指定值。這種狀況下，每個欄位會使用該型別的預設值。依型別不同會有不同的預設值。

程式 3-8 使用預設值進行初始化的範例

```

1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6      // 欄位宣告
7      String symbol; // 初始值為 null
8      double price; // 初始值為 0.0
9      Date date; // 初始值為 null
10
11     // 沒有建構子宣告
12
13     // 方法宣告
14
15 }
```

表 3-5 為 **Java** 的基本資料型別預設值。

表 3-5 基本型別預設值列表

資料型態	預設值
boolean	false
byte	0
char	'\u0000'
short	0
int	0
long	0L
float	0.0f
double	0.0d
Any class type	null

明確指定欄位的初始值

請看程式 3-9 的 Stock 類別

程式 3-9 明確指定欄位初始值的範例

```
1 package finance;
2
3 import java.util.Date;
4
5 class Stock {
6     // 欄位宣告
7     String symbol;
8     double price;
9     Date date = new Date();
10
11     // 沒有建構子
12
13     // 方法宣告
14
15 }
```

這個類別沒有建構子。宣告了一個欄位且具有初始值。表 3-6 為執行下列程式，**Stock** 實例的欄位初始值列表：

```
Stock stock = new Stock();
```

表 3-6 Stock 實例欄位初始值列表

欄位名稱	初始值	註解
symbol	null	預設值
price	0.0	預設值
date	參考至新的 Date 物件。此 Date 物件具有 時間與日期。.	明確指定初始值

使用建構子將欄位初始化

請參閱程式 3-10 **Stock** 類別。

程式 3-10 具有單一建構子的 **Stock** 類別

```

1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6      // 欄位宣告
7      String symbol;
8      double price;
9      Date date = new Date();
10
11     // 建構子宣告
12     Stock(String stockSymbol) {
13         symbol = stockSymbol;
14     }
15
16     // 省略方法的宣告
17
18 }
```

Stock 類別宣告一建構子。宣告了一個欄位且具有初始值。表 3-7 為執行下列程式，**Stock** 實例的欄位初始值列表：

```
Stock stock1 = new Stock("SUNW");
```

表 3-7 **Stock** 類別的 **Stock1** 實例欄位初始值列表

欄位名稱	初始值	備註
symbol	指向字串 SUMW 的參考	建構子中的設值動作覆寫了初始值
price	0.0	使用預設值
date	參考至新的 Date 物件。此 Date 物件具有 時間與日期。	明確指定初始值

建構子裏面包含了一連串用來將類別實例初始化的指令。此外也可以傳遞參數給建構子，就像使用方法時傳參數一樣。最簡單的宣告格式如下：

```
<class_name> ( <argument>* ) {
    <statement>*
```

```
}
```

建構子的名稱必須和類別名稱相同。〈argument〉參數宣告方式和方法的參數宣告方式一樣。



注意 — 建構子不是方法。他們不會回傳任何值。建構子在使用 **new** 關鍵字時才會被呼叫。

建構子多載 (overloading)

當要將物件初始化時，程式可以提供多組建構子來為物件設定初值。例如：程式 3-11 的 **Stock** 類別擁有兩個建構子。

程式 3-11 **Stock** 類別擁有兩個建構子

```
1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6  // 欄位宣告
7      String symbol;
8      double price;
9      Date date = new Date();
10
11 // 建構子宣告
12     Stock(String stockSymbol) {
13         symbol = stockSymbol;
14     }
15
16     Stock(String stockSymbol, double newPrice) {
17         this(stockSymbol); // 呼叫另一個建構子
18         price = newPrice;
19     }
20
21 // 方法宣告
22
23 }
```

表 3-8 為執行下列程式時，**stock1** 物件的欄位初始值列表：

```
Stock stock1 = new Stock("ENCY");;
```

表 3-8 stock 類別的 stock1 實例欄位初始值列表

欄位名稱	初始值	註解
symbol	指向字串 ENCY 的參考	建構子中的設值動作覆寫了初始值
price	0.0	預設值
date	參考至新的 Date 物件。此 Date 物件具有 時間與日期。	明確指定初始值

表 3-9 為執行下列程式，stock 類別 stock2 實例的欄位初始值列表：

```
Stock stock2 = new Stock("ZUMZ", 52.00);
```

表 3-9 stock 類別的 stock2 實例欄位初始值列表

欄位名稱	初始值	註解
symbol	指向字串 ZUMZ 的參考	建構子中的設值動作覆寫了初始值
price	52.00	建構子中的設值動作覆寫了初始值
date	參考至新的 Date 物件。此 Date 物件具有 時間與日期。	明確指定初始值

預設建構子

每個類別至少有一個建構子。假如你沒有宣告建構子，**Java** 會提供一個預設建構子。預設建構子沒有參數，也沒有任何實作。程式 3-12 的 **Stock** 類別使用了預設建構子

程式 3-12 使用預設建構子的範例

```
1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6      // Field declarations
7      String symbol;
8      double price;
9      Date date;
10
11     // 沒有宣告建構子
12
13     // 沒有宣告方法
14
15 }
```

有了預設建構子就可以使用下列方式產生 **Stock** 物件：`new Stock();`。



注意 — 假如 **Stock** 類別中已宣告任一建構子，就不能使用預設建構子。例如程式 3-10 的 **Stock** 類別就無法使用預設建構子，若加入 `new Stock()` 敘述，在編譯時期就會產生錯誤，除非開發人員自行定義一個沒有參數的建構子。

你可以自定一無參數建構子取代預設建構子，如程式 3-13 所示。

程式 3-13 **Stock** 類別自訂無參數建構子

```
1  package finance;
2
3  import java.util.Date;
4
5  class Stock {
6      // 欄位宣告
7      String symbol;
8      double price ;
9      Date date;
10
11     // 建構子宣告
12     Stock() {
13         date = new Date();
14     }
15
16     // 為求簡潔不列出宣告之方法
17
18 }
```

註解，空白和關鍵字

這節將會介紹下列三個元素在 **Java** 中使用的規則以及相關的資訊：

- 註解
- 空白
- 關鍵字

註解

註解格式有三種：

// 單行註解

```
/* comment on one  
* 多行註解  
*/
```

```
/** documentation comment  
* 文件註解，可多行  
*/
```

文件註解置放於變數、方法及類別宣告之前，這些註解可以用來自動產生文件（例如：使用 **javadoc** 命令來自動產生 **HTML** 格式的文件檔）。



注意 — **Javadoc™** 工具使用的註解格式在 **JDK** 文件中有詳細說明。您也可以使用線上連結 <http://java.sun.com/javase/6/docs/technotes/guides/javadoc/>。

空白

你可以在程式碼中的兩個元素間加入空白。空白長度沒有限制。可以使用 **tabs**、**spaces**、或是換行等方式來增加程式的可讀性。請您比較下列兩種寫法：

```
{int x;x=23*54;}
```

with:

```
{
    int x;

    x = 23 * 54;
}
```

Java 程式語言的關鍵字

Java 程式語言的關鍵字由 Java 程式語言規範來定義。關鍵字對於 Java 編譯器來說具有特殊的意義，透過它們可以識別出資料型別和程式結構名稱。

表 3-10 為 Java 程式語使用的關鍵字列表。

表 3-10 Java 程式語言所使用的關鍵字

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

下列為使用關鍵字要注意的事項：

建立類別：欄位與建構子語法

Copyright 2007 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G

Unauthorized reproduction or distribution prohibited. Copyright © 2010, Oracle and/or its affiliates.

- 和 **C++** 一樣，**true**, **false**, 和 **null** 等關鍵字都是小寫不能使用大寫。嚴格來講對於 **Java** 來說，這些不是關鍵字而是一固定值，但無論如何只有在學術上才會這樣區分。
- 沒有 **sizeof** 運算子
- **goto** 和 **const** 關鍵字在 **Java** 中不能使用。

目錄結構與套件

套件被儲存為目錄樹狀結構中，而每個子目錄則是套件名稱。例如圖 3-1 所示 **Company.class** 所在的目錄結構。

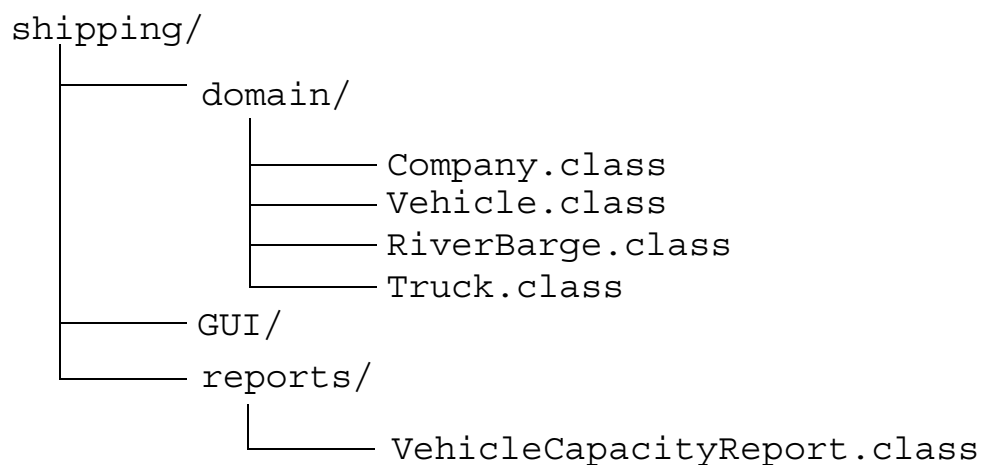


圖 3-1 範例程式的套件目錄樹狀結構

開發時期的目錄結構

同時開發多個專案是常見的情形。有很多種方法來管理這些開發中的檔案。以下會介紹其中一種常見的管理方式。

圖 3-2 為專案開發時的檔案結構範例。在這種結構中有個重點：將原始檔與編譯檔 (.class) 分別儲存於不同目錄。

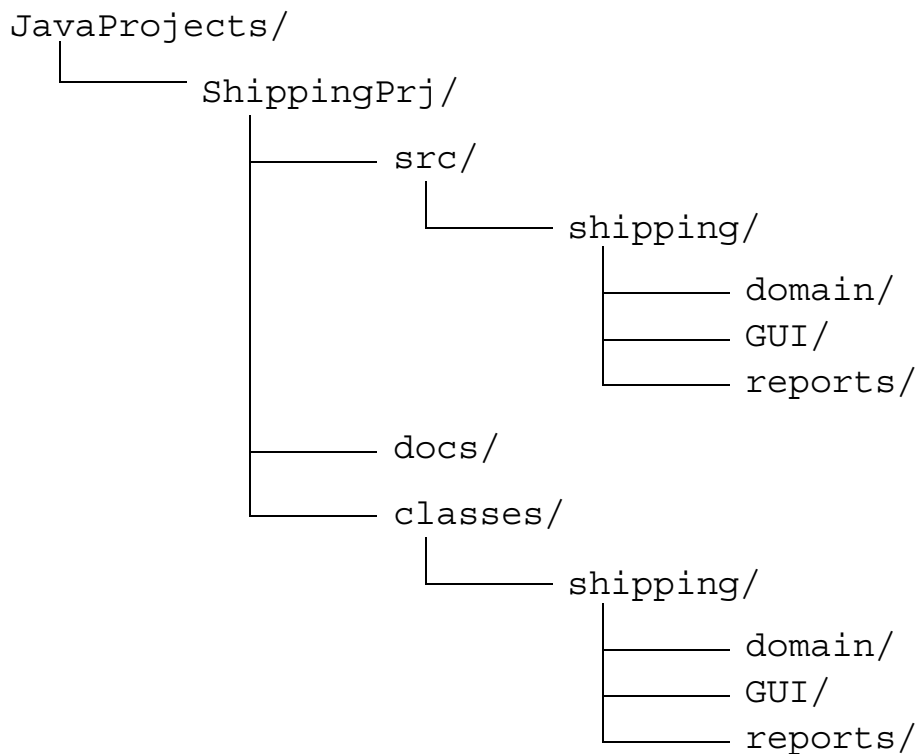


圖 3-2 專案開發案的目錄結構範例

編譯時使用 -d 選項

通常 Java 編譯器會將 class 檔置放於和原始檔相同的目錄。你可以使用 `javac` 命令的 `-d` 選擇將 class 檔放置於指定的目錄下。要編譯 Java 程式最簡單的方式就是在根套件的上一層目錄中使用 `javac` 命令（以上面的例子來講就是 `src`）。因此要編譯 `shipping.domain` 套件下所有的檔案，然後將 class 檔放置於 `ShippingPrj/classes/` 下，便可使用下列命令：

```
cd JavaProjects/ShippingPrj/src
javac -d ../classes shipping/domain/*.java
```

在 `src` 目錄下使用下列命令執行 `MyClass`（屬於 `shipping.domain` 套件）：

```
java -cp ../classes shipping.domain.MyClass
```

佈署

你可以佈署程式到用戶端而不需要修改使用者的 CLASSPATH 環境變數。通常最好的方法是產生可執行的 **Java archive (JAR)** 檔。要產生可執行的 JAR 檔，你必須先建立一個暫時性的檔案，其內容指定主類別，該檔案的內容如下所示：

Main-Class: mypackage.MyClass

接著使用 **jar** 命令再加上額外的選項，將上述暫時性的檔複製成為 **META-INF/MANIFEST.MF**，如下所示：

jar cmf tempfile MyProgram.jar MyApp

最後便可使用下列命令執行程式：

java -jar /path/to/file/MyProgram.jar

注意 — 有些作業系統只要快速的點選 JAR 檔兩次便可執行程式。



佈署函式庫

有時候你需要用到 JAR 中的函式庫，在這種狀況下您可將 JAR 檔複製到 JRE 的 lib 目錄下的 ext 目錄中。請注意，這樣的作法通常會開啟其它類別使用該 JAR 中所有類別的權限，且執行時期可能會有類別同名衝突問題發生。

宣告類別：類別方法 (Methods) 的語法

單元重點

完成這個單元後，您將能夠：

- 了解類別方法以及其撰寫方式
- 了解運算式
- 了解運算式的進階議題
- 建立敘述 (**statements**)，包括區塊 (**block**)、分支及迴圈
- 進階的類別方法設計技巧：
 - 方法的多載
 - 按值傳遞 (**pass-by-value**)
 - **this** 參考

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O’ Reilly Media. 2005.

了解類別方法

方法 (methods) 的目的在於提供類別實例的行為。在這個單元中，藉由討論下列各 Java 語言構件與類別方法間的關係來說明類別方法宣告的相關內容。

- 類別和方法的關係
- 方法介面 (method interface) 與方法本體 (body) 內容的關係
- 欄位，參數，區域變數與方法本體的關係
- 變數有效範圍和程式區塊的關係

以下使用程式 4-1 MoneyJar 類別說明上述的關係。

程式 4-1 MoneyJar 類別的方法

```

1  public class MoneyJar {
2      String purpose;
3      double amount;
4      double target;
5      double surplus;
6
7      void add(double sum) {
8          amount = amount + sum;
9          surplus = amount - target;
10     }
11
12     double takeOut(double sum) {
13         double sumReturned = 0;
14         if (amount >= sum) {
15             amount = amount - sum;
16             sumReturned = sum;
17         }
18         else {
19             sumReturned = amount;
20             amount = 0;
21         }
22         surplus = amount - target;
23         return sumReturned;
24     }
25
26     double getAmount() {
27         return amount;
28     }
29

```

```

30 void setTarget(double newTarget) {
31     target = newTarget;
32     surplus = amount - target;
33 }
34
35 double getTarget() {
36     return target;
37 }
38
39 // 其它方法
40 }

```

了解類別與類別方法間的關係

類別是一種資料型別，包含下列幾個元素：

- 欄位
欄位定義資料型別的屬性（或狀態），換句話說，也就是類別物件的屬性。
- 方法
方法定義了資料型別的行為，換句話說，也就是定義類別物件的行為。

方法會存取欄位（資料存放的地方）同時也會改變存放在欄位中的資料。就這個觀點來看，可將方法區分為兩種：

- 可變更欄位資料的方法（Mutator）
所謂可變更欄位資料的方法指的是：方法會改變一個或多個存放在欄位中的資料。
- 不可變更欄位資料的方法（Non-Mutator）
所謂不可變更欄位資料的方法指的是：方法不會改變任何欄位資料。這種方法基本目的在於回傳欄位值（取得資料）或是用於欄位及參數的計算。
- 表 4-1 透過可不可以變更欄位資料的觀點，將程式 4-1 MoneyJar 類別的方法做分類。

表 4-1 MoneyJar 類別方法的分類

方法名稱	方法種類	目的
add(double sum)	Mutator	變更 amount

表 4-1 MoneyJar 類別方法的分類

方法名稱	方法種類	目的
takeOut(double sum)	Mutator	變更 amount
getAmount()	Non-mutator	回傳 amount
setTarget(double newTarget)	Mutator	Mutator 變更 target
getTarget()	Non-mutator	回傳 target

了解方法介面 (method interface) 與方法本體間的關係

程式 4-2 呈現了 MoneyJar 類別 setTarget 方法與其內容

程式 4-2 方法範例

```

1    public void setTarget(double newTarget) // 方法介面
2    {                                     // 方法內容起始
3        target = newTarget;
4        surplus = amount - target;
5    }                                     // 方法內容結束

```

Java 中的方法包含兩種結構：

- 方法介面

方法介面定義了此方法所提供的服務（或行為）。可以將其想像為使用者（呼叫者）和方法間的約定。方法介面包含下列幾個元素：

- 方法的回傳資料型態
- 方法名稱
- 方法的參數

- 方法本體

方法本體包含為了實作方法介面所定義的服務而撰寫的程式碼。方法的本體用一對大括弧來標定表示起始與結束。

比較方法介面與方法簽章 (method signature)

不要混淆了方法介面與方法簽章。Java 規範中所謂的方法簽章只包含方法名稱及參數，不包含回傳值。

表 4-2 列出了程式 4-1 中 **MoneyJar** 類別的方法介面和方法簽章。

表 4-2 方法介面與方法簽章的比較

方法名稱	方法介面	方法簽章
add	void add(double sum)	add(double sum)
takeOut	double takeOut(double sum)	takeOut(double sum)
getAmount	double getAmount()	getAmount()
setTarget	void setTarget(double newTarget)	setTarget(double newTarget)
getTarget	double getTarget()	getTarget()

了解欄位，參數，區域變數與方法本體的關係

欄位，參數，區域變數都是資料的持有者。由這個角度來看方法本體，所有的欄位，參數，同一個有效範圍（**scope**）內的區域變數都是運算元（**operands**）。表 4-3 呈現了程式 4-1 **MoneyJar** 類別其欄位，參數，區域變數等的可存取性。

表 4-3 **MoneyJar** 類別中的資料持有者

變數名稱	分類	可存取者
purpose	欄位	所有方法
amount	欄位	所有方法
target	欄位	所有方法
surplus	欄位	所有方法
sum（在 add 方法中宣告）	參數	Add 方法
sum（在 takeOut 方法中宣告）	參數	takeOut 方法
sumReturned	區域變數	takeOut 方法
newTarget	參數	setTarget 方法

表 4-4 為欄位，參數，區域變數特性的比較表。

表 4-4 欄位，參數，區域變數特性的比較表

分析項目	欄位 (Field)	參數 (Parameter)	區域變數 (Local Variable)
目的	儲存物件屬性	傳遞輸入值給方法	方法使用的暫時變數
宣告位置	類別本體之內方法之外	方法簽章中	方法本體之內
如何宣告	型別 名稱 [= 值]; type name [= value];	<i>型別 名稱</i>	型別 名稱 [= 值]; type name [= value];
如何初始	預設，指定值或使用建構子	呼叫方法	指定值
變數有效範圍	類別內容（包含所有的方法）	與方法簽章相對應的方法內容	從宣告的地方開始至程式區塊的結束

了解方法本體

方法的本體實作了其行為。也就是說，行為是由撰寫在方法中的 **Java** 程式語言敘述句來實現。您可以將敘述句區分為以下幾種：

- 運算式敘述句
- 宣告敘述句
- 設值敘述句
- 流程控制敘述句
 - 分支敘述句
 - 迴圈敘述句
- 區塊敘述句

大部分的敘述句都使用了運算式 (**expressions**)。程式 4-3 為敘述句與運算式的範例。程式中使用粗體字的部分屬於運算式。

程式 4-3 運算式範例

```

1  public double takeOut(double sum) {
2      double sumReturned = 0;           // 宣告敘述句
3      if (amount >= sum) {               // 分支敘述句起始
4          amount = amount - sum;         // 指定敘述句
5          sumReturned = sum;
6      }                                   // 分支敘述句結束
7      else {
8          sumReturned = amount;
9          amount = 0;
10     }
11     surplus = amount - target;
12     return sumReturned;               // 回傳敘述句
13 }
```



注意 -- 程式 4-3 中包含了幾個區塊敘述句的範例。有的時候區塊 (**block**) 也稱為複合敘述句 (**compound statement**)，是由一群敘述句集合而成，使用一組開始與結束的大括弧表示區塊的範圍 ({ })。

了解運算式 (Expressions)

本節將介紹 **Java** 運算式。所有的運算式都有下列的共同特性：

- 一個運算式至少要有一個運算子 (**operator**)。依據運算子使用的數目可分為簡單及複合兩種。例如：

```
x + 5; // 簡單運算式具有一個運算子
```

```
x + 5 * y; // 複合運算式具有多個運算子
```

- 一個運算子有幾個運算元是由運算子決定。有些是二元，有些則是單一運算元。例如：

```
x < 2; // 二元運算子範例
```

```
++x; // 單元運算子範例
```

- 運算式結果會產生一資料型別

資料型別取決於下列兩個因素：

- 運算子
- 運算元資料型別

表 4-5 列舉 **Java** 運算式執行結果的資料型別。

表 4-5 運算式執行結果的資料型別

運算式	運算子	運算元資料型別	執行結果的資料型別
<code>x + y;</code>	數值加總	數值	數值
<code>x < 2;</code>	比較	數值	布林值
<code>"sun" + 22</code>	字串連結	至少有一個運算元是字串	字串
<code>x & 22;</code>	位元的 AND	int 或 long	int 或 long

了解簡單運算式

以下將介紹簡單運算式，也就是只包含一個運算子的運算式。

算術運算子簡介

Java 同時支援二元以及單元算術運算子。表 4-6 列出了 **Java** 語言所支援算術型別的二元運算子。表 4-6 的結果欄位為 $x = 7$ ， $y = 3$ 所計算出的結果。

表 4-6 二元算術運算子

目的	運算子	範例	結果
加	+	<code>result = x + y;</code>	10
減	-	<code>result = x - y;</code>	4
乘	*	<code>result = x * y;</code>	21
除	/	<code>result = x / y;</code>	2
餘數	%	<code>result = x % y;</code>	1

表 4-7 列出 **Java** 語言所支援算術型態的單元運算子。

表 4-7 單元算術運算子

目的	運算子	範例	num1 先運算	結果	num1 後運算
加	+	<code>result = +num1;</code>	7	7	7
減	-	<code>result = -num1;</code>	7	-7	7
先加	++	<code>result = ++num1;</code>	7	8	8
後加	++	<code>result = num1++;</code>	7	7	8
先減	--	<code>result = --num1;</code>	7	6	6
後減	--	<code>result = num1--;</code>	7	7	6

位元邏輯運算子和位移運算子

位元的操作包含邏輯操作和平移 (shift) 操作，屬於低階的操作，會直接影響整數的位元資料。這些操作不常用於企業系統，但對於圖形、科學、控制系統則是非常重要。這種操作位元資料的能力可以省下大量的記憶體，也可加速計算的效率，也簡化對於位元集合的操作。例如由平行埠讀取或寫入資料。

Java 程式語言提供對於整數資料型別的位元操作。下列的位元運算子： \sim ， $\&$ ， \wedge ，和 \mid 分別對應 NOT，AND，XOR，OR。表 4-8 為 Java 提供的邏輯運算子和位移運算子使用範例。

表 4-8 邏輯運算子和位移運算子

目的	運算子	使用範例
位元反轉	\sim	$\sim x$
位元 OR 位元 AND 位元 XOR	\mid $\&$ \wedge	$x \mid y$ $x \& y$ $x \wedge y$
位元號數左移 位元號數右移 位元無號數右移	\ll \gg \ggg	$x \ll y$ $x \gg y$ $x \ggg y$

圖 4-1 位元運算子範例。

\sim	<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	0	0	1	1	1	1	1	0	1	1	0	0	0	0	$\&$	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	0	0	1	0	1	1	0	1	0	1	0	0	1	1	1	1	0	0	0	0	1	1	0	1								
0	1	0	0	1	1	1	1																																												
1	0	1	1	0	0	0	0																																												
0	0	1	0	1	1	0	1																																												
0	1	0	0	1	1	1	1																																												
0	0	0	0	1	1	0	1																																												
\wedge	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	1	0	1	1	0	1	0	1	0	0	1	1	1	1	0	1	1	0	0	0	1	0	$ $	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	0	0	1	0	1	1	0	1	0	1	0	0	1	1	1	1	0	1	1	0	1	1	1	1
0	0	1	0	1	1	0	1																																												
0	1	0	0	1	1	1	1																																												
0	1	1	0	0	0	1	0																																												
0	0	1	0	1	1	0	1																																												
0	1	0	0	1	1	1	1																																												
0	1	1	0	1	1	1	1																																												

圖 4-1 用位元運算子的範例

了解運算式 (Expressions)

圖 4-2 呈現了平移運算子對於正負數的位元操作結果，分別使用 `>>`，`>>>` 和 `<<`。

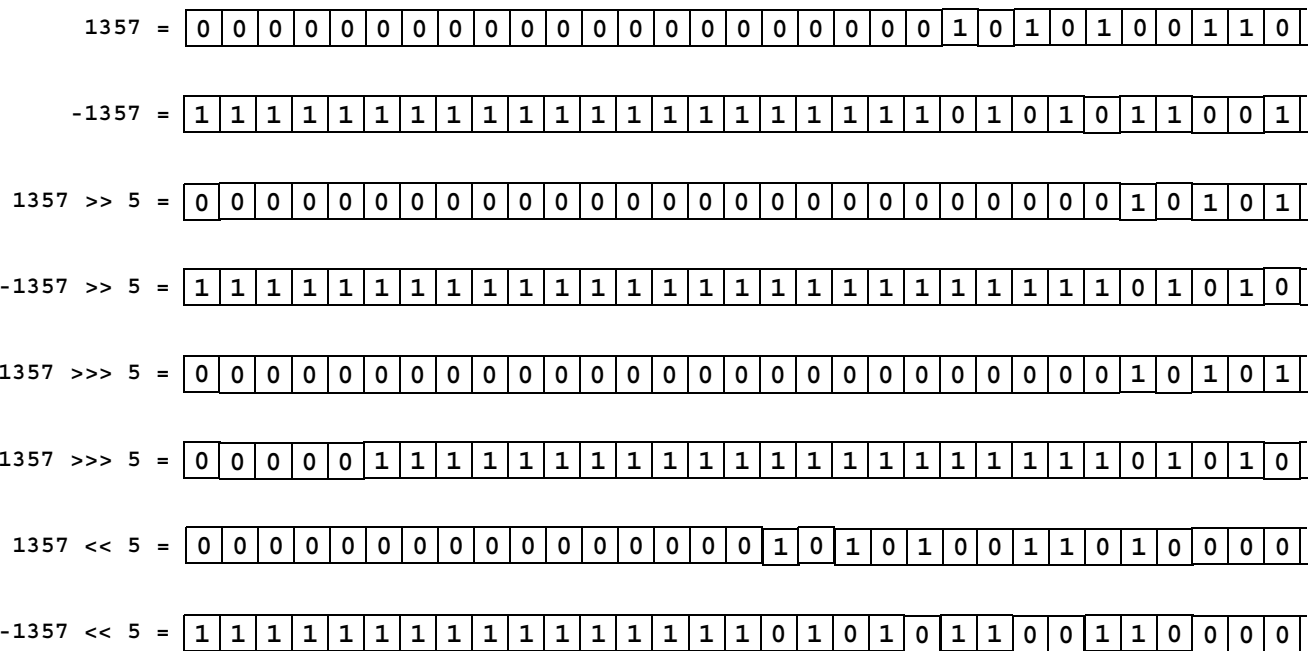


圖 4-2 位元平移運算子範例

設值 (Assignment) 運算子

設值運算子可區分為簡單及複合兩種。複合設值是由算術操作與位元操作或是位元的平移操作加上設值操作所組合而成。表 4-9 列出了在 **Java** 技術中能夠使用的簡單與複合設值運算子。

表 4-9 指定運算子

目的	運算子	範例	等同運算式
簡單設值	=	x = y;	
複合算術操作設值	+=	x += y;	x = x + y;
	-=	x -= y	x = x - y;
	*=	x *= y	x = x * y;
	/=	x /= y	x = x / y;
	%=	x %= y	x = x % y;
複合位元操作設值	=	x = y;	x = x y;
	&=	x &= y;	x = x & y;
	^=	x ^= y;	x = x ^ y;

表 4-9 指定運算子

目的	運算子	範例	等同運算式
複合平移操作設值	<<= >>= >>>=	x <<= y; x >>= y; x >>>= y;	x = x << y; x = x >> y; x = x >>> y;

關係（邏輯）運算子

關係運算子得出的結果為布林值：**true** 或 **false**；表 4-10 的“結果”欄位中的值是使用下列資料所計算後的結果：

```
int x=7; int y=3; boolean a=true; boolean b=false;..
```

表 4-10 關係運算子

目的	運算子	範例	結果
大於	>	result = x > 7;	false
大於等於	>=	result = x >= 7;	true
等於	==	result = x == y;	false
不等於	!=	result = x != y;	true
小於	<	result = x < y;	false
小於等於	<=	result = x <= y;	false
邏輯 OR		result = a b;	true
邏輯 AND	&&	result = a && b;	false
邏輯 OR		result = (x<3) (y>2);	true
邏輯 AND	&&	result = (x<3) && (y>2);	false

字串串接運算子

運算子 **+** 可以對 **String** 物件執行串接操作，產生新的 **String** 物件。

```
String salutation = "Dr. ";
String name = "Pete" + " " + "Seymour";
String title = salutation + " " + name;
```

最後一行的結果為：

```
Dr. Pete Seymour
```

只要任何一個運算元是 **String** 物件，所有的物件都會被自動轉換成 **String** 物件，但有時在運算後可能會產生我們不能理解的結果。非 **String** 物件的 `toString()` 方法會被呼叫以轉換為成 **String** 物件。

運算式：進階議題

以下會討論下列三個進階的運算式議題：

- 簡單運算式中數值型態的升級轉換 (promotion)
- 形別轉換
- 執行複合運算式

簡單運算式中數值型態的升級 (Promotion) 轉換

請參考下列運算式：

```
result = operand1 + operand2
```

result 資料型別會根據運算元的資料型別而定。表 4-11 為各種運算元組合的列表。

表 4-11 數值型別的升級轉換

Operand 1 資料型別	Operand 2 資料型別	結果
Byte, Character, Short, byte, char, short	Byte, Character, Short, byte, char, or short	int
Byte, Character, Short, Integer, byte, char, short, int	Integer or int	int
Byte, Character, Short, Integer, byte, char, short, int	Long or long	long
Byte, Character, Short, Integer, Long, byte, char, short, int, long	Float or float	float
Byte, Character, Short, Integer, Long, Float, byte, char, short, int, long, float	Double or double	double

Java 於運算之前會先將運算元的資料型別轉換成與運算結果相同的資料型態，然後再進行計算。這種情況稱之為數值型別的升級 (promotion)。數值型的升級適用於下列運算子：

- 算術運算子
+ - * / %
- 位元運算子

& | ^

- 關係運算子

< <= == != > >=

型別轉換

如果資料不會失真，則變數會被自動升級至資料長度較長的型別（例如由 `int` 到 `long`）。

```
long bigval = 6;      // 6 為 int, OK
int smallval = 99L;   // 99L 為 long, 不合法
```

```
double z = 12.414F;   // 12.414F 為 float, OK
float z1 = 12.414;     // 12.414 為 double, 不合法
```

通常上，假如變數資料型別與結果資料型別長度一樣，則您可以將運算式視為可相容的設值動作。至於二元運算子（例如 `+`），當兩個運算元都是基本型別時，結果的資料型別則取決於資料長度最長的一個或是 `int`。因此，所有的二元運算子在運算基本型別時，結果的資料型別至少都是 `int`，但也可能是比 `int` 還長的運算元，例如 `long`、`float` 和 `double` 等，因此可能會有溢位或是遺失精準度的問題。

例如下列的程式片段：

```
short a, b, c;
a = 1;
b = 2;
c = a + b; // 造成錯誤
```

因為每個 **short** 都升級為 **int** 所以造成錯誤。然而如果宣告 **c** 為 **int** 或是用下面的轉換：

```
c = (short) (a + b);
```

那就可以成功。

基本資料型別的 Autoboxing

假如您要將基本資料型別換成相對應的物件（稱為 **boxing**）時必須使用包覆類別。若要由包覆類別物件將基本資料型別中的值取出時（稱為 **unboxing**），則需要使用包覆類別的方法。**Boxing** 和 **Unboxing** 可能會讓您的程式碼變的複雜而且不容易閱讀與理解。**autoboxing** 的功能是 **Java SE 5.0** 所提供的新功能，能夠在設值或存取基本型別資料時不再需要使用包覆類別。程式 4-4 是 **autoboxing** 和 **autounboxing** 範例。請您將程式 4-4 和程式 3-6 加以對照比較。

程式 4-4 Autoboxing 範例

```
int pInt = 420;
Integer wInt = pInt; // 為 autoboxing
int p2 = wInt; // 為 autounboxing
```

Java SE 5.0 或是後續版本的編譯器會在設值或存取基本型別資料時，自動產生包覆類別物件，該操作可在傳遞參數到類別方法中時或是在運算式中自動完成。



注意 —— 不要過分使用 **autoboxing** 功能。因為有 **autoboxed** 或 **autounboxed** 潛藏的效能問題。基本型別與包覆物件混合的算術運算式在迴圈程式上可能會導致效能的問題。

執行複合運算式

複合運算式包含兩個以上的運算子，例如：

```
x + 5 * y/25;    // 複合運算式包含多個運算子
```

執行複合運算式時，必須注意下列事項：

- 運算子優先次序
- 運算子的關聯性

表 4-12 列出運算子優先次序（L to R 表示左到右關聯；R to L 表示右到左關聯）。

表 4-12 運算子和優先次序

關聯性	運算子
R to L	++ -- + - ~ ! (<data_type>)
L to R	* / %
L to R	+ -
L to R	<< >> >>>
L to R	< > <= >= instanceof
L to R	== !=
L to R	&
L to R	^
L to R	
L to R	&&
L to R	
R to L	<boolean_expr> ? <expr1> : <expr2>
R to L	= *= /= %= += -= <<= >>= >>>= &= ^= =

敘述句 (Statement)

一個 **Java** 敘述句構成一個完整可執行的單位。敘述句機制讓類別方法可以實作其行為。在 **Java** 中，敘述句是在方法內撰寫，如程式 4-5 所示。

程式 4-5 敘述句範例

```
1 public double takeOut2(double sum) {
2     double sumReturned = 0;           // 宣告敘述句
3     if (amount > sum)                   // 分支敘述句開始
4     {                                   // 區塊敘述句開始
5         amount = amount - sum;
6         sumReturned = sum;
7     }                                   // 區塊敘述句結束
8     else
9     {
10        sumReturned = amount;
11        amount = 0;
12    }                                   // 分支敘述句結束
13    surplus = amount - target;          // 指定敘述句
14                                         // 方法呼叫敘述句
15    System.out.println("taking out"+ sumReturned);
16    return sumReturned;                 // 回傳敘述句
17 }
```

表 4-13 列出 **Java** 最常用的敘述句的類別。

表 4-13 Java 敘述句的類型

敘述句主要類型	敘述句次要類型	註解和範例
運算敘述句	設值運算 任何使用 ++ 或 -- 方法呼叫 建構物件 運算式	x = a + b; x++; System.out.println("hello"); j = new Integer(25);
區塊敘述句		區塊是一群介於兩個成對的大括弧間的敘述句，可視為一個敘述句
宣告敘述句		int x;

Yu-Wei SHEN (SHEN_Yu-Wei@yahoo.com.tw) has a non-transferable license to use this Student Guide

敘述句 (Statement)

表 4-13 Java 敘述句的類型

敘述句主要類型	敘述句次要類型	註解和範例
流程控制敘述句	分支 迴圈 流程控制	if 敘述句 switch 敘述句 for loop while loop do while loop Label 敘述句 continue 敘述句 break 敘述句

下面的章節會進一步說明區塊敘述句和流程控制敘述句。

區塊敘述句

區塊，有時稱之為複合敘述句，是由一群在成對的大括弧內敘述句所組成。
下列為區塊敘述句範例：

```
// 區塊敘述句
{
    x = y + 1;
    y = x + 1;
}

// 含有區塊敘述句的類別定義
public class MyDate {
    private int day;
    private int month;
    private int year;
}

// 區塊敘述句可以位於另一個區塊敘述句內
// another block statement
while ( i < large ) {
    a = a + i;
    // 巢狀區塊
    if ( a == max ) {
        b = b + a;
        a = 0;
    }
    i = i + 1;
}
```

分支敘述句

條件式敘述句使程式可以根據某些運算結果，選擇要執行的部份程式。Java 程式語言支援 **if** 和 **switch** 敘述句，分別提供雙向和多向的分支。

簡單的 if 敘述句

基本的 if 敘述句為：

```
if ( <boolean_expression> )
    <statement_or_block>
```

例如：

```
if ( x < 10 )
    System.out.println("Are you finished yet?");
```

然而，應該要將敘述句以大括弧圍住。 例如：

```
if ( x < 10 ) {
    System.out.println("Are you finished yet?");
}
```

簡單的 if, else 敘述句

假如您需要 **else** 子句，那您就要用 **if-else** 敘述句：

```
if ( <boolean_expression> )
    <statement_or_block>
else
    <statement_or_block>
```

例如：

```
if ( x < 10 ) {
    System.out.println("Are you finished yet?");
} else {
    System.out.println("Keep working...");
}
```

複雜的 if, else 敘述句

假如您需要一連串的條件檢查時可以用一連串的 **if-else-if** 敘述句

```
if ( <boolean_expression> )
    <statement_or_block>
else if ( <boolean_expression> )
    <statement_or_block>
```

例如：

```
int count = getCount(); // 類別中的方法
if (count < 0) {
    System.out.println("Error: count value is negative.");
} else if (count > getMaxCount()) {
    System.out.println("Error: count value is too big.");
} else {
    System.out.println("There will be " + count +
                       " people for lunch today.");
}
```

上述程式中 **else** 的敘述句，在沒有任何相對應的動作時可以忽略不寫。

switch 敘述句

switch 敘述句的語法為：

```
switch ( <expression> ) {  
    case <constant1>:  
        <statement_or_block>*  
        [break;]  
    case <constant2>:  
        <statement_or_block>*  
        [break;]  
    default:  
        <statement_or_block>*  
        [break;]  
}
```

在 **switch** (<expression>) 敘述句中，<expression> 必須是與 **int** 基本型別相容的運算式。型態的升級會發生在使用 **byte**, **short**, or **char** 等。浮點數，**long** 運算式或是物件參考都不可使用於 **expression**。



注意 — 列舉型別的值也可以於 <expression> 和 <constantN> 中使用。列舉型別不在本章的討論範圍。

選擇性的 **default** 標籤表示當 <expression> 結果無法符合所有的條件則 **default** 部分的程式會被執行。若是進入 **case** 後，沒有 **break** 敘述句則下一個 **case** 仍會被執行且不檢查其 <expression> 結果。如果 **switch** 是在一個迴圈內，那 **continue** 敘述句的使用會使程式離開整個 **switch**。



注意 — 大部分的 **switch** 敘述句在每個 **case** 區塊中會需要 **break**。於 **switch** 忘記使用 **break** 大都會造成程式錯誤。

請參閱程式 4-6 **switch** 敘述句的範例

程式 4-6 switch 敘述句範例 1

```
witch ( carModel ) {  
  case 'D': // 豪華版  
    addAirConditioning();  
    addRadio();  
    addWheels();  
    addEngine();  
    break;  
  case 'S': // 標準版  
    addRadio();  
    addWheels();  
    addEngine();  
    break;  
  default:  
    addWheels();  
    addEngine();  
}
```

程式 4-6 依照 **carModel** 來設定 **car** 物件。假如 **carModel** 為 (D)eluxe，則汽車會加上空調，音響，輪子和引擎。然而若是 **carModel** 為 (S)tandard，則只會有音響，輪子和引擎。最後汽車預設會有輪子和引擎。

程式 4-7 呈現第二個 **switch** 敘述句範例。

程式 4-7 switch 敘述句範例 2

```
switch ( carModel ) {  
  case 'D': // 豪華版  
    addAirConditioning();  
    // 繼續執行  
  case 'S': // 標準版  
    addRadio();  
    // 繼續執行  
  default:  
    addWheels();  
    addEngine();  
}
```

程式 4-7 利用流程的控制解決上個範例中多餘的程式碼。例如：假如 **carModel** 為 (D)eluxe 那就會執行 **addAirConditioning** 方法，往下繼續執行下一個 **case** 敘述句中的 **addRadio** 方法最後到 **default** 敘述句呼叫 **addWheels** 和 **addEngine** 方法。

迴圈敘述句

迴圈敘述句可以使您重複執行敘述句的區塊。**Java** 程式語言支援 4 種迴圈結構：**for**（包含兩種變形），**while**，以及 **do**。**while** 迴圈和其中一種 **for** 迴圈相同，會在進入迴圈內容時先檢查迴圈條件，**do** 迴圈則是先執行迴圈內容後再驗證條件。這表示 **do** 迴圈至少執行迴圈內容一次。第二種 **for** 迴圈，也就是加強版 **for** 迴圈，是用來依序存取一群集合中的物件，不需指定條件式。

for 迴圈

For 迴圈語法為：

```
for ( <init_expr>; <test_expr>; <alter_expr> )  
    <statement_or_block>
```

例如：

```
for ( int i = 0; i < 10; i++ )  
    System.out.println(i + " squared is " + (i*i));
```

建議您應該將迴圈內容寫成區塊敘述句。例如：

```
for ( int i = 0; i < 10; i++ ) {  
    System.out.println(i + " squared is " + (i*i));  
}
```

前一個例子中 **int i** 是在 **for** 區塊中宣告。變數 **i** 有效範圍只有在 **for** 迴圈內。



注意 --**Java** 程式語言允許在 **for()** 結構中使用逗號作為分隔標記。例如，**for (i = 0, j = 0; j < 10; i++, j++) { }** 是合法的，他指定 **i** 和 **j** 初始值為 0，每執行一次迴圈後 **i** 和 **j** 各加 1。

while 迴圈

while 迴圈語法為：

```
while ( <test_expr> )  
    <statement_or_block>
```

例如：

```
int i = 0;  
while ( i < 10 ) {  
    System.out.println(i + " squared is " + (i*i));  
    i++;  
}
```

請留意要確保迴圈控制變數在執行迴圈內容前要有適當的初始化動作，並且要適當的更新迴圈控制變數以防止無窮迴圈。

do/while 迴圈

do/while 迴圈的語法為：

```
do  
    <statement_or_block>  
while ( <test_expr> );
```

例如：

```
int i = 0;  
do {  
    System.out.println(i + " squared is " + (i*i));  
    i++;  
} while ( i < 10 );
```

使用 **while** 迴圈時，也要確保迴圈控制變數於執行迴圈內容前有適當的初始化，且要適當的更新迴圈控制變數，此外，也必須定義適當的測試條件。

For 迴圈用於事先已經決定要執行的次數的重覆工作。**while** 和 **do** 則適用於未事先決定執行次數狀況下的重覆工作。

特殊的迴圈流程控制機制

您可以使用下列的敘述句控制迴圈的流程。

- **break** [*<label>*];
使用 **break** 敘述句可以提早離開 **switch** 敘述句，迴圈敘述句和標籤區塊。
- **continue** [*<label>*];
使用 **continue** 敘述句可以略過迴圈內容或是跳到迴圈內容的結束點，然後繼續執行迴圈控制敘述句。
- *<label>* : *<statement>*
Label 敘述句用來標識敘述句或是區塊，大部分會用在迴圈外一開始的位置。使用 **Label** 配合 **break** 敘述句可以提早離開 **Label** 所對應的區塊。
.
使用 **Label** 的 **continue** 敘述句必須使用在迴圈的結構。一但程式執行到使用 **Label** 的 **continue** 敘述句，JVM 會略過該區塊以下的程式跳至對應 **Label** 區塊的起始位置。

Break 敘述句

下面為未使用 **Label** 的 **break** 敘述句：

```

1  do {
2      statement;
3      if ( condition ) {
4          break;
5      }
6      statement;
7  } while ( test_expr );
```

continue 敘述句

下面為未使用 **Label** 的 **continue** 敘述句：

```

1  do {
2      statement;
3      if ( condition ) {
4          continue;
5      }
6      statement;
7  } while ( test_expr );
```

使用 Label 的 break 敘述句

下面範例為使用 label 的 break 敘述句：

```
1  outer:
2      do {
3          statement1;
4          do {
5              statement2;
6              if ( condition ) {
7                  break outer;
8              }
9              statement3;
10         } while ( test_expr );
11         statement4;
12     } while ( test_expr );
```

使用 continue 敘述句與 Label

下面的範例為使用 label 的 continue 敘述句：

```
1  test:
2      do {
3          statement1;
4          do {
5              statement2;
6              if ( condition ) {
7                  continue test;
8              }
9              statement3;
10         } while ( test_expr );
11         statement4;
12     } while ( test_expr );
```

類別方法的進階設計議題：方法多載

在某些情況下，您可能要在一個類別撰寫幾個方法，這些方法做同樣的事，但具有不同的參數。

如果您有個方法只是要以文字方式輸出其參數值，這個方法假設叫作 `println()`。現在您可能有不同的 `print` 方法輸出個別的 `int`, `float`, and `String` 等型態。這種狀況是合理的，因為不同的資料型態有不同個格式，可能有不同的處理方式。您可以個別撰寫方法分別為 `printlnInt()`, `printlnFloat()` 和 `printlnString()`。但是這樣看起來不怎麼好用。

Java 程式語言允許您重複使用方法名稱。這種方式只有在能區分方法的情況下才能運作。就上面三個 `print` 方法而言，可以經由參數的數量與型別的不同來做區分。透過使用相同的類別方法識別名稱，可以定義下列幾個方法：

```
public void println(int i)
public void println(float f)
public void println(String s)
```

當您寫程式呼叫其中一個方法時，會依據您所傳遞的參數型別找到合適的方法。

方法多載須遵守兩個規則：

- 必須具有不同的參數

呼叫的敘述句需要具有足夠的相異參數來決定呼叫合適的方法。一般的型別升級可能可以適用於此項規則，但某些情況下也容易造成混淆。

- 回傳型別可以不同

雖然回傳的型態可以不同，但仍無法分辨。因此還是要具有不同的參數。

在類別方法中使用變動參數

方法多載的另一個使用時機是當您需要一個方法，而這個方法具有相同型別但數量不同的參數。例如：您需要產生一個計算平均值的方法，您可能會定義下列的方法：

```
public class Statistics {
    public float average(int x1, int x2) {}
    public float average(int x1, int x2, int x3) {}
    public float average(int x1, int x2, int x3, int x4) {}
}
```

這些方法會以下列的方式被呼叫：

```
Statistics stats = new Statistics();
float gradePointAverage = stats.average(4, 3, 4);
float averageAge = stats.average(24, 32, 27, 18);
```

這三個多載的方法有著同樣功能。如果能將三個方法變為一個會更好。因此自 **Java SE 5.0** 開始提供一個新功能，稱之為變動參數 (**varargs** or **variable arguments**)，讓您可以透過下列方式撰寫共通的方法：

```
public class Statistics {
    public float average(int... nums) {
        int sum = 0;
        for ( int x : nums ) {
            sum += x;
        }
        return ((float) sum) / nums.length;
    }
}
```

呼叫這種新的可變參數方法與之前的多載方法運作方式相同，**JVM** 會透過內建的一個長度屬性取得參數數量。

類別方法的進階設計議題：按值傳遞

Java 程式語言在傳遞參數時都是以傳值的方式，因此您無法在呼叫方法時改變參數值。然而當傳遞的參數是物件的時候，這時候傳給方法的不是物件本身，而是「物件參考」的複製。您可以在方法中透過這些物件參考改變物件中的資料，但無法改變原來的物件參考值。

對很多人而言，這樣看起來很像按址傳遞，而且行為上和按址傳遞有很多的共同處。有兩個理由可以解釋這樣的認知是錯誤的。第一，改變被傳遞者的能力只適用於類別型別，不適用於基本型別。第二，參數變數所持有的值是相對應物件的參考值，而非物件本身。如果深入且清楚理解上述的不同點的話，這是一個很重要的特點，說明了 **Java** 程式語言是按值傳遞。

下面的範例說明這個特點：

```
1  public class PassTest {
2
3  // 方法改變 value 的值
4  public static void changeInt(int value) {
5      value = 55;
6  }
7  public static void changeObjectRef(MyDate ref) {
8      ref = new MyDate(1, 1, 2000);
9  }
10 public static void changeObjectAttr(MyDate ref) {
11     ref.setDay(4);
12 }
13
14 public static void main(String args[]) {
15     MyDate date;
16     int val;
17
18 // 指定值
19     val = 11;
20 // 試著改變
21     changeInt(val);
22 // 目前的值是？
23     System.out.println("Int value is: " + val);
24
25 // 指定日期
26     date = new MyDate(22, 7, 1964);
27 // 試著改變
28     changeObjectRef(date);
29 // 目前的值是？
30     System.out.println("MyDate: " + date);
```

```
31
32     // 透過物件參考
33     // 改變 day 屬性
34     changeObjectAttr(date);
35     // 目前的值是?
36     System.out.println("MyDate: " + date);
37 }
38 }
```

程式的輸出結果如下：

```
java PassTest
```

```
Int value is: 11
MyDate: 22-7-1964
MyDate: 4-7-1964
```

MyDate物件在呼叫changeObjectRef後並沒有變。但是changeObjectAttr方法改變了 MyDate 物件的 day 屬性。

類別方法的進階設計議題：this 參考

this 關鍵字是用來

- 解決物件變數與方法內部參數的模糊性
- 將目前物件做為參數傳遞到另一個方法中

程式 4-8 的 **MyDate** 類別說明了這種用法。**MyDate** 類別於第 2-4 行宣告物件變數 **day**，6-10 行建構子的宣告也用到 **day** 的名稱，在這種狀況下 **this** 關鍵字解決了二個 **day** 變數的模糊性。在 **addDays** 方法中（第 18 行），產生了一個 **newDate** 物件。在 **MyDate** 建構子中，也使用了 **this** 關鍵字在建構子中指稱目前物件。

程式 4-8 **this** 關鍵字範例

```

1  public class MyDate {
2      private int day = 1;
3      private int month = 1;
4      private int year = 2000;
5
6      public MyDate(int day, int month, int year) {
7          this.day = day;
8          this.month = month;
9          this.year = year;
10     }
11     public MyDate(MyDate date) {
12         this.day = date.day;
13         this.month = date.month;
14         this.year = date.year;
15     }
16
17     public MyDate addDays(int moreDays) {
18         MyDate newDate = new MyDate(this);
19         newDate.day = newDate.day + moreDays;
20         // // 尚未實作...
21         return newDate;
22     }
23     public String toString() {
24         return "" + day + "-" + month + "-" + year;
25     }
26 }
```


建構類別：使用封裝 (Encapsulation) 機制

單元重點

當完成這個單元後，您將能夠：

- 了解封裝的概念
- 使用 **Java** 語言實作封裝
- 使用 **static** 關鍵字
- 了解 **static import** 的用法

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O' Reilly Media. 2005.

了解封裝概念

封裝是將資料型別（類別）的介面從實作中分開。圖 5-1 呈現了用來儲存日期以及股票資訊的資料元素，這些資料元素尚未進行封裝。

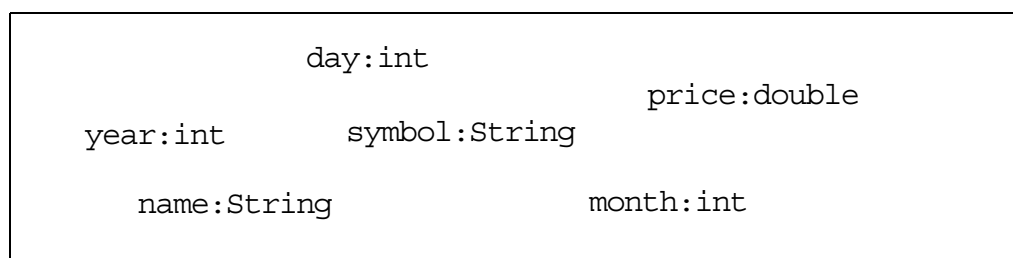


圖 5-1 未封裝的資料

當封裝資料元素時，必須進行下列步驟：

1. 將日期相關資料元素由股票的資料中分離出來。圖 5-2 呈現了分開後的兩組資料元素，一個是日期的資料而另一個則是股票的資料。

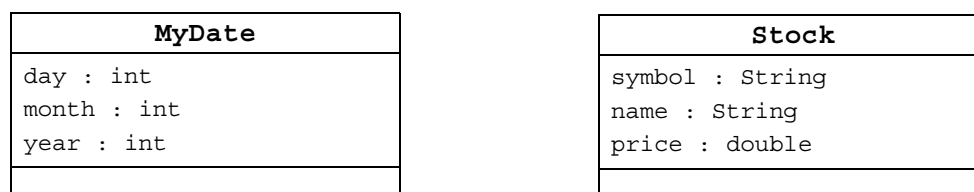


圖 5-2 封裝步驟 1：將相關的資料聚集成群

2. 將日期資料元素集中成為一個單元並加上相關的操作方式（類別方法）。同樣地，將股票資料元素也集中成為一個單元，也加上相關的操作方式。圖 5-3 呈現了將資料元素集中並加上類別方法的結果。

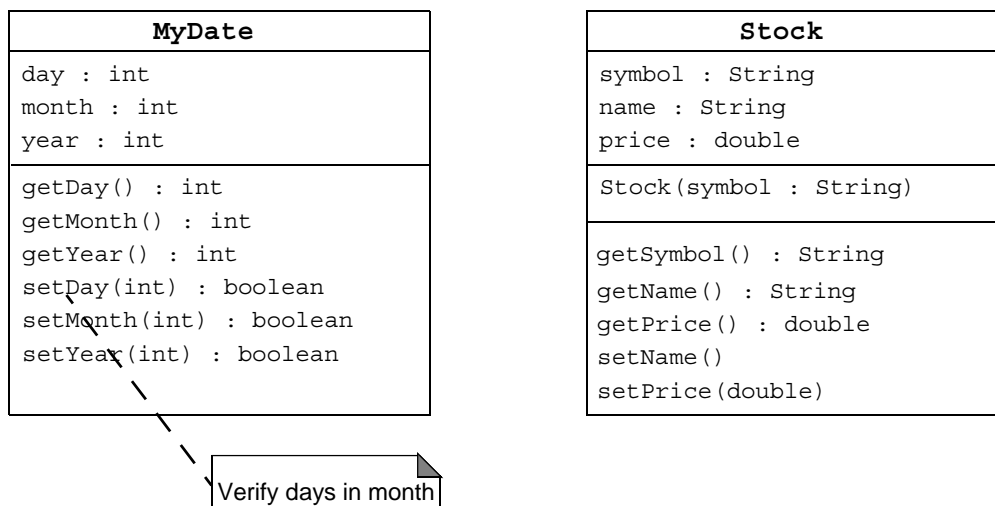


圖 5-3 封裝步驟 2：集中資料元素後加上行為

3. 最後一個步驟是為了防止外界直接存取資料元素，所以提供間接存取資料元素的方法。這個步驟稱為實作存取控制。圖 5-4 呈現了實作存取控制後的結果。

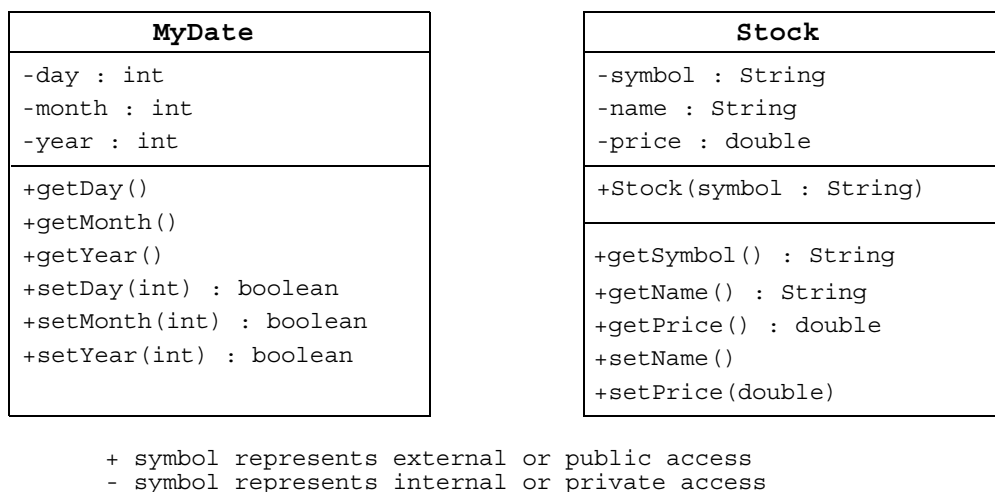


圖 5-4 封裝步驟 3：實作存取控制

封裝的好處有：

- 保護資料的完整性

封裝提供控制存取內部資料權限的機制。設計人員可以藉由這種的機制限制外界只能透過公開 (**public**) 方法存取內部資料。這些方法可以確保商業規則的執行並保持資料的完整性。例如：在 **MyDate** 類別中，**set** 方法可以確保 **MyDate** 物件是個有效的日期，像是二月絕對不會有 30 號。

- 應用程式的可維護性

限制資料只能透過公開介面存取的好處是使用者不會因為內部資料型別改變而必須跟著修改程式。

使用 Java 技術進行封裝

Java 技術提供下列語言元素來支援封裝機制：

- **package** 敘述

藉由 **package** 敘述可將類別放置於套件（函式庫）中。**package** 敘述能夠將彼此有關聯的類別封裝在一起，並放置於套件下。下面說明了套件等級（Package-Level）的封裝原則：

- 在同一個套件下的類別可以互相存取。
- 當類別為 **public** 時，除了在同一個套件下的類別可以互相存取，也允許被其他套件下的類別存取。
- 沒有標示為 **public** 的類別無法被其他套件下的類別存取。

- **class** 敘述

class 敘述將屬性、建構子（進行屬性的初始化）、和方法封裝成一個可編譯的單元。

- 存取修飾詞（Access Modifiers）

存取修飾詞對類別提供較簡單的存取限制，但對於類別中的成員則提供較細緻的存取控管。

Java 技術規範中定義四種存取修飾詞：

- 公開的（Public）

“公開的存取限制”使用 **Public** 關鍵字來定義。**Public** 修飾詞可以用於類別和類別成員的宣告敘述。

- 受保護的（Protected）

“受保護的存取限制”使用 **Protected** 關鍵字來定義。“受保護的存取限制”和類別間的繼承關係有關。當子類別繼承於另一個類別，受保護的父類別成員仍可以被子類別存取。

注意 — 關於繼承和 **Protected** 修飾詞使用的討論已經超過本單元的範圍。.



- 私有的

“私有的存取限制”使用 **private** 關鍵字來定義。**private** 修飾詞限制了私有成員（屬性、建構子及方法）只能存取相同類別中的成員。

- 預設的

Java 語言規範中沒有提供定義預設的關鍵字。如果沒有定義任何存取修飾詞（**public**, **private**, 或 **protected**）就表示為預設。

表 5-1 提供上述四種存取限制在各種可存取範圍下的列表。這四種可能的存取範圍分別為：

- 同一類別
“同一類別” 適用於使用方法存取位在同一類別中的任何成員。
- 同一套件
“同一套件” 適用於在同一個套件下，類別中的成員可以被另一類別的方法存取。
- 子類別
“子類別” 適用於子類別可以存取位在不同套件中的任何父類別成員。
- 全域
“全域” 適用於類別中的任何成員可以被不同套件下的類別的方法存取。

表 5-1 存取修飾詞與存取範圍

修飾詞	同類別	同套件	子類別	全域
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

程式 5-1 為精簡版的 **Date** 類別，它是一個公開的類別，且同時含有 **private**, **public**, 和 **default** 等成員

程式 5-1 **Date** 類別

```

1  package com.abc.util;
2
3  public class Date {
4      private int day;
5
6      public Date() { //... }
7
8      public void addDays(int days) { }
9          int getDaysInMonth(int month) { }
10 }

```

程式 5-2 為精簡版的 **Stock** 類別，與 **Date** 類別分別位在不同套件中。

程式 5-2 The Stock Class

```
1 package com.abc.brokerage;
2
3 public class Stock {
4     private String symbol;
5     public Stock(String symbol, double price) { }
6
7     public String getSymbol() { }
8     public void setSymbol(String symbol) { }
9 }
```

程式 5-3 為精簡版的 **StockAnalyzer** 類別，與 **Date** 類別分別位在不同套件中但與 **Stock** 類別位於同套件中。

程式 5-3

```
1 package com.abc.brokerage;
2 import com.abc.util.Date;
3
4 class StockAnalyzer {
5     private Date date;
6
7     double sell(Stock stock, int quantity) { }
8     public double buy(Stock stock, int quantity) { }
9 }
```

使用 static 關鍵字

static 關鍵字用於宣告和「類別」相關而和「類別實體」不相關的成員（包含欄位，方法，與巢狀類別）。

以下將會說明 **static** 關鍵字最常見的用法：類別專屬欄位和類別專屬方法。

類別專屬欄位

有時候需要宣告一個所有屬於同一個類別的物件實例都可以存取的欄位。例如：您可以使用這種欄位作為所有類別實例的溝通機制或是用來追蹤類別實例的產生數量（請看圖 5-5）。

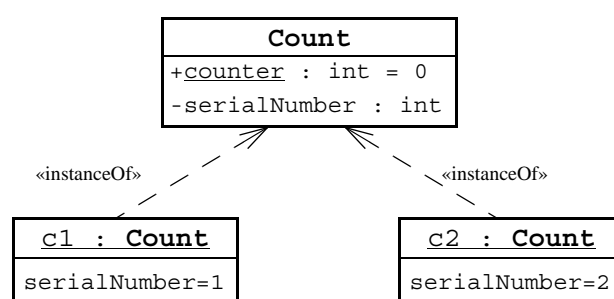


圖 5-5 **Count** 類別的 UML 物件圖以及兩個不同的 **Count** 類別實例

您可以使用 **static** 關鍵字達成分享欄位資料的效果。這類型的欄位有時稱為「類別專屬欄位 (class field)」，以便和「成員欄位 (member field)」或「實例欄位 (instance field)」區分。程式 5-4 呈現了類別欄位的使用範例。

程式 5-4 類別專屬欄位範例

```

1 public class Count {
2     private int serialNumber;
3     public static int counter = 0;
4
5     public Count() {
6         counter++;
7         serialNumber = counter;
8     }
9 }
  
```

這個範例中每個物件產生時都指定一個唯一的連續號碼，數字由 1 開始遞增。`counter` 欄位被所有的實例共同擁有，所以當建構子增加 `counter` 值後，下一個要產生的物件也可取得遞增後的值。`Static` 欄位某些方面就像其他語言的全域變數一樣。`Java` 程式語言沒有提供所謂的全域欄位 (`global field`)，但 `Static` 欄位可以被所有屬於同一類別的實例存取。

假如 `Static` 欄位沒有標記為 `private`，則您可以在該類別之外存取此欄位。如此一來即使沒有類別實例仍然可以透過類別名稱來存取它。如程式 5-5 所示。

程式 5-5 存取類別專屬欄位

```
1 public class OtherClass {
2     public void incrementNumber() {
3         Count.counter++;
4     }
5 }
```

類別專屬方法

有時候您沒有某個特定物件的實例，但仍需要使用到其程式碼。使用 `Static` 關鍵字來標記定義在類別中的方法可達到上述的效果，這樣的方法有時稱之為「類別專屬方法」。程式 5-6 的 `getTotalCount` 方法就是一個類別專屬方法的範例。

程式 5-6 類別專屬方法的範例

```
1 public class Count2 {
2     private int serialNumber;
3     private static int counter = 0;
4
5     public static int getTotalCount() {
6         return counter;
7     }
8
9     public Count2() {
10         counter++;
11         serialNumber = counter;
12     }
```

您應該透過類別名稱來呼叫 **static** 方法，而非透過物件的參考，如程式 5-7 所示：

程式 5-7 使用類別專屬方法 d

```

1  public class TestCounter {
2      public static void main(String[] args) {
3          System.out.println("Number of counter is "
4                              + Count2.getTotalCount());
5          Count2 count1 = new Count2();
6          System.out.println("Number of counter is "
7                              + Count2.getTotalCount());
8      }
9  }
```

TestCounter 程式輸出結果為：

```

Number of counter is 0
Number of counter is 1
```

由於您可直接呼叫一個類別的 **static** 方法而不需要產生實例，因此沒有 **this** 的存在。所以，**static** 方法除了區域變數、**static** 屬性，以及方法的參數外，無法存取任何的變數。如果試圖存取非 **static** 屬性時，則會導致編譯上的錯誤。沒有標示為 **static** 的屬性只能存在特定的類別實例中，所以也只能透過該實例的參考來存取，如程式 5-8 所示。

程式 5-8 未使用類別實例來參考非 **static** 變數的範例

```

1  public class Count3 {
2      private int serialNumber;
3      private static int counter = 0;
4
5      public static int getSerialNumber() {
6          return serialNumber; // 編譯錯誤！
7      }
8  }
```



注意 --main() 方法是 **static**，因為 JVM 在執行 main 方法時不會產生該類別的實例。所以假如您要存取成員資料，必須要產生物件後才能存取。

Static 初始化區塊

類別可以有 **static** 程式區塊，它們並不屬於正常方法的一部分。**static** 程式區塊在類別載入後只會被執行一次。假如類別有多個 **static** 區塊，則會依照出現的先後順序被執行。.

```
1  import com.mycompany.utilities.Database;
2  public class Count4 {
3      public static int counter;
4      static {
5          if (Database.tableExists("COUNTER")) {
6              counter = Database.getSingleField("COUNTER");
7          }
8      }
9  }

1  public class TestStaticInit {
2      public static void main(String[] args) {
3          System.out.println("counter = "+ Count4.counter);
4      }
5  }
```

static 初始化區塊會由 **Database** 取得已存在的 **COUNTER** 值（在這裏的套件、類別及方法都是虛構的），或是由 0 開始。

靜態引入 (Static Imports)

假如您要存取某類別中的 **static** 成員，您必須要指出這個成員位於哪個類別，就像程式 5-7 的第 3 和第 7 行一樣。Java SE 5.0 版後，Java 程式語言提供靜態引入的功能，可以在未指定類別名稱下直接呼叫 **static** 方法。程式 5-9 呈現了使用靜態引入的範例。其中第 2 行告訴編譯器編譯時需將 **static** 變數 **Math.PI** 放到符號表 (symbol table) [審註] 中。因此第 12 行和 16 行可以使用 **PI** 而不需要使用 **Math** 為前置的空間名稱。

程式 5-9 Circle 程式使用了靜態引入機制

```

1  package com.myproject.shapes;
2  import static java.lang.Math.PI;
3
4  public class Circle {
5      private double radius;
6
7      public Circle(double radius) {
8          this.radius = radius;
9      }
10
11     public double area() {
12         return PI*radius*radius;
13     }
14
15     public double circumference {
16         return 2*PI*radius;
17     }
18 }
```



警告 -- 請盡量避免使用靜態引入。當過度使用靜態引入功能，會讓程式難以理解和維護，並破壞了空間名稱的使用。程式的讀者（包含您在數個月後重新閱讀程式）將無法知道 **static** 方法是屬於那個類別。匯入一個類別的所有 **static** 方法也有會破壞程式的可讀性；假如您只需要一兩個 **static** 成員，請不要使用靜態引入。適當的運用可以移除重複的類別名稱，增加程式的可讀性。

建立陣列

單元重點

當完成這個單元後，您將能夠：

- 宣告陣列
- 產生陣列並參考陣列
- 初始化一個陣列
- 了解多維度陣列
- 了解陣列邊界 (array bounds) 與陣列長度是不能改變的特性

其他資源



其他資源 — 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O’ Reilly Media. 2005.

宣告陣列

陣列能夠將同樣型別的物件聚集在一起，讓您使用相同的名稱來參考一群物件。

您可以宣告任何資料型別的陣列：基本型別或類別型別。例如，宣告基本型別 **char** 的陣列，如下所示：

```
char[] s;
```

同樣的，您也可以宣告 **Point** 類別的陣列（如程式 6-1 所示）：

```
Point[] p; // Point 是個類別
```

程式 6-1 **Point** 類別

```
public class Point {  
    int x;  
    int y;  
    // 建構子  
    // 方法  
}
```

當您在變數的左邊使用中括號宣告陣列時，位在括號右方的變數都會受到這個括號的影響：

```
char[] myChars, yourChars, theirChars;  
Point[] myPoints, yourPoints, theirPoints;
```

在 **Java** 程式語言中陣列是一個物件，即使是由基本型別組成的陣列也一樣。當陣列由是類別型別組成時，陣列宣告並不會產生該類別的物件。反之，陣列的宣告會替您建立一個陣列的參考。陣列元素實際所用到的記憶體是經由 **new** 敘述或是陣列初始化來動態配置。您可以在變數名稱後使用中括號宣告陣列：

```
char s[];  
Point p[];
```

您看過上述兩種陣列宣告格式，建議您在使用時只選擇其中一種，並且之後都採取一致的宣告格式。宣告時並不需要指定陣列的長度。

建立陣列

與其它物件一樣，您可以使用 **new** 關鍵字來建立陣列。例如，產生一個 **char** 的陣列：

```
s = new char[26];
```

上述程式產生 26 個 **char** 的資料。陣列建立後，陣列元素會初始化為預設值（字元初始預設為 `'\u0000'`）。在您使用陣列前必須先把資料填入，例如：

```
1 public char[] createArray() {
2     char[] s;
3     s = new char[26];
4     for ( int i=0; i<26; i++ ) {
5         s[i] = (char) ('A' + i);
6     }
7     return s;
8 }
```

上述程式會在記憶體中產生一個陣列，並填入大寫英文字母。圖 6-1 呈現了記憶體中的陣列。

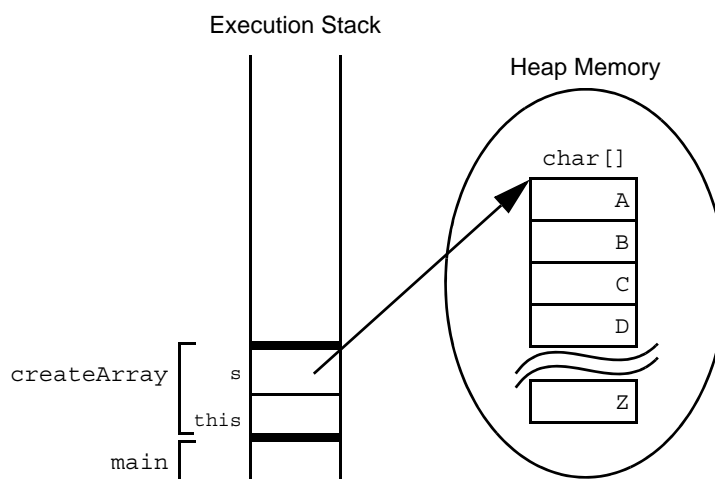


圖 6-1 產生 **char** 型別的陣列

陣列中的索引 (**index**) 是由 0 開始，且必需維持在合法的範圍內，也就是大於或等於 0 且小於陣列長度。使用任何超過這兩個索引來存取陣列元素都會造成執行時期例外 (**runtime exception**) 而產生錯誤。

由物件參考所組成的陣列

您可以使用同樣的語法，產生一物件陣列：

```
p = new Point[10];
```

上述程式產生 10 個指向 **Point** 物件參考。但是並沒有產生 10 個 **Point** 物件，必須使用下列方式來分別產生 10 個物件：

```
1 public Point[] createArray() {
2     Point[] p;
3
4     p = new Point[10];
5     for ( int i=0; i<10; i++ ) {
6         p[i] = new Point(i, i+1);
7     }
8     return p;
9 }
```

上述程式會在記憶中產生一個陣列，並填入 **Point** 的物件參考，如圖 6-2 所示。

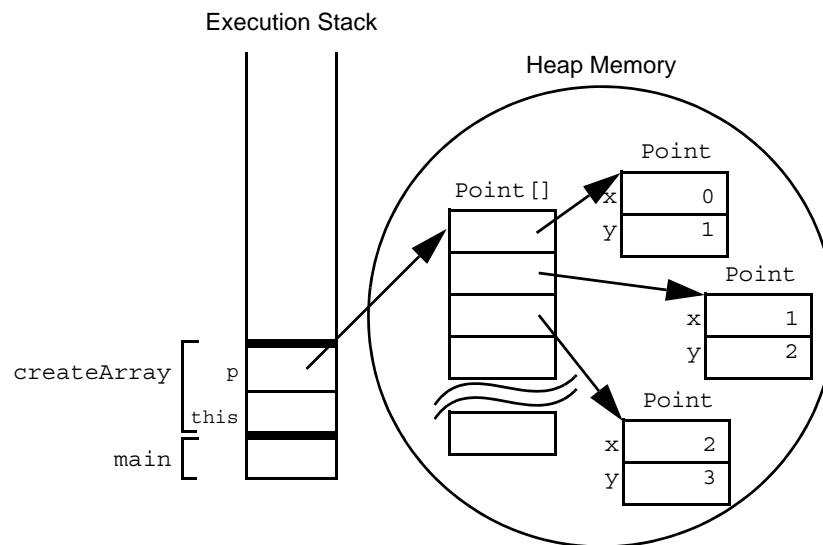


圖 6-2 建立包含 **Point** 物件參考的陣列

將陣列初始化

當您建立一個陣列後，陣列中每個元素就都已經被初始化了。就上一節提到的 **char** 陣列而言，每個元素均被初始化為 **null** (‘\u0000’) 字元。對上一節所提到的陣列 **p** 而言，每個元素均初始化為 **null**，這表示尚未填入任何 **Point** 物件的參考，經過 **p[0] = new Point()** 指定之後，第一個元素就會參考到真正的 **Point** 物件。



注意 -- 初始化所有變數與陣列元素，對系統的安全性來說相當重要。您無法在變數未初始前使用它們。

Java 程式語言可以使用簡便的方式在建立陣列時一併給予初始值：

```
String[] names = {  
    "Georgianna",  
    "Jen",  
    "Simon"  
};
```

上述程式相當於：

```
String[] names;  
names = new String[3];  
names[0] = "Georgianna";  
names[1] = "Jen";  
names[2] = "Simon";
```

您可以使用類似的簡便方式初始任何元素，例如：

```
MyDate[] dates = {  
    new MyDate(22, 7, 1964),  
    new MyDate(1, 1, 2000),  
    new MyDate(22, 12, 1964)  
};
```

多維度陣列

Java 程式語言處理多維陣列的方式和其他語言不一樣。因為您可以藉由任何資料型別作為基本的陣列宣告，接者就能夠再宣告該陣列的陣列，請參閱以下的二維陣列：

```
int [] [] twoDim = new int [4] [] ;  
twoDim[0] = new int [5] ;  
twoDim[1] = new int [5] ;
```

在第一行程式中，使用 **new** 產生了一個具有四個元素的陣列物件。每個元素都是 **null**，它必須參考到一個以 **int** 為基礎的陣列，您必須個別初始化每個元素並且填入參考位置。



注意 -- 雖然宣告的格式可以將中括號放置於變數名稱的左邊或右邊，但是這種彈性並不適用於所有陣列語法，例如：**new int [] [4]** 是不合法的。

因為可以個別初始化，所以您可以產生一非方正的多維陣列，例如您可以用下列的方式初始 **twoDim** 的每一個元素：

```
twoDim[0] = new int [2] ;  
twoDim[1] = new int [4] ;  
twoDim[2] = new int [6] ;  
twoDim[3] = new int [8] ;
```

因為這種初始方式的不夠簡潔，而且通常陣列大多數屬於矩形陣列，因此您可以使用簡便的語法產生 2 維陣列。例如，您可以用下列的程式產生 4 X 5 整數元素的陣列。

```
int [] [] twoDim = new int [4] [5] ;
```

陣列邊界 (Array Bounds)

Java 程式語言中，陣列的索引是由 0 開始。陣列元素的數量記錄於陣列物件的 **length** 屬性。假如存取陣列元素的索引超過邊界範圍，會產生執行時期錯誤。

程式 6-2 使用了 **length** 屬性來依序存取 **list** 陣列元素：

程式 6-2 使用陣列的 **length** 屬性依序存取陣列元素

```
public void printElements(int[] list) {  
    for ( int i = 0; i < list.length; i++ ) {  
        System.out.println(list[i]);  
    }  
}
```

使用陣列的 **length** 屬性可以讓程式易於維護，因為您可能無法預知在編譯時陣列中的元素數量有多少。

使用加強版的 for 迴圈

在程式中經常會去讀取陣列裡的每一個元素。在 **Java SE 5.0** 平台上新增了「加強版的 **for** 迴圈」功能，使得讀取陣列裡的每一個元素更加容易。如程式 6-3 所示。

程式 6-3 使用加強版的 **for** 迴圈讀取陣列裡的每一個元素

```
public void printElements(int[] list) {  
    for ( int element : list ) {  
        System.out.println(element);  
    }  
}
```

上述的 **for** 迴圈可以解讀為「依序讀取放在 **list** 裡的每一個元素，並將其值存入 **element** 變數中」。編譯器會自動處理這些的程式碼，因此程式 6-3 的效果等同於程式 6-2。

注意 — 使用加強版的 **for** 迴圈無法在依序存取陣列元素時修改個別元素。



陣列長度是無法改變的

在陣列產生之後，就不能改變其長度。然而，您可以使用同樣的物件參考指向另一個全新的陣列。

```
int[] myArray = new int[6];  
myArray = new int[10];
```

這個範例中，第一個陣列已經無法再被使用，除非在其它地方還有物件參考仍然指向它。

使用繼承 (Inheritance) 機制建立子類別

單元重點

當完成這個單元之後，您將能夠：

- 了解繼承概念
- 使用 **Java** 技術實作繼承機制
- 覆寫 **Object** 類別的方法
- 透過子類別產生多型
- 使用 **instanceof** 運算元
- 使用 **final** 關鍵字

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O'Reilly Media. 2005.

了解繼承概念

繼承是一種經由已存在的型別產生一或多個子型別的機制。在 **Java** 技術中，一個類別代表一種型別。繼承可以由已存在的類別產生子類別。在寫程式的時候，您會經常為了寫作某些東西的模型（例如，建立一個代表員工的類別），接著又會需要一個比原來更特殊的模型。例如，可能會需要一個代表經理的類別，經理事實上也是員工的一種，它算是一個具有額外特徵的員工。

圖 7-1 顯示了員工 (**Employee**) 和經理 (**Manager**) 類別的 UML 類別圖模型

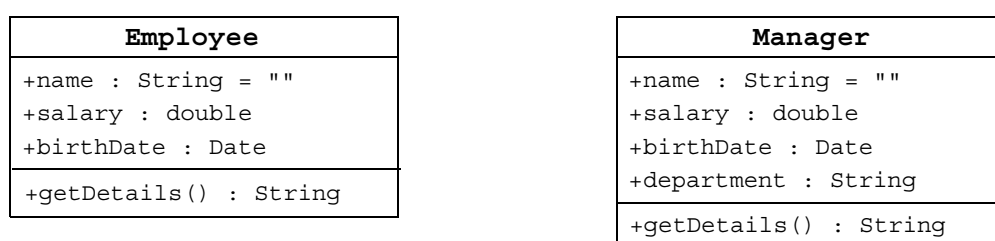


圖 7-1 **Employee** 和 **Manager** 類別的 UML 類別圖

程式 7-1 和程式 7-2 顯示了圖 7-1 中 **Employee** 和 **Manager** 類別可能的實作程式碼。

程式 7-1 **Employee** 類別可能的實作

```
import java.util.Date;
public class Employee {
    public String name = "";
    public double salary;
    public Date birthDate;

    public String getDetails() {...}
}
```

程式 7-2 **Manager** 類別可能的實作

```
import java.util.Date;
public class Manager {
    public String name = "";
    public double salary;
    public Date birthDate;
    public String department;

    public String getDetails() {...}
}
```

```

    }

```

由這個範例可看出 **Manager** 和 **Employee** 類別兩者有重複的資料。而且，可能有些方法可同時適用於 **Manager** 和 **Employee** 類別。因此，您需要一種能夠從已存在的類別中產生新類別的機制。物件導向語言提供了一種根據之前所制定的類別來定義新類別的機制。圖 7-2 是 **Manager** 繼承 **Employee** 類別的 UML 類別圖。

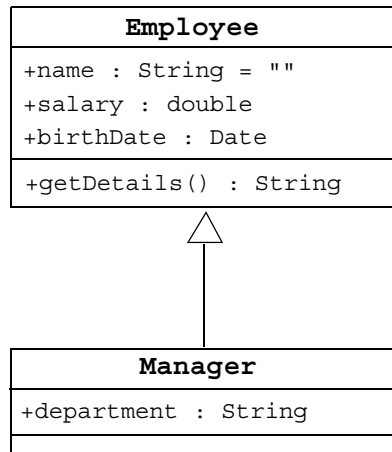


圖 7-2 **Manager** 繼承 **Employee** 類別的 UML 類別圖

程式 7-3 為圖 7-2 **Manager** 類別繼承 **Employee** 類別的實作程式碼。

程式 7-3 **Manager** 類別的另一種實作

```

public class Manager extends Employee {
    public String department;
}

```

了解繼承的優點

繼承具有下列的優點：

- 可以產生更為特殊的型別

Manager 類別繼承了 **Employee** 類別，所以可以說是一種變形或是特殊版本的 **Employee** 類別。這個特殊的類別除了具有父類別的所有特性，也同時具有額外的屬性。

- 消除重複的程式碼

如程式 7-3 所示，使用繼承可減少重複的程式碼（相較於程式 7-1 和程式 7-2）。

- 提高程式的可維護性

建立一組階層性類別之後，這群類別就具有單點修改 (**single point of touch**) 的維護性。假如受影響的類別是子類別，只有子類別需要修改，其餘的皆不變。若是受影響的是共同的父類別，也只有父類別需要修改。修改的部份也同樣會被所有的子類別繼承。

使用 Java 技術實作繼承機制

以下為使用 **Java** 技術建立子類別所需的步驟：

1. 選擇父類別。
2. 檢視父類別有哪些特性會被子類別繼承。
3. 宣告子類別。
4. 建立子類別特有的屬性和方法。
5. 需要時，覆寫父類別的方法。
6. 建立子類別所需的建構子。

以下會對每個步驟做更詳細的說明。



注意 -- **Java** 程式語言只允許一個類別繼承另一個類別。這個限制稱之為單一繼承。單一繼承和多重繼承的各項議題在物件導向程式設計師間有著廣泛的討論。**Java** 程式語言具有一種稱為介面的機制可提供了多重繼承的優點而沒有多重繼承缺點。介面的討論已超過本單元的範圍。

步驟 1：選擇父類別

圖 7-3 展示基底（父）類別 **Employee** 和 3 個子類別：**Engineer**，**Manager** 和 **Secretary**。**Director** 是 **Manager** 的子類別。

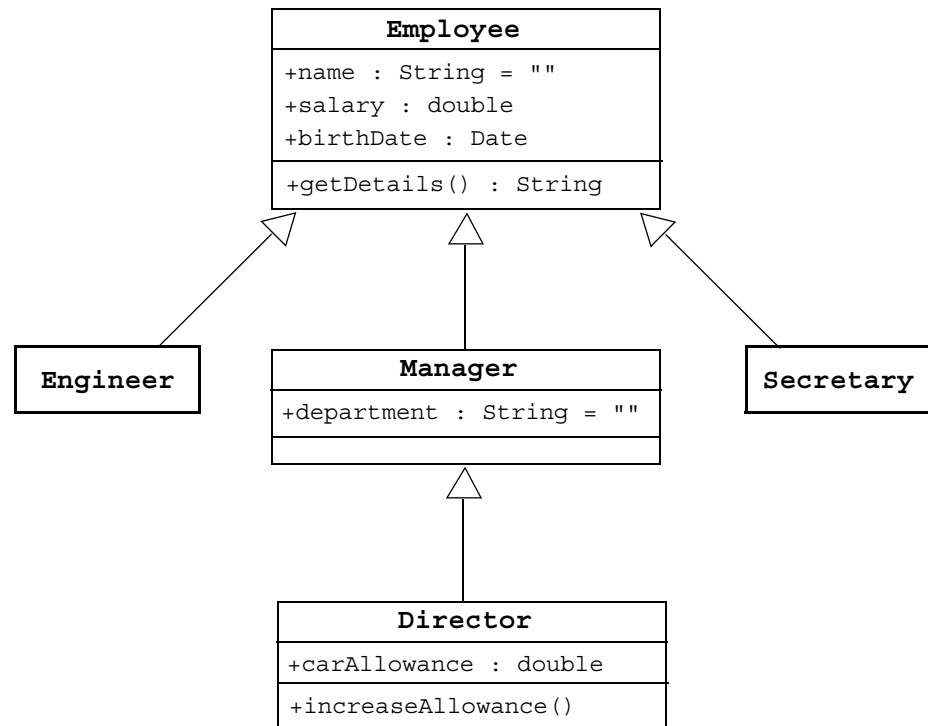


圖 7-3 繼承的範例

在圖 7-3 中，**Employee** 類別被選定為父類別，並產生 3 個子類別（**Engineer**，**Manager** 和 **Secretary**）。

此外在圖 7-3 中，**Manager** 類別也被選為父類別，被 **Director** 所繼承。

步驟 2：父類別哪些特性會被繼承

在這個步驟，您會檢視父類別並得知子類別會繼承哪些特性。程式 7-4 展示了精簡後的 **Employee** 類別。為了介紹相關議題，因此我們在程式 7-4 的變數、方法與建構子上使用了各種存取修飾詞。

程式 7-4 父類別：Employee

```
import java.util.Date;
public class Employee {
    protected String name;
    private double salary;
    protected Date birthDate;

    public Employee(String name) {}

    public void setSalary(double salary) {}
    double getSalary() {return salary;}

    public String getDetails() {
        return "Name: " + name + "\n"
            + "Salary: " + salary;
    }
}
```

表 7-1 列出了如何得知哪些屬性或方法可繼承至 **Employee** 類別的相關規則。

表 7-1 父類別屬性和方法的繼承規則

修飾詞	繼承規則	註解
private	繼承但無法存取	例如， Employee 類別的 salary 屬性，子類別無法存取
default	若是父類別和子類別在同一個套件中，則為繼承且可存取	例如， Employee 子類別也在同一套件中，則 getSalary 可被繼承和使用
protected	繼承且可存取	例如， protected 屬性 name 和 birthDate 為繼承和可存取
public	繼承且可存取	例如， public setSalary 方法為繼承和可存取



注意 -- 父類別的建構子無法被繼承。父類別建構子會在「步驟 6: 加入所需的建構子」中討論。

步驟 3: 宣告子類別

子類別的宣告使用 **extends** 關鍵字，如程式 7-5 所示。

程式 7-5 子類別的宣告範例

```
public class Manager extends Employee {  
    //  
}
```



注意 -- **Java** 程式語言只允許繼承至一個子類別。這個限制稱之為單一繼承。

步驟 4: 加入子類別特有的屬性和方法

在這個步驟，您會在子類別上加入獨特的屬性和方法。例如，程式 7-6 **Manager** 類別中具有額外的屬性 **department** 和相對應的 **get** 與 **set** 方法。

程式 7-6 Manager 類別的屬性和方法

```
public class Manager extends Employee {  
    protected String department;  
  
    public String getDepartment() {return department;}  
    public void setDepartment(String department) { }  
}
```

步驟 5：需要時，覆寫父類別方法

請參閱程式 7-6 的 **Manager** 類別，該類別並未定義 **getDetails** 方法。它由 **Employee** 類別繼承了 **getDetails** 方法（程式 7-7）。然而，**Employee** 類別並未提供任何 **Manager** 類別的屬性，例如 **department**。

程式 7-7 Employee 類別 getDetails 方法

```
1  import java.util.Date;  
2  public class Employee {  
3      protected String name;  
4      protected double salary;  
5      protected Date birthDate;  
6  
7      public Employee(String name) {}  
8  
9      public void setSalary(double salary) {}  
10     public double getSalary() {return salary;}  
11  
12     public String getDetails() {  
13         return "Name: " + name + "\n"  
14             + "Salary: " + salary;  
15     }  
16 }
```

這種情況下，**Manager** 類別需要有自己的 **getDetails** 方法，因此必須覆寫父類別的 **getDetails** 方法，如程式 7-8 所示。

程式 7-8 **Manager** 類別覆寫父類別的 **getDetails** 方法

```

1  public class Manager extends Employee {
2      protected String department;
3
4      public String getDepartment() {return department;}
5      public void setDepartment(String department) { }
6
7      @Override
8      public String getDetails() {
9          return "Name: " + name + "\n"
10             + "Salary: " + salary + "\n"
11             + "Manager of: " + department;
12     }
13 }
```

Java 技術規範中定義了宣告覆寫方法的所需遵守的規則。

- 方法介面 (method interface) 要一樣

假如子類別定義的方法名稱，回傳型別及參數都和父類別一樣，此新方法便是覆寫了 (override) 舊的方法。自 **Java SE 5.0** 平臺之後，這個規則有些許改變。子類別所覆寫的父類別方法，其回傳型別可以是父類別方法回傳型別的子類別。稱之為 **covariant return**。

- 選擇性的 **Override** 標註 (Annotation)

自 **Java SE 5.0** 平臺之後，您可以使用 **override** 標註告訴編譯器您要覆寫父類別方法。儘管 **override** 標註是可有可無的，但它可以幫助我們防止錯誤。假如方法被標註為 **override**，但沒有正確的覆寫父類別方法時，編譯器會產生錯誤訊息。有關標註的詳細資訊請參考下列連結：

<http://java.sun.com/javase/6/docs/technotes/guides/language/annotations.html>

- 覆寫方法不可以降低可存取範圍

子類別方法透過存取修飾詞所宣告的可存取範圍不可以比所要覆寫的父類別方法小。

例如程式 7-9 是錯誤的程式碼。

程式 7-9 無效修飾詞的設定

```

public class Parent {
    public void doSomething() {}
}
```

```
}

public class Child extends Parent {
    private void doSomething() {} // 不合法
}
```

呼叫被覆寫的方法

子類別可以使用 **super** 關鍵字呼叫父類別的方法。 **super** 關鍵字指的是該類別的父類別。適用於引用父類別的屬性和方法。

子類別可以使用 **super** 關鍵字來呼叫被它所覆寫的方法，如下所示：

```
import java.util.Date;
public class Employee {
    private String name;
    private double salary;
    private Date birthDate;

    public String getDetails() {
        return "Name: " + name + "\nSalary: " + salary;
    }
}

public class Manager extends Employee {
    private String department;

    public String getDetails() {
        // 呼叫父類別
        return super.getDetails()
            + "\nDepartment: " + department;
    }
}
```

super.method() 的呼叫可能伴隨了副作用。因為某些方法可能會被重複呼叫。被呼叫的方法不一定是在父類別所定義，它可能是在整個繼承體系中的任何一個類別。

注意 -- 上個例子中，成員屬性被宣告為 **private**。這不是必須的，但是個好的撰寫習慣。



步驟 6：加入所需的建構子

假如您為子類別提供建構子，那它的目的就是為了初始化子類別的屬性值。建構一個子類別的實例之前，必須先建立一個父類別實例並加以初始化。例如，建構 **Manager** 類別實例時，會先產生 **Employee** 類別實例。當您在子類別宣告建構子時，必須注意下列事項：

- 除非父類別正在開發中，否則不應該修改父類別，或在父類別中加入新的建構子。
- 子類別建構子必須負責呼叫適當的父類別建構子，不管是以明確的或隱含的方式。
- 父類別的欄位必須比子類別的欄位先初始化。

您可以使用下列步驟宣告子類別的建構子：

步驟 A：分析父類別的建構子

這一個小節中展示三種父類別 **Employee** 的定義方式。每一種皆具有不同的建構子，程式 7-10 為由編譯器提供預設建構子的範例。

程式 7-10 父類別只有無參數的建構子

```
1 public class Employee {
2     private String name;
3     // 方法
4 }
```

程式 7-11 範例中父類別沒有無參數的建構子，但明確定義了一個有參數的建構子。

程式 7-11 父類別沒有無參數的建構子

```
1 public class Employee {
2     private String name;
3     public Employee(String name) {
4         this.name = name;
5     }
6 }
```

程式 7-12 範例中父類別定義無參數的建構子以及有參數的建構子。

程式 7-12 父類別定義無參數以及有參數的建構子

```
1 public class Employee {
2     private String name;
3     public Employee() { // 程式碼 }
```

```
4     public Employee(String name) {
5         this.name = name;
6     }
7 }
```

步驟 B: 宣告子類別的建構子

當您要將子類別的變數初始化時，應該要宣告子類別的建構子。程式 7-13 是 **Employee** 子類別宣告建構子的範例。

程式 7-13 具有建構子的子類別

```
1 public class Manager extends Employee {
2     private String department;
3     public Manager(String department) {
4         this.department = department;
5     }
6     // 以下程式略
7 }
```

步驟 C: 經由子類別建構子呼叫適當父類別建構子

在這個步驟中，您需要在初始化子類別屬性前先初始化父類別的屬性。實際上的意義是在執行子類別的建構子前，父類別的建構子要先執行。這樣可能需要修改前一個步驟所定義的子類別建構子。修改的部份包含：

- 對於每個子類別建構子，決定其要先呼叫的父類別建構子
- 假如先呼叫的父類別建構子是有參數的，則子類別需將這些參數值傳給父類別建構子。可於子類別建構子程式的第一行使用 **super** (<arg-list>) 來呼叫父類別建構子以傳遞參數。



注意 -- 明確呼叫父類別建構子時有一個例外，您不需要明確呼叫無參數的建構子或是父類別的預設建構子。編譯器會自動將 **super** 或 **this()** 加在子類別建構子的第一行。

如程式 7-14 所示，**Manager** 類別程式的第 4 行包含了一個選擇性的父類別建構子呼叫，明確呼叫了 **Employee** 類別的無參數建構子（或預設建構子）。

程式 7-14 範例 1：明確呼叫父類別無參數建構子

```

1  public class Employee {
2      private String name;
3      // 方法
4  }

1  public class Manager extends Employee {
2      private String department;
3      public Manager(String department) {
4          super();           // 注意：不一定要明確的使用 super
5          this.department = department;
6      }
7  }
```

如程式 7-15 所示，在 **Manager** 類別程式的第 4 行，子類別明確呼叫了父類別 **Employee** 具參數的建構子。請注意 **Manager** 類別的建構子傳遞參數給父類別建構子。

程式 7-15 範例 2：明確呼叫父類別具參數的建構子

```

1  public class Employee {
2      private String name;
3      public Employee(String name) {
4          this.name = name;
5      }
6  }

1  public class Manager extends Employee {
2      private String department;
3      public Manager(String name, String department) {
4          super(name);       // 注意：需要明確使用 super
5          this.department = department;
6      }
7  }
```

如程式 7-16 所示，在 **Manager** 類別程式的第 4 行，子類別明確呼叫了父類別 **Employee** 具有參數的建構子。這種情況下，如果您沒有使用 **super(name)** 明確呼叫父類別建構子，編譯器不會有任何錯誤訊息。編譯器會自動加上父類別無的參數建構子。雖然可以編譯，但是就應用程式的觀點來看仍然是錯誤。

程式 7-16 範例 3：父類別具有無參數以及有參數建構子

```
1 public class Employee {
2     private String name;
3     public Employee() { // 程式碼 }
4     public Employee(String name) {
5         this.name = name;
6     }
7 }

1 public class Manager extends Employee {
2     private String department;
3     public Manager(String name, String department) {
4         super(name); // 注意：移除此行仍可以編譯
5         this.department = department;
6     }
7 }
```

複雜的建構子鏈結 (Chaining) 範例

許多類別，包括父類別和子類別都具有多載 (overload) 的建構子。這個情況下可使用 **this** 關鍵字來鏈結這些建構子，如程式 7-17 所示：

程式 7-17 範例 4: **Employee** 和 **Manager** 類別使用建構子鏈結

```

1  public class Employee {
2      private String name;
3      private double salary;
4
5      public Employee(String name, double salary) {
6          this.name = name;
7          this.salary = salary;
8      }
9
10     public Employee(String name) {
11         this(name, 15000.00);
12     }
13 }

1  public class Manager extends Employee {
2      private String department;
3
4      public Manager(String name, double salary, String department) {
5          super(name, salary);
6          this.department = department;
7      }
8
9      public Manager(String name, String department) {
10         super(name);
11         this.department = department;
12     }
13
14     public Manager(String name) {
15         this(name, "***NO DEPARTMENT YET**");
16     }
17 }

```

假如有外部程式要產生 **Manager** 實例如下所示：

```
Manager m = new Manager("Joe Soap");
```

然後一連串的事件發生順序如下：

- 先開始於 **Manager** 類別建構子第 14 行
- 傳送 **name** 和 **department** 值到 **Manager** 第 9 行建構子
- 呼叫 **Employee** 第 10 行建構子並傳入參數值 **name**
- 傳送 **name** 和 **salary** 值到 **Employee** 第 5 行建構子
- 自動呼叫 **Employee** 的父類別 (**Object** 類別) 的無參數建構子
- 執行 **Employee** 第 6 行，儲存 “Joe Soap” 到 **name** 欄位
- 執行 **Employee** 第 7 行，儲存 15000.00 到 **salary** 欄位
- 回到 **Manager** 執行第 11 行，儲存 “**NO DEPARTMENT YET**” 到 **department** 欄位



注意 --**super()** 只有在建構子沒有呼叫任何父類別建構子下才會自動加入。若要在建構子中使用 **this** 或 **super** 關鍵字，必須在第一行就要加入。

多型 (Polymorphism)

以 **Manager** 「is a」 **Employee** 來描述 **Manager** 和 **Employee** 兩者間個關係，事實上不只是方便而已。回想一下，**Manager** 擁有了父類別 **Employee** 的所有成員包括屬性和方法。這表示所有對 **Employee** 上正當合法操作都適用於 **Manager**。假如 **Employee** 有 **getDetails** 方法，則 **Manager** 也有。

產生一 **Manager** 實例，然後將其參考指定給 **Employee** 型別的變數，這個行為看起來似乎不實際，然而這是合法的，而且有很多情況您會需要這個功能。

一個物件只有一個型別：被建構時的型別。然而變數是多型的，因為它們可指向不同型別的物件。

就像大部分的物件導向程式語言一樣，**Java** 程式語言允許變數指向具有相同父類別的物件。所以您可以說：

```
Employee e = new Manager(); // 合法
```

使用變數 **e**，您只能使用屬於 **Employee** 部分特性的物件，屬於 **Manager** 特有的則被隱藏了。就編譯器而言，這是因為 **Employee** 不是 **Manager**。因此下面的程式是不允許的：

```
// 非法指定 Manager 欄位
e.department = "Sales"; // 假設 department 宣告為 public 欄位
// 雖然是 Manager 物件，
// 但變數宣告為 Employee 型態
```

虛擬方法調用 (Virtual Method Invocation)

我們先看一下底下這個案例：

```
Employee worker = new Employee();
Manager boss = new Manager();
```

假如您呼叫 **worker.getDetails()** 和 **boss.getDetails()**，會導致不同的行為。**Employee** 物件執行 **Employee** 類別的 **getDetails()** 方法，而 **Manager** 物件則執行 **Manager** 類別的 **getDetails()**。

如果為下列程式，則到底是呼叫誰的方法則較不明顯：

```
Employee worker = new Manager();
worker.getDetails();
```

例如方法中的參數或是異質集合中的物件。

事實上，您可以在執行時期變數所參考到的物件的行為，而非編譯時期決定的。這是多型的其中一種面相，也是物件導向語言的重要特性。這種行為稱之為虛擬方法調用 (virtual method invocation)。

在上一個範例中 `worker.getDetails()` 呼叫的是 `Manager` 類別中的方法。

多型和 `static` 方法

您不可以覆寫 `static` 方法，因為要覆寫的方法必須非 `static` 方法，但您可以隱藏它。兩個具有相同簽章的方法在同一個類別階層中代表兩個獨立的類別方法。假如在物件中使用 `static` 方法，所呼叫的方法即為所宣告類別的所屬方法。

例如：

```
// 假設 ClassB 繼承 ClassA
// 假設 ClassA 有 static 方法 staticMethod
// 假設 ClassB 宣告 static 方法 staticMethod 和 ClassA 的一樣
ClassA obj = new ClassB();
obj.staticMethod(); // classA.staticMethod 會被呼叫
```

異質 (Heterogeneous) 集合

您可以產生一群有具相同類別的物件。這樣的群集 稱之為同質 (Homogeneous) 集合。例如：

```
MyDate[] dates = new MyDate[2];
dates[0] = new MyDate(22, 12, 1964);
dates[1] = new MyDate(22, 7, 1964);
```

Java 程式語言有個 **Object** 類別，可以群集任意型別的物件，因為他們都繼承了 **Object**。這樣的群集稱之為異質集合。異質集合是一群所屬類別不相同的物件。在物件導向語言中，您可以產生一群物件，這些物件具有共同的祖先類別，**Employee** 類別。例如：

```
Employee[] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

您甚至可以撰寫用來排序的方法，不管是哪一種員工，依照員工的年齡或薪資來排序。



注意 -- 所有的類別都是 **Object** 的子類別，所以您可以產生一 **Object** 陣列，由不同型別的物件構成。唯一不能加到 **Object** 陣列的是基本型別變數。然而您仍可以使用包覆類別來產生物件。

多型參數

您可以撰寫接受任何物件為其參數的方法（在這個例子為 **Employee** 類別），且傳入物件也可以是此參數物件的子類別。您或許會於應用程式的類別中產生一個方法，這個方法傳入 **employee** 然後比較它的薪資，決定其課稅。使用多型特性，您可以這樣作：

```
public class TaxService {
    public TaxRate findTaxRate(Employee e) {
        // 計算後回傳課稅比率
    }
}
```

```
// 同時，在程式的某個地方...
TaxService taxSvc = new TaxService();
Manager m = new Manager();
TaxRate t = taxSvc.findTaxRate(m);
```

上面的敘述是合法的，因為 **Manager** 是一種 **Employee**。但是 **findTaxRate** 方法只能存取定義於 **Employee** 類別的成員（屬性和方法）。

instanceof 運算子

您可以使用父類別的變數參考來傳遞子類別物件，但有時候您會想知道真正的物件是什麼。這時候就可以用 **instanceof** 運算子。假設您有下列的類別階層：

```
public class Employee
public class Manager extends Employee
public class Engineer extends Employee
```

假如您收到一個物件是使用 **Employee** 類別的參考變數，它可能是 **Manager** 或 **Engineer** 物件。您可以使用 **Employee** 來測試它的型別，如下所示：

```
public void doSomething(Employee e) {
    if ( e instanceof Manager ) {
        // 處理 Manager
    } else if ( e instanceof Engineer ) {
        // 處理 Engineer
    } else {
        // 處理其它種類的 Employee
    }
}
```

強制轉型物件參考

某些情況下您接收到父類別的參考，您已經使用 **instanceof** 運算子知道其型別，此時您便可以強制轉型物件而達到使用該物件的所有功能的目的。

```
public void doSomething(Employee e) {  
    if ( e instanceof Manager ) {  
        Manager m = (Manager) e;  
        System.out.println("This is the manager of "  
                             + m.getDepartment());  
    }  
    // 其它操作  
}
```

假如您沒有強制轉型，執行 **e.getDepartment()** 便不會成功，因為編譯器無法找到 **Employee** 類別的 **getDepartment** 方法。

假如您沒有先用 **instanceof** 作測試，則冒險強制轉型可能會失敗。通常強制轉型前會有幾項測試：

- 向上強制轉型是被允許的，而且不需要使用強制轉型的運算子。直接透過平常的方式設值即可。
- 向下強制轉型，至少需通過編譯器的檢查確認是可行的。例如，將 **Manager** 參考轉型為 **Engineer** 是不允許的，因為 **Manager** 不是一種 **Engineer**。有效的強制轉型必須是目前參考變數型別的子類別。
- 假如編譯器允許強制轉換，則物件型別是在執行時期檢查。例如，若沒有事先以 **instanceof** 檢查型別，就強制轉換型別，結果物件的類別不同於要轉換的類別，此時就產生執行時期例外 (**runtime exception**)。例外 (**exception**) 是下一章的主題。

Object 類別

於 Java 程式語言中 **Object** 類別是所有類別的根類別。假如宣告類別時未使用 **extends**，則編譯器會自動加上內定的 **extends Object** 到宣告類別的地方。例如：

```
public class Employee {  
    // 程式省略  
}
```

等同於：

```
public class Employee extends Object {  
    // 程式省略  
}
```

如此一來就可以覆寫 **Object** 類別的多個方法。以下介紹兩個 **Object** 類別的重要方法。

Equals 方法

== 運算子執行相等的比較。假如有兩個參考變數 *x* 和 *y*，假如 *x* 和 *y* 參考同一個物件（位址相同）則 *x==y* 為 **true**。

Object 類別位於 **java.lang** 套件中，具有 **public boolean equals(Object obj)** 方法，用於比較兩個物件是否相同。若未覆寫此方法，則預設只有在兩個參考變數參考同一個物件（位址相同）時才會傳回 **true**。

但 **equals()** 方法的目的是在於儘可能比較物件的「內容」是否相等。這就是為什麼此方法常被覆寫的原因。例如 **String** 類別，當兩個 **String** 物件不是 **null** 且具有相同順序的字元則 **equals()** 方法會傳回 **true**。



注意 -- 當您覆寫 **equals** 方法時也應該覆寫 **hashCode** 方法。一種簡單的 **hashCode** 實作方式是針對會被測試是否相等的物件屬性的 **hashCode** 做位元層級的 **XOR(bitwise XOR)** 運算。

Equals 方法範例

程式 7-18 顯示了 **equals** 方法範例，其中 **MyDate** 類別已經修改過，且具有 **equals** 方法來檢驗年月日的屬性值是否相同。

程式 7-18 **equals** 和 **hashCode** 方法範例

```

1  public class MyDate {
2      private int day;
3      private int month;
4      private int year;
5
6      public MyDate(int day, int month, int year) {
7          this.day    = day;
8          this.month  = month;
9          this.year   = year;
10     }
11
12     @Override
13     public boolean equals(Object o) {
14         boolean result = false;
15         if ( (o != null) && (o instanceof MyDate) ) {
16             MyDate d = (MyDate) o;
17             if ( (day == d.day) && (month == d.month)
18                 && (year == d.year) ) {
19                 result = true;
20             }
21         }
22         return result;
23     }
24
25     @Override
26     public int hashCode() {
27         return (day<<4 ^ month ^ year);
28     }
29 }

```

hashCode 方法

程式 7-18 展示 **hashCode** 方法的實作方式，對日期屬性使用位元層級的 **XOR** 運算。這樣可以保證如果具有相同的雜湊碼的 **MyDate** 物件，必具有相同屬性值。如果 **hashCode** 不同則具有不同屬性。

以下程式（程式 7-19）測試兩個不同 **MyDate** 物件，但其年月日屬性值相同。

程式 7-19 Hash code 測試

```

1  class TestEquals {
2      public static void main(String[] args) {
3          MyDate date1 = new MyDate(14, 3, 1976);
4          MyDate date2 = new MyDate(14, 3, 1976);
5
6          if ( date1 == date2 ) {
7              System.out.println("date1 is identical to date2");
8          } else {
9              System.out.println("date1 is not identical to date2");
10         }
11
12         if ( date1.equals(date2) ) {
13             System.out.println("date1 is equal to date2");
14         } else {
15             System.out.println("date1 is not equal to date2");
16         }
17
18         System.out.println("set date2 = date1;");
19         date2 = date1;
20
21         if ( date1 == date2 ) {
22             System.out.println("date1 is identical to date2");
23         } else {
24             System.out.println("date1 is not identical to date2");
25         }
26     }
27 }

```

上述範例輸出下列的結果：

```

date1 is not identical to date2
date1 is equal to date2
set date2 = date1;
date1 is identical to date2

```

toString 方法

`toString` 方法將物件轉換成以 `String` 格式。編譯器會判斷必須使用 `String` 的自動轉換機制時，就會呼叫此方法。例如，`System.out.println()` 的呼叫：

```
Date now = new Date();  
System.out.println(now);
```

等同於：

```
System.out.println(now.toString());
```

`Object` 類別有定義預設的 `toString` 方法，該方法會回傳類別名稱和 `hash code`。大多的類別會覆寫 `toString` 方法以提供有用的資訊。例如所有的包圍類別都覆寫 `toString` 方法來提供他們所代表的值。有些類別即使沒有字串型式的表示方式，也會覆寫 `toString` 方法來提供資訊以利程式除錯。

final 關鍵字

Java 技術規範中有定義 **final** 關鍵字，可用於類別，方法和變數。被標示為 **final** 的語言元素就會成為不可改變 (immutable)，下面提供了使用 **final** 關鍵字更詳細的說明。

Final 類別

Java 程式語言允許您對類別使用 **final** 關鍵字。這樣表示此類別不可被繼承。例如 **java.lang.String** 類別就是一個 **final** 類別。這麼做的原因是基於安全因素考量，確保參考到 **String** 類別實體的方法所持有的都是 **String** 類別的實體，而非被修改過的 **String** 子類別的實體。

Final 方法

您可以標示方法為 **final**。標示為 **final** 的方法不可被覆寫。有時基於安全因素，我們不希望方法的實作被修改，或有時為了怕方法被覆寫時會會影響到合理的物件狀態，此時我們應將該方法標示為 **final**。

將方法標示為 **final** 不會有效能上的影響。編譯器會直接在呼叫方法的地方加入 **final** 方法的程式碼，而非執行時期的虛擬方法呼叫。假如方法標示為 **static** 或 **private** 則可以使用 **final** 來利用編譯器做最佳化，因為這兩種方法不會有動態聯結的狀況。

Final 變數

假如變數標示為 **final**，其效果就是如同宣告為常數。任何意圖要改變此變數值的行為，都會讓編譯產生錯誤。如下面範例所示，使用 **final** 變數宣告屬性：

```
public class Bank {  
    private static final double  DEFAULT_INTEREST_RATE=3.2;  
    // 省略  
}
```

注意 -- 假如您宣告 **final** 變數來參考物件，則此變數不可以變更參考，來指向其他物件。但是您可以改變物件的內容，因為只有變數本身是 **final**。



空白 **Final** 變數

空白 **final** 變數為宣告 **final** 變數時未初始化，初始化被延遲。空白 **final** 變數必須在物件實例初始尚未結束前被指派一個值。要達到這個目的，空白 **final** 變數必須在所有的建構子中指派值，且只能指定一次。假如空白 **final** 變數是個區域變數，則可於任何時間點於方法中指定，但只能指定一次。下面程式碼片斷展示了如何於類別中使用空白 **final** 變數：

```
public class Customer {
    private final long  customerID;

    public Customer() {
        customerID = createID();
    }
    public long getID() {
        return customerID;
    }
    private long createID() {
        return ... // 產生新的 ID
    }
    ... // 省略
}
```


使用例外 (Exception) 類別和 Assertions 進行錯誤處理

單元重點

當完成這個單元之後，您將能夠：

- 定義例外和 **Assertion**
- 使用 **try-catch** 敘述句、多個 **catch** 子句和 **finally** 子句進行例外處理。
- 描述常見的例外
- 了解例外處理或宣告的規則
- 了解方法覆寫和例外
- 產生自訂例外
- 使用 **Assertions**

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O' Reilly Media. 2005.

例外 (Exceptions) 和 Assertions

例外的機制在多種程式語言中都有提供，用於描述發生非預期的狀況時要怎麼處理。典型的例外是由某些錯誤所造成，例如當呼叫方法時傳遞了錯誤的參數，或是網路連線失敗，開啟不存在的檔案等等。

Assertions 是用於測試程式邏輯的一種方法。例如，您覺得某個時間點某個變數應該是正數，您便可使用 **Assertions** 來測試。**Assertions** 可用於測試類別方法中的程式邏輯，但通常不會用來測試外部程式。**Assertions** 的一個重要特性是，您可以在執行時期將它們全部由程式中移除。這表示您可以於開發階段開啟 **Assertion** 功能，最後產品交貨給客戶時將 **Assertions** 移除。這是例外和 **Assertion** 最重要的不同點。

例外

必須檢查的例外為開發人員預期可能發生的例外並加以處理。例如，找不到檔案，網路斷線等。

必須檢查的例外為開發人員預期可能發生的例外並加以處理。例如，找不到檔案，網路斷線等。

可不檢查的例外為程式的臭蟲或是狀況難以處理。所謂「可不檢查」是因為當它們發生時，您不需要檢查或是處理。通常例外發生可能是程式有臭蟲，這種例外稱之為執行時期例外 (**runtime exceptions**)，例如存取陣列超過其邊界範圍 (**index out of bound**)。

如果例外的發生是由於環境因素而且無法回復，則稱之為錯誤 (**errors**)。例如記憶體耗盡。錯誤屬於可不檢查的例外。

Exception 類別為必須檢查的和可不檢查的例外的基底類別。與其發生例外而結束程式，不如於程式中處理例外並繼續運行。

Error 類別為某些可不檢查例外的基底類別，在這種情況下，程式無法從嚴重錯誤狀況回復。在大部分的情況下，假如遇到這類的嚴重錯誤，您應該要停止程式，當然您可以試著盡量保存使用者的作業狀態。

RuntimeException 類別為另一些可不檢查例外的基底類別，此類例外可能是由於程式臭蟲引起。大部分的案例中，您應該要停止程式，當然試著盡量保存使用者的作業狀態也是好的。

當例外發生，可以在發現錯誤的方法中處理此例外，或是丟回 (throw) 給呼叫此方法者，告知有問題發生。然後呼叫此方法者可有同樣的選擇：處理例外或是丟回給呼叫者。假如例外已經回到最外層的執行緒，執行緒會被結束。這種設計提供開發人員撰寫處理程式 (handler) 在適當的地方處理例外。

您可以經由瀏覽 API 決定方法丟出的必須檢查例外。有時您會發現文件中會描述多個可不檢查例外，這種情況不多，但您不可以依賴這些資訊。

例外的範例

程式 8-1 展示加總輸入值的程式。

程式 8-1 丟出例外的程式範例

```
1  public class AddArguments {
2      public static void main(String args[]) {
3          int sum = 0;
4          for ( int i = 0; i < args.length; i++ ) {
5              sum += Integer.parseInt(args[i]);
6          }
7          System.out.println("Sum = " + sum);
8      }
9  }
```

如果所有的輸入值都是整數，那程式便沒問題。例如：

```
java AddArguments 1 2 3 4
Sum = 10
```

如果輸入值不是整數就會失敗：

```
java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException:
For input string: "two"at
java.lang.NumberFormatException.forInputString(NumberFormatE
xception.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AddArguments.main(AddArguments.java:5)
```

try-catch 敘述句

Java 提供了得知那個例外被丟出，以及如何回復的機制。程式 8-2 展示修改過的 `AddArguments` 程式，使用 `try-catch` 敘述句捕捉例外。這個例子為 `java.lang.NumberFormatException`。

程式 8-2 處理例外的程式範例

```
1 public class AddArguments2 {
2     public static void main(String args[]) {
3         try {
4             int sum = 0;
5             for ( int i = 0; i < args.length; i++ ) {
6                 sum += Integer.parseInt(args[i]);
7             }
8             System.out.println("Sum = " + sum);
9         } catch (NumberFormatException nfe) {
10            System.err.println("One of the command-line "
11                               + "arguments is not an integer.");
12        }
13    }
14 }
```

程式捕捉到例外後將程式停止並顯示訊息。

```
java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```


使用更小的例外處理區塊

try-catch 敘述句可以用於一小塊的程式中。程式 8-3 展示了如何將 **try-catch** 敘述句用於 **for** 迴圈內。如此一來可以使程式跳過非整數的輸入值。這個程式展示了比 **AddArguments2** 更適當的例外處理。

程式 8-3 精煉後的 **AddArguments** 程式

```

1  public class AddArguments3 {
2      public static void main (String args[]) {
3          int sum = 0;
4          for ( int i = 0; i < args.length; i++ ) {
5              try {
6                  sum += Integer.parseInt(args[i]);
7              } catch (NumberFormatException nfe) {
8                  System.err.println "[" + args[i] + "] is not an integer"
9                      + " and will not be included in the sum.");
10             }
11         }
12         System.out.println("Sum = " + sum);
13     }
14 }
```

這個程式捕捉了每一個非整數的例外，產生警告訊息。這個程式仍然加總其它有效的整數值

```

java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the sum.
[3.0] is not an integer and will not be included in the sum.
Sum = 5
```

使用多個 catch 子句

在 **try** 區塊之後可以有許多個 **catch** 區塊，每個區塊處理不同的例外型別。程式 8-4 展示了使用多個 **catch** 子句的語法。

程式 8-4 使用多個 **catch** 子句的範例

```
try {  
    // 程式可能會丟出多個例外  
} catch (MyException e1) {  
    // 假如是 MyException 則執行這部份程式  
} catch (MyOtherException e2) {  
    // 假如是 MyException 則執行這部份程式  
} catch (Exception e3) {  
    // 其它的 Exception 執行這部份程式  
}
```

假如在 **try** 區塊有多個 **catch** 子句，而順序錯誤的 **catch** 子句會造成編譯錯誤。因為例外處理是由 **try** 區塊後可處理的第一個 **catch** 子句負責。在這個範例中，如果將 **Exception** 子句放到第一個，則該子句將處理所有的例外，後面緊接著的 **MyException** 或 **MyOtherException** 將永遠不會被呼叫。

呼叫堆疊 (Call Stack) 機制

假如程式丟出例外，而例外沒有立即在方法中處理，例外會被丟到呼叫端的方法之中。假如又沒有被呼叫端的方法處理，又會被丟到呼叫此方法者。這個程序會一直發生，假如一直到了 **main()** 方法還沒有處理且 **main()** 方法中也未處理，會讓程式不正常停止。

請考慮下面這個例子，在 **main()** 方法呼叫 **first()** 方法，**first()** 方法有呼叫 **second()** 方法。假如在 **second()** 方法發生例外，而它也未處理，則會都回至 **first()** 方法。想像一下 **first()** 方法有捕捉到該例外，則例外不會繼續往上丟。然而如果 **first()** 方法沒有捕捉該例外，那在位於呼叫堆疊的下一個方法 **main()**，會被檢查是否有捕捉處理該例外，如果沒有例外訊息會輸出到標準輸出裝置，則程式會停止。

Finally 子句

Finally 子句所定義的程式區塊「一定」會被執行，即便是有例外發生且未處理。下面的範例來自於 **Frank Yellin Low Level Security in Java**:

```
1  Hose myHose = new Hose();
2  try {
3      myHose.startFaucet();
4      myHose.waterLawn();
5  } catch (BrokenPipeException e) {
6      logProblem(e);
7  } finally {
8      myHose.stopFaucet();
9  }
```

這個例子，不管例外有無發生，當開啟水龍頭，灑水後都會關閉。在 **try** 大括弧後的程式稱為被保護 (**protected**) 的程式。只有一種狀況會防止 **finally** 子句的程式被執行，就是虛擬機器關閉（例如執行 **System.exit** 方法，或是關機）。這樣表示控制流程可能脫離正常的執行順序。例如，在 **try** 區塊如果包含 **return** 敘述，則 **finally** 子句會先執行然後再才回傳。

例外的分類

本章節前面有介紹將例外分為兩大類。下面將會檢視這兩個分類的類別階層。`java.lang.Throwable` 類別為所有例外的父類別，且也可以使用例外處理機制丟出和捕捉此例外。`Throwable` 類別定義了存取錯誤訊息的方法以及輸出例外發生的 `stack trace` 來顯示例外發生的地方。`Throwable` 有三個主要的子類別：`Error`，`RuntimeException`，和 `Exception`，如圖 8-1 所示。

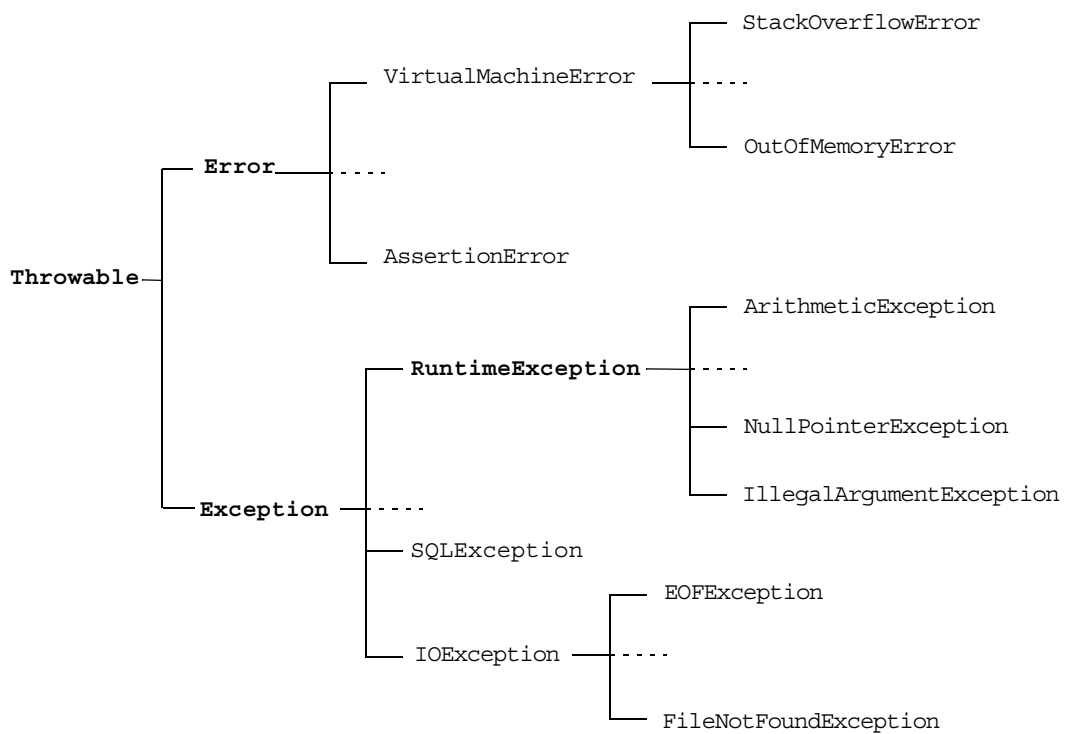


圖 8-1 例外的子類別

- 您不應該使用 `Throwable` 類別，您應該使用其子類別來描述特定的例外。下面介紹每個例外的目的：
- ● **Error** 表示問題嚴重到難以回復。例如記憶體不足。程式通常不會處理這類例外。
- ● **RuntimeException** 表示設計或實作上的問題。表示如果正常操作，該情況不應該發生。例如 `NullPointerException` 便不應該發生，假如程式員有將變數參考至新產生的物件。因為如果程式設計或實作正確，這類例外不會發生。通常也不會處理這類例外。這類例外會產生執行時期訊息，確保問題被解決，而不是被隱藏起來。

- 其他例外表示執行時期的例外通常由環境所造成，可以被處理。例如，無效 **URL** 例外，表示無法找到檔案（使用者打字錯誤），假如使用者打字錯誤，這些最容易常常發生。因為這些是由使用這所造成的錯誤，您應該要處理這類例外。

常見的例外

Java 提供幾種預先定義的例外，有些是相當常見的例外：

- `NullPointerException`
這是要存取物件的屬性或方法時，變數未參考任何物件所引起。例如，當變數未被初始或沒有產生實例。

```
Employee emp = null;  
System.out.println( emp.getName() );
```
- `FileNotFoundException`
這個例外是因為要讀取的檔案不存在。
- `NumberFormatException`
這個例外是因為要將字串轉換成數字（整數或浮點數）的格式錯誤造成。
- `ArithmeticException`
這個錯誤是因為除數為零。

```
int i = 0;  
int j = 12 / i;
```
- `SecurityException`
瀏覽器所丟出的典型例外。**SecurityManager** 類別丟出此例外，因為 **applets** 試著要存取伺服器上的檔案或是操作。下列為可能造成安全例外的操作範例：
 - 存取用戶端檔案系統。
 - 建立與提供 **applet** 的伺服器 **socket** 連結。
 - 於執行環境又執行其他程式。

處理或宣告例外的規則

為了鼓勵開發人員撰寫較穩健的程式，**Java** 要求：任何必須檢查的例外（**Exception** 的子類別但非 **RuntimeException** 子類別的例外）可能發生於任何地方，方法中必須明確定義如何處理這些例外的發生。

您可以做下列的事項以滿足上述要求：

- 使用 **try-catch-finally** 區塊處理例外

在方法中使用 **try-catch** 區塊，將可能產生例外的程式包起來。**catch** 子句必須指定預期可能產生的例外類別或是其父類別。即使 **catch** 子句內沒有任何程式碼，也算是有處理例外。

- 在方法宣告可能丟出的例外

這種做法主要用來告知呼叫者，被呼叫的方法不會處理該項例外，如果發生例外，會丟回由呼叫端處理。換句話說，就是宣告可能產生的例外，並將處理例外的責任交由呼叫端來進行。

下面這個方法宣告了一個例外，此例外可能於該方法的程式中使用 **throw** 子句丟出，如下所示：

```
void trouble() throws IOException {...}
```

緊接於 **throws** 關鍵字之後的一連串例外可以從方法中丟回給呼叫者。雖然這個例子只有一個例外，但是您可以使用逗號將一連串例外隔開。如下所示：

```
void trouble() throws IOException, OtherException  
{...}
```

RuntimeException 和 **Error** 子類別都是可不檢查的例外，不需要宣告於方法的 **throws** 子句。

您也可以選擇處理執行時期例外。通常大部分的執行時期例外表示程式邏輯有誤。最好是找出錯誤的地方，在上線前解決問題。因此也不需要於 **catch** 子句中處理執行時期例外。有些較重要的例外，例如 **NumberFormatException** 用於解析使用者輸入字串是否可用。



注意 -- 因為例外類別也像其他類別一樣具有階層，您可以使用相同的 **catch** 子句捕捉一群例外，如果是此例外類別的子類別即可。例如雖然有多種不同的 **IOExceptions** (**EOFException**, **FileNotFoundException** 等等)，只要捕捉 **IOException**，您就可以捕捉所有 **IOException** 子類別的例外。

方法覆寫和例外

如果被覆寫的方法有宣告它將丟出例外，則覆寫它的方法只能宣告同樣的例外或是該例外的子類別。例如，假如父類別方法宣告丟出 `IOException`，則覆寫的方法可以宣告丟出 `IOException`，或 `FileNotFoundException` (`IOException` 的子類別)，但不可以是 `Exception` (`IOException` 的父類別) 或是 `SQLException`。總之您可以宣告比原來少或比原來更「特定」（意指原宣告例外的子類別）的例外。下列範例有三個類別： `TestA`，`TestB1`，和 `TestB2`。 `TestA` 是 `TestB1` 和 `TestB2` 的父類別。

```

1  public class TestA {
2      public void methodA() throws IOException {
3          // 做某些事
4      }
5  }

1  public class TestB1 extends TestA {
2      public void methodA() throws EOFException {
3          // 做某些事
4      }
5  }

1  public class TestB2 extends TestA {
2      public void methodA() throws Exception {
3          // 做某些事
4      }
5  }
```

`TestB1` 類別可以通過編譯，因為 `EOFException` 是 `IOException` 的子類別。但是 `TestB2` 編譯會失敗，因為 `Exception` 是 `IOException` 的父類別。您可以在覆寫的方法中宣告多個例外或是不宣告任何例外。新的例外宣告會變成其子類別所能宣告例外的限制。

產生自訂例外

使用者自訂例外可用繼承 **Exception** 類別。例外類別和一般的類別一樣。下面為自訂例外的範例，此例外類別具有建構子，屬性和方法。

```
1 public class ServerTimeoutException extends Exception {
2     private int port;
3
4     public ServerTimeoutException(String message, int port) {
5         super(message);
6         this.port = port;
7     }
8
9     public int getPort() {
10         return port;
11     }
12 }
```



注意 -- 透過繼承自 **Exception** 類別的 **getMessage** 方法，可獲得例外發生的原因。

丟出自訂例外可使用下列的語法：

```
throw new ServerTimeoutException("Could not connect", 80);
```

上述程式碼建立了一個例外的實例，且和 **throw** 子句放在一起，因為例外訊息會包含發生例外的程式行數訊息。

丟出使用者自訂例外

考慮一個 **client-server** 程式。在用戶端的程式碼中，您試著連結伺服器且預期伺服器會在 5 秒內回覆。假如伺服器沒有回覆，您可以丟出下列例外（例如使用者自訂了 **ServerTimedOutException**）：

```
1  public void connectMe(String serverName)
2      throws ServerTimedOutException {
3      boolean successful;
4      int portToConnect = 80;
5
6      successful = open(serverName, portToConnect);
7
8      if ( ! successful ) {
9          throw new ServerTimedOutException("Could not connect",
10                                             portToConnect);
11      }
12 }
```

處理使用者自訂例外

使用 **try** 敘述句捕捉例外：

```
1 public void findServer() {  
2     try {  
3         connectMe(defaultServer);  
4     } catch (ServerTimeoutException e) {  
5         System.out.println("Server timed out, trying alternative");  
6         try {  
7             connectMe(alternativeServer);  
8         } catch (ServerTimeoutException e1) {  
9             System.out.println("Error: " + e1.getMessage() +  
10                " connecting to port " + e1.getPort());  
11         }  
12     }  
13 }
```



注意 -- 您可以使用巢狀 **try** 和 **catch** 區塊，如上述範例所示。

您可以處理部份的例外然後又丟出，例如：

```
try {  
    connectMe(defaultServer);  
} catch (ServerTimeoutException e) {  
    System.out.println("Error caught and rethrown");  
    throw e;  
}
```

Assertions

下面兩個為 **Assertion** 敘述句的語法：

```
assert <boolean_expression> ;
assert <boolean_expression> : <detail_expression> ;
```

上述兩種方式，只要結果是 **false** 就會丟出 **AssertionError**，千萬不要捕捉這個錯誤，您必須讓程式以非正常方式停止。假如用第二種語法，**detail_expression** 結果可以是任何型態，最後會被轉換成 **String** 以提供 **Assertion** 相關資訊。



注意 — 於 **JDK 1.3** 前的版本若程式使用到 **assert** 關鍵字宣告，會影響程式的正常執行。您可以使用 **-source 1.3** 設定告訴 **javac** 命令列編譯成 **JDK 1.3** 的版本。

建議的 Assertion 使用方式

Assertion 可看出開發人員的對於程式邏輯的假設與期望，可做為很有價值的文件。也因此 **Assertion** 對於共同開發與維護有相當的助益。.

Assertion 通常是用來檢驗內部單一方法的或是一小組相關方法的程式邏輯。其目的不是要丟出例外，而是檢驗程式碼正確性。**Assertion** 可廣泛用來確保內部的不變性，流程控制的不變性、後置條件 (**postcondition**) 和類別不變性等。

內部不變性

內部不變性指的是當您相信情況總是這樣或不是這樣，例如下面的程式：

```
1  if (x > 0) {
2      // 作這個
3  } else {
4      // 做那個
5  }
```

假如您已經假設 **x** 值為 0，決不會是負數，一旦 **x** 變成負數，程式就會產生非預期的錯誤行為。更糟糕的是，程式沒有因而停止並提出警訊，您將無法找出問題點，直到造成損害才會察覺。這種問題很難察覺。這種狀況下，**Assertion** 的使用可以幫助確認邏輯的正確性，找出臭蟲或是人為修改程式造成邏輯錯誤。**Assertion** 的使用範例如下：

```
1  if (x > 0) {
2      // 作這個
3  } else {
```

使用例外 (**Exception**) 類別和 **Assertions** 進行錯誤處理

```
4      assert (x == 0);
5      // 做那個，除非 x 是負數
6  }
```

流程控制的不變性

流程控制不變性與內部不變性類似，但是它與執行流程較相關。例如，在 **switch** 敘述句中，您可能相信會遇到各種可能的控制值，因此 **default** 敘述句決不會被執行到。這種情況下就可以加入 **Assertion**，範例如下：

```
1  char myGrade = employee.getGrade();
2  switch (myGrade) {
3  case 'A': // ...
4      break;
5  case 'B': // ...
6      break;
7  case 'C': // ...
8      break;
9      default: assert false : "impossible grade: " + myGrade;
10 }
```

後置條件和類別不變性

後置條件 (**Postcondition**) 指的是方法執行完成後對於變數值或其關係的假設。舉例來說，在執行 **Stack** 類別的 **pop** 方法之後，堆疊中應該要少一個元素，除非堆疊本來就是空的。程式範例如下：

```
1  public Object pop() {
2      int size = this.getElementCount();
3      if (size == 0) {
4          throw new IllegalStateException
5              ("Attempt to pop from empty stack");
6      }
7
8      Object result = /* 取出一個元素的程式碼 */ ;
9
10     // 測試後置條件
11     assert (this.getElementCount() == size - 1);
12
13     return result;
14 }
```

假如 `pop` 方法呼叫並沒有丟出例外，且堆疊是空的，那就很難表示 **Assertion**，因為原本的大小就是 0，最後結果也是 0。此時可使用前置條件 (precondition) 來測試，也就是確保呼叫前若堆疊是空的話，就不做診斷。這是因為方法本身邏輯並沒有錯，所以遇到這種狀況並不視為錯誤。我們通常不使用 **Assertion** (反而是用簡單的例外) 來測試外部行為。使用 **Assertion** 可以確保測試被執行，且不被解除。

類別不變性是用於所有類別的方法呼叫結束後的測試。例如，堆疊類別不變的條件是，元件的數量不可能是負數。

不當使用 Assertion

不要使用診斷檢查公開方法的參數。應該要測試每個參數並丟出適當的例外，如 `IllegalArgumentException`。不使用 **Assertion** 檢查參數的原因是因為 **Assertion** 功能可以被關閉，但是您仍必須對參數做檢查。

不要 **Assertion** 敘述中呼叫方法，可能會有副作用，因為 **Assertion** 可能於執行時期關閉。假如程式依賴這些副作用，在關閉診斷後程式就會有不同的行為出現。

控制執行時期 Assertion 的執行

診斷的其中一個重要優點是，它可以在執行時期解除檢查功能。假如可以解除，就不會有過度使用的問題，程式的執行速度與未加任何 **Assertion** 時一樣快。這種方法比條件式編譯好的原因有兩個。第一，產品不需要不同階段的編譯。第二，**Assertion** 可以再啟動，檢查因為環境因素所造成的影響。

Assertion 的預設值為解除狀態，可以使用下列其中一種命令式來開啟：

```
java -enableassertions MyProgram  
java -ea MyProgram
```

Assertion 可以設定開啟範圍為套件，套件階層，或是個別類別，詳細資訊請參考文件：

<http://java.sun.com/javase/6/docs/technotes/guides/language/assert.html> 或是安裝的 JDK。

宣告和使用特殊的類別型式

單元重點

當完成這個單元後，您將能夠：

- 宣告和使用抽象方法和類別
- 宣告和使用介面
- 宣告和使用巢狀類別
- 使用列舉類別

其他資源



其他資源 — 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O’ Reilly Media. 2005.

抽象方法 (Abstract Methods) 和抽象類別 (Abstract Classes)

本章節將介紹抽象方法和抽象類別的概念。請參閱圖 9-1 的繼承層級。

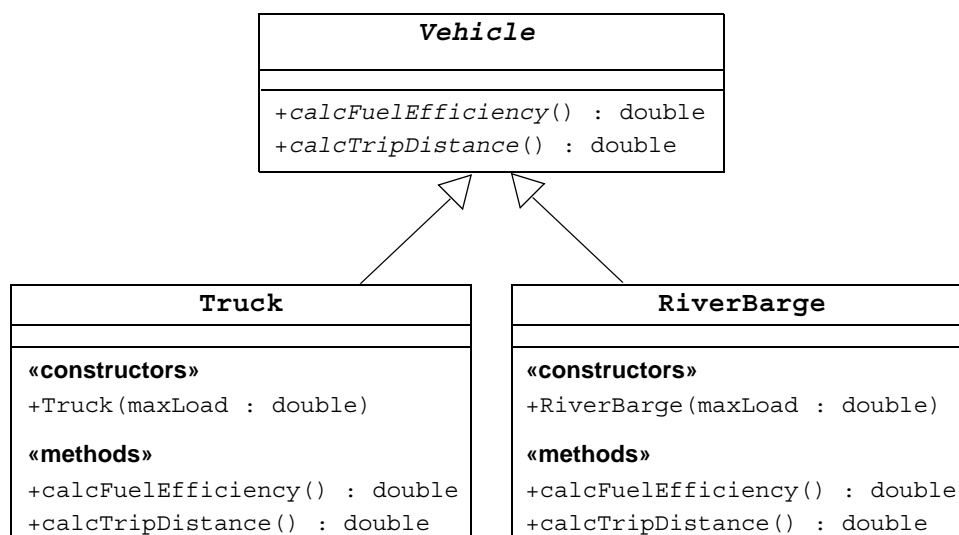


圖 9-1 Vehicle 繼承層級

圖 9-1 展示 **Vehicle** 類別以及兩個子類別的 UML 圖形。**Vehicle** 類別於此層級中代表通用類別，其子類別具有共同的元素。**Vehicle** 類別有下列兩個方法：

- `calcFuelEfficiency`
- `calcTripDistance`

這個案例的問題在於，兩個子類別對於這兩個方法的實作完全不同。結果您無法於 **Vehicle** 類別提供一個共通有意義的實作（具體的方法）。其中一種方法是像下列程式一樣，為 **Vehicle** 類別中最少的實作程式碼，而子類別可利用覆寫機制來實作這兩個方法。

```
public double calcFuelEfficiency() { return -1; }
public double calcTripDistance() { return -1; }
```

但是，這樣並不安全，因為可能會有子類別沒覆寫方法。

要避免這種狀況的發生，**Java** 技術規範允許宣告抽象方法和對應的抽象類別。

- 抽象方法 (abstract methods)

抽象方法 (反之為具體方法 (concrete methods)) 是一個沒有實作內容的方法介面宣告。例如：

```
public abstract double calcFuelEfficiency();  
public abstract double calcTripDistance();
```

抽象方法使用 **abstract** 關鍵字。抽象方法宣告以分號 ‘;’ 做結束。包含抽象方法的抽象 (父) 類別可以選擇是否提供具體方法。抽象父類別的具體子類別必須實作父類別的抽象方法。

- 抽象類別 (abstract classes)

抽象類別 (反之為具體類別 (concrete classes)) 是一個宣告為 **abstract** 的類別。它可以包含零到多個抽象方法。例如：

```
public abstract class Vehicle {  
    protected String registrationAuthority;  
    public abstract double calcFuelEfficiency();  
    public abstract double calcTripDistance();  
    public getAuthority() {  
        return registrationAuthority;  
    }  
}
```

下面規則適用於抽象類別：

- 1 抽象類別的宣告必須使用 **abstract** 關鍵字。
- 1 抽象類別可以有抽象方法，但也可以沒有抽象方法。
- 1 抽象類別可以有具體方法。
- 1 抽象類別可以宣告類別屬性。

一個抽象類別的範例

Java 程式語言允許開發人員定義父類別但不必提供方法實作。這些方法稱為抽象方法 (**abstract method**)。方法的實作由子類別提供。任何具有抽象方法的類別即稱之為抽象類別 (**abstract class**) (請參閱圖 9-2)。

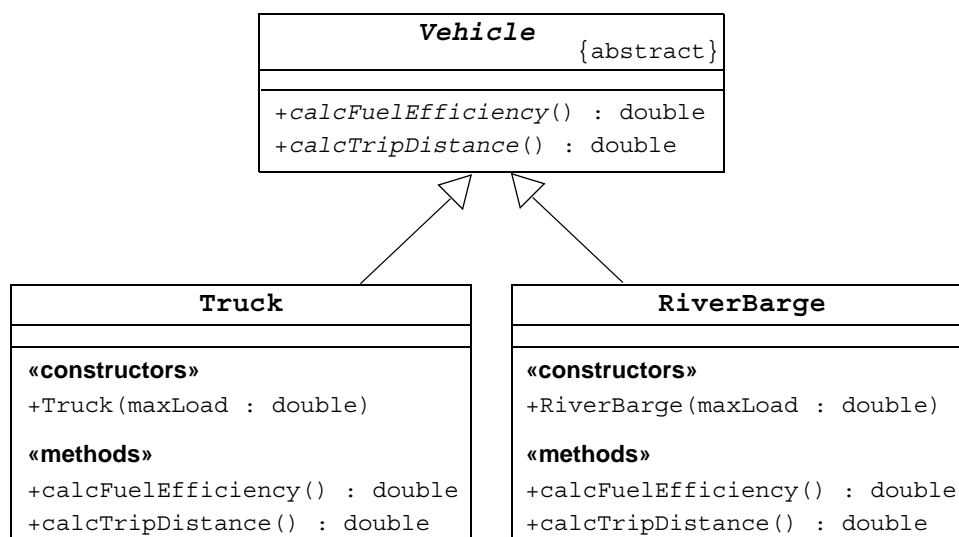


圖 9-2 抽象的 **Vehicle** 類別

圖 9-2 為上述解決方案的 UML 模型。Vehicle 為抽象類別，有兩公開的抽象方法。



注意 --UML 於類別圖中使用斜體字表示抽象元素。您也可以使用 `{abstract}` 標記來表示抽象類別。

Java 編譯器會防止您初始化抽象類別。例如使用 `new Vehicle()` 是不合法的。程式 9-1 是 **Vehicle** 的類別宣告。

程式 9-1 是 **Vehicle** 的類別宣告。

程式 9-1 抽象的 **Vehicle** 類別

```

1 public abstract class Vehicle {
2     public abstract double calcFuelEfficiency();
3     public abstract double calcTripDistance();
4 }
    
```

程式 9-2 展示具體的 **Truck** 類別

程式 9-2 Concrete Truck Class

```
1 public class Truck extends Vehicle {
2     public Truck(double maxLoad) {...}
3     public double calcFuelEfficiency() {
4         // 計算 truck 耗油量
5     }
6
7     public double calcTripDistance() {
8         // 計算在高速公路上行駛的里程數
9     }
10 }
```

程式 9-3 展示具體的 **RiverBarge** 類別

程式 9-3 Concrete RiverBarge Class

```
1 public class RiverBarge extends Vehicle {
2     public RiverBarge(double maxLoad) {...}
3     public double calcFuelEfficiency() {
4         // 計算耗油量
5     }
6
7     public double calcTripDistance() {
8         // 計算在河道中行駛的里程數
9     }
10 }
```

然而抽象類別可以有資料欄位，具體方法，和建構子。例如，**Vehicle** 類別可以有 **load** 和 **maxLoad** 欄位和一個建構子來將欄位初始化。將建構子宣告為 **protected** 而非 **public** 是個好做法，因為這些建構子是只給子類別呼叫用。將建構子宣告為 **protected** 對於開發人員來說也比較容易了解類別設計者的意圖，表示不可以直接呼叫抽象類別建構子。

介面 (Interfaces)

下列敘述介面的特性：

介面所宣告的方法介面沒有實作內容。

介面的方法由類別來實作。

多個類別可以實作同一個介面。

一個類別可以實作多個介面。

介面（和類別類似）定義一個 **Java** 型別。

您可以使用介面名稱來指稱型別，也可以在介面參考和它所指向的物件間相互轉型。透過 **instanceof** 運算子則可以決定所指稱的物件是否為實作介面的物件。

Flyer 範例

想像一群物件都有同樣的能力：飛。您可以建立一公開的介面，稱之為 **Flyer**，有三個操作：**takeOff**，**land**，和 **fly**。（請參閱圖 9-3）

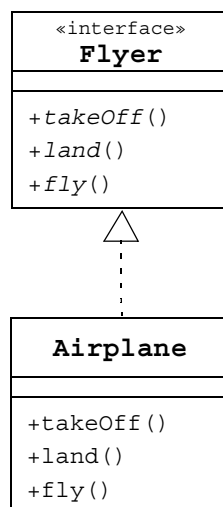


圖 9-3 Flyer 介面和 Airplane 實作

程式 9-4 展示 **Flyer** 介面的宣告。

程式 9-4 Flyer 介面

```

1 public interface Flyer {
2     public void takeOff();
3     public void land();
  
```

```

4     public void fly();
5 }

```

Airplane 類別實作了 **Flyer** 介面

程式 9-5 **Flyer** 介面的實作

```

1  public class Airplane implements Flyer {
2      public void takeOff() {
3          // 加速直到上升
4          // 收起機輪
5      }
6      public void land() {
7          // 放下機輪
8          // 降低轉速直到接觸地面
9          // 使用煞車
10     }
11     public void fly() {
12         // 引擎繼續運轉
13     }
14 }

```

可能會有多个類別實作 **Flyer** 介面如圖 9-4 所示。飛機 (**Airplane**) 可以飛，鳥 (**Bird**) 可以飛，超人 (**Superman**) 也可以飛，諸如此類。

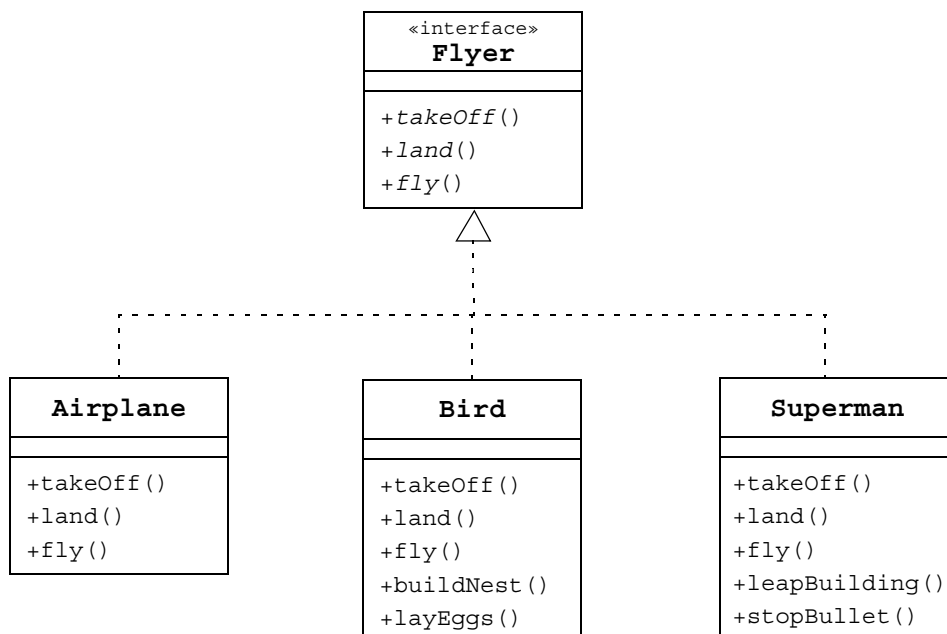


圖 9-4 **Flyer** 介面有多个實作

飛機是一種會飛的交通工具，小鳥是一種會飛的動物。下面範例展示繼承自不同類別的子類別，也可以實作相同介面。

這看起來像是多重繼承，但又不完全是。多重繼承危險的地方在於當繼承自兩個不同的類別且這兩個類別有相同的方法時，容易發生混淆。但使用介面就不會發生這種狀況，因為介面只宣告方法而沒有實作。如圖 9-5 所示，單一類別可以繼承多個介面。

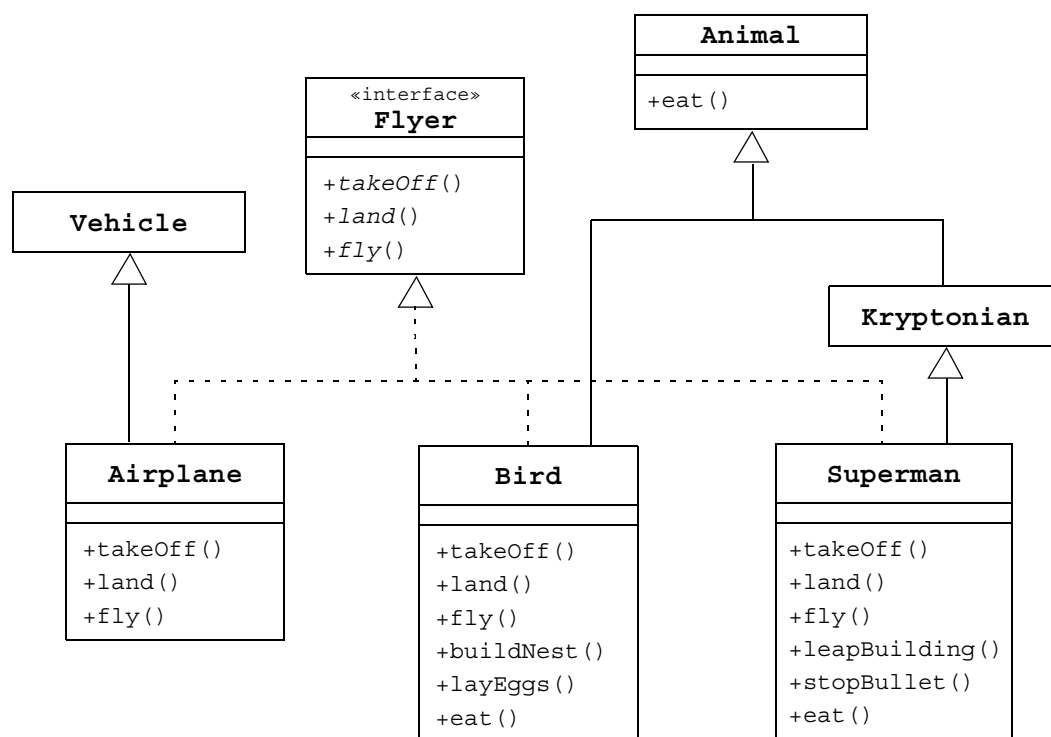


圖 9-5 混合類別繼承和介面實作

程式 9-6 是 **Bird** 類別的內容：

程式 9-6 **Bird** 類別宣告

```

public class Bird extends Animal implements Flyer {
    public void takeOff() { /* take-off 實作 */ }
    public void land() { /* landing 實作 */ }
    public void fly() { /* fly 實作 */ }
    public void buildNest() { /* 築巢行為 */ }
    public void layEggs() { /* 孵蛋行為 */ }
    public void eat() { /* 覆寫吃的行為 */ }
}

```

extends 子句必須在 **implements** 子句之前。 **Bird** 類別可提供自有的方法 (**buildNest** 和 **layEggs**)，也可以覆寫 **Animal** 類別的方法 (**eat**)。

假設您在建立飛機的控制系統軟體，它必須有賦予所有物體下降和起飛的權利。如圖 9-6 所示

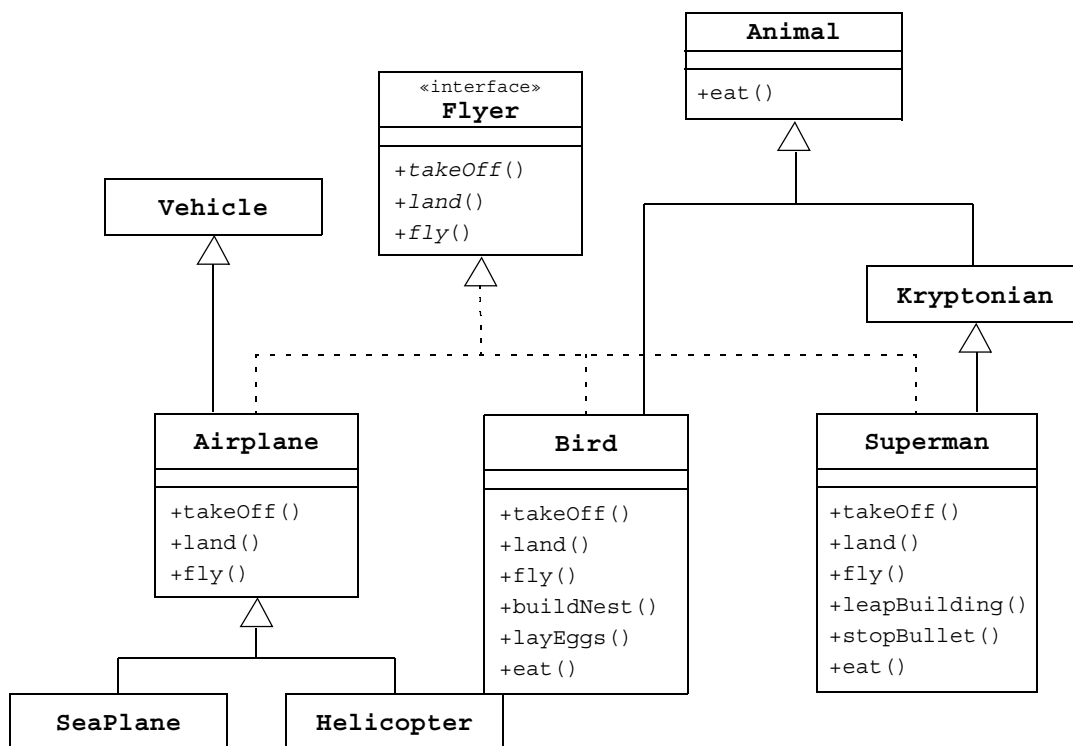


圖 9-6 Airport 範例的類別層級。

程式 9-7 展示 **Airport** 類別的宣告範例：

程式 9-7 **Airport** 類別宣告

```
1  public class Airport {
2      public static void main(String[] args) {
3          Airport metropolisAirport = new Airport();
4          Helicopter copter = new Helicopter();
5          SeaPlane sPlane = new SeaPlane();
6
7          metropolisAirport.givePermissionToLand(copter);
8          metropolisAirport.givePermissionToLand(sPlane);
9      }
10
11     private void givePermissionToLand(Flyer f) {
12         f.land();
13     }
14 }
```

多重介面的範例

類別可以實作單一或多個介面。**SeaPlane** 不只可以飛，也可以航行。**SeaPlane** 類別繼承 **Airplane** 類別，所以它也繼承了 **Flyer** 介面的實作。**SeaPlane** 類別也實作了 **Sailer** 介面。如圖 9-7 所示。

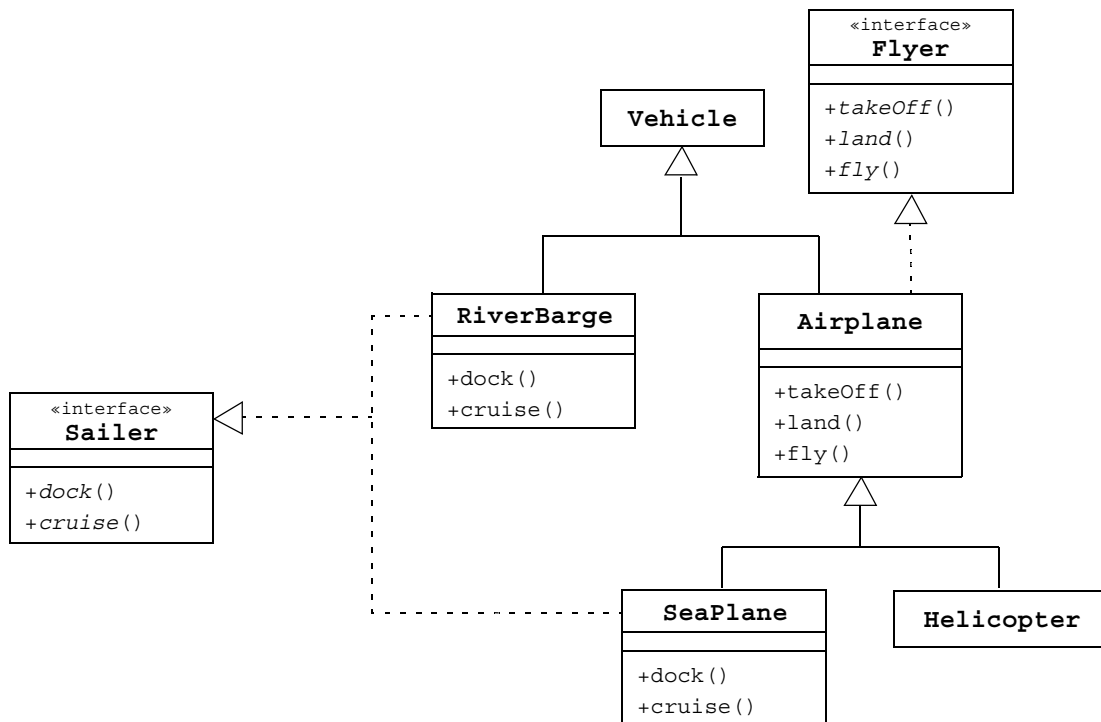


圖 9-7 多重實作範例

現在撰寫 **Harbor** 類別，具有賦予停泊權的權利：

```

1  public class Harbor {
2      public static void main(String[] args) {
3          Harbor bostonHarbor = new Harbor();
4          RiverBarge barge = new RiverBarge();
5          SeaPlane sPlane = new SeaPlane();
6
7          bostonHarbor.givePermissionToDock(barge);
8          bostonHarbor.givePermissionToDock(sPlane);
9      }
10     private void givePermissionToDock(Sailer s) {
11         s.dock();
12     }
13 }

```

Seaplane 可以由 **Metropolis** 機場起飛然後停泊於 **Boston** 港。

介面宣告和使用規則

類別的公開介面 (**public interface**) 是類別為提供服務而與客戶端程式的一種合約。

- 具體類別需要實作介面的每一個方法
- 抽象類別可以宣告抽象方法來延遲方法實作
- **Java** 介面只宣告合約沒有實作

介面不可以被初始化：他們只可以被類別實作（或是被其它介面繼承）。具體類別必須實作所有宣告將實作的介面中的所有方法。多個類別可以實作同一個介面。這些類別不需要在同一個類別階層中。此外，一個類別也可以實作多個介面，這種能力使的 **Java** 技術在多重繼承 (**multiple inheritance**) 上比其他語言來要強大，亦不會增加複雜度。



注意 -- 所有宣告在界面上的方法都是 **public** 和 **abstract**。甚至這些修飾詞不需明確於程式中指定。同樣的，所有的屬性都是 **public**, **static**, 和 **final**；換言之，您只可以宣告常數屬性。

介面只可以包含下列部分：

- 常數（介面中宣告的欄位都是 **public**, **static**, 和 **final**。）
- 方法介面（但無方法實作）

巢狀類別 (Nested Classes)

Java 技術允許您在類別中宣告其它類別。這些類別稱之為巢狀類別(nested classes)。程式 9-8 是一個巢狀類別的範例。

程式 9-8 Inner Class Example

```

1  public class StackOfInts {
2      private int[] stack;
3      private int next = 0; // index of last item in stack + 1
4
5      public StackOfInts(int size) { //... }
6      public void push(int on) { // ... }
7      public int pop() { // ... }
8
9      private class StepThrough { // ... }
10         private int i;
11         public increment() { // ... }
12         public int current() { // ... }
13         public boolean isLast() { // ... }
14     }
15
16     public StepThrough stepThrough() { // ... }
17     public static void main(String[] args) { // ... }
18 }

```

巢狀類別可以存取外部類別的所有欄位和方法。巢狀類別通常是用來輔助外部類別，也就是說，他與外部類別的關係非常密切，高度依賴於外部類別。巢狀類別有下列優點：

- 另一層的封裝 - 封裝指的是將相關的資料和方法封裝於同一個類別。於外部類別宣告巢狀類別也可支援封裝，但兩個類別的耦合度非常高。
- 提高程式的可讀性和維護性 - 於要使用的類別宣告巢狀類別可以讓兩個類別間的關係較清楚。而且因為巢狀類別和外部類別具有同樣的命名空間，程式可以更簡潔。
- 另一層的類別階層組織 - 巢狀類別提供了比套件更精細結構來組織輔助類別 (Helper classes) 的方法。佈署和發佈相關或依賴的類別非常簡單，因為這些類別可以與外部類別結合在一起。



注意 -- 巢狀類別是在編譯時期產生檔案。編譯的時候，Java 會將巢狀類別自動產生檔名例如 `StackOfInts$StepThrough.class`。同時也會在外部類別產生額外的程式碼（例如，特別的 `getter` 和 `setter` 方法）讓內部類別可以存取其 `private` 成員。這些額外的程式碼會以位元碼方式存在您個程式中。執行時期 JVM 對於巢狀類別並不會有任何特別處理。

了解巢狀類別語法

程式 9-9 展示巢狀類別的語法。

程式 9-9 巢狀類別的語法

```

1  [public] class OuterClass {
2      ...
3      [public | protected | private | static]class NestedClass {
4          ...
5      }
6  }
```

巢狀類別可以宣告為 `private`, `public`, `protected`, 或是 `package private` 就像一般的類別一樣。最外一層的類別只可以宣告為 `public` 或 `default` (`package private`)。和最外一層的類別不一樣的是，巢狀類別可以宣告為 `static`。巢狀類別可以分為兩類：

- `Non-static` 巢狀類別，稱之為內部類別 (inner classes)。
- `static` 巢狀類別

內部類別 (Inner Classes)

內部類別和實例變數一樣，在執行時期與外部的類別物件實例緊密結合。內部類別可以直接存取外部類別物件的方法和欄位，以及其所繼承的方法和欄位。但是因為內部類別只能存在於外部類別實例中，因此不能具有 **static** 成員。

程式 9-10 簡單版的內部類別宣告語法：

程式 9-10 內部類別語法

```
class OuterClass {  
    ...  
    class InnerClass {  
        ...  
    }  
    ...  
}
```

內部類別物件必須存在於外部類別實例中。這些物件可以直接存取外部類別實例的方法和欄位：要在外部類別以外的地方初始內部類別，您必須先初始外部類別，然後再使用外部類別物件產生內部類別：

```
OuterClass.InnerClass innerObject = outerObject.new  
InnerClass();
```

此外，有兩種特別的內部類別：區域類別 (**local classes**) 和匿名內部類別 (**anonymous inner classes**)。

區域內部類別

區域內部類別（有時稱之為區域類別 (**local class**)）宣告於方法程式區塊中。就像區域資料欄位一樣，區域內部類別的有效範圍只在於宣告的區塊中。但是它可以存取方法中宣告為 **final** 的區域變數或參數。程式 9-11 為區域內部類別範例。

程式 9-11 區域內部類別

```
class OuterClass {  
    ...  
    ...  
    void aMethod() {  
        ...  
        ...  
        class InnerClass {  
            ...  
        }  
        ...  
        InnerClass x = new InnerClass();  
        // 使用實例變數 x  
        ...  
    }  
    ...  
    ...  
}
```

匿名內部類別

匿名內部類別（也稱之為匿名類別 (**anonymous class**)）為宣告時不具名的類別。但是因為沒有類別名稱，所以宣告時語法有些不同：

```
new existingTypeName ( [ argumentList ] ) {  
    // 程式碼  
}
```

匿名內部類別並未使用一般的 **extends** 語法宣告匿名類別，因此使得程式看起來就像是在呼叫父類別的建構子一樣。例如，假設父類別是 **Object**，參數列必須是空的，因為 **Object** 的建構子沒有參數。

程式 9-12 是一個匿名類別的範例，它繼承了程式 9-5 中的 **Airplane** 類別。

程式 9-12 匿名類別範例

```
1  metropolisAirport.givePermissionToLand(new AirPlane() {  
2      public void land() {  
3          // 做些只有 UFOs 會做的事...  
4      } /land 方法的結束  
5  } // 匿名類別的結束  
6  ) // givePermissionToLand 方法的結束  
7  ; // 第一行語句的結束
```



注意 -- 您可能也會在第一行撰寫 **new Flier()**。這表示匿名類別實作 **Flier** 介面。這個情況下您必須實作 **Flier** 介面所有的方法。

匿名類別在下列的情況下非常有用：

- 宣告類別時隨即使用該類別
- 該類別只包含了很少量的程式碼

匿名類別特性：

- 不用指定，匿名類別就是 **final**
- 不可以是 **abstract** 和 **static**

Static 巢狀類別

Static 巢狀類別指的是於最外部的類別所在命名空間下宣告的 **static** 類別（或是命名空間內的另一個 **static** 巢狀類別）。**Static** 巢狀類別不可以直接使用外部類別的變數或是方法；使用限制規則如同類別的 **static** 方法。巢狀類別只能存取外部類別物件實例的變數或是方法。程式 9-13 展示 **static** 巢狀類別的簡單範例。

程式 9-13 **Static** 巢狀類別

```
public class Outer {
    ...
    public static class StaticNested {
        ...
    }
    ...
}
```

您可以透過外部類別物件參考存取 **static** 巢狀類別：

```
Outer.StaticNested
```

例如，要產生 **static** 巢狀類別，其語法如下：

```
Outer.StaticNested nestedObject = new Outer.StaticNested();
```

內部介面，列舉型別和標註 (Annotations)

介面，列舉型別和標註型別也可以內部方式宣告。因為在 **Java** 中這些都是 **static**，它們的使用限制和 **static** 巢狀類別相似。您可以使用介面產生內部匿名類別：

```
new interfaceName () { interfaceSpecification }
```

列舉型別

在 Java 5.0 引入一新的關鍵字 **enum** 來提供一組數量有限且型別安全的列舉值。Enum 型別的內容為一系列以逗號區隔的列舉常數 (**enum constants**)。列舉常數和 **static final** 欄位類似，但編譯器強迫列舉常數的參考指向單一的物件型別。下面為列舉型別宣告的範例：

```
public enum TrafficSignal {  
    STOP, CAUTION, GO  
}
```

您可以想像 **TrafficSignal** 型別為一個具有一組數量有限，且沒有重複值的類別。每個值都有一個象徵的名稱列於型別定義中。例如，**TrafficSignal.STOP** 是 **TrafficSignal** 的一種。就像 Java 中的其它常數的命名慣一樣，列舉常數的名稱通常使用大寫。以上面的例子來說，**TrafficSignal** 型別的變數，只能指定下列的值，否則會有錯誤：

- `TrafficSignal.STOP`
- `TrafficSignal.CAUTION`
- `TrafficSignal.GO`
- 或 `null`.

Switch 敘述句

Switch 敘述句可以使用列舉型別；但是有些許限制。例如，**case** 標籤不可以包含列舉類別的名稱，在下面的範例中，請注意 **case** 標籤並未使用 **TrafficSignal**。

```
public class Test {  
    public enum TrafficSignal {STOP, CAUTION, GO};  
  
    public static void main(String [] args) {  
        TrafficSignal theLight= TrafficSignal.GO;  
        switch (theLight) {  
            case STOP:  
                System.out.print("red");  
                break;  
            case CAUTION:  
                System.out.print("yellow");  
                break;  
            case GO:  
                System.out.print("green");  
                break;  
        }  
    }  
}
```

```
}
}
```



注意 -- 回傳結果: **green**。

For 迴圈

列舉型別也可用於加強版的 **for** 迴圈。下面程式使用 **for** 迴圈逐一列舉 **TrafficSignal** 類別的常數值，如下所示：

```
public class Test{
    public enum TrafficSignal {STOP, CAUTION, GO};

    public static void main(String[] args) {
        for( TrafficSignal t : TrafficSignal.values() ) {
            System.out.println(t);
        }
    }
}
```



注意 -- **values** 是個 **static** 方法，它回傳一個包含所有 **TrafficSignal** 列舉類別的陣列。



注意 -- 回傳結果：

```
STOP
CAUTION
GO
```

具有欄位，方法和建構子的列舉型別

列舉型別也可以有資料欄位和方法，就像宣告類別一樣。這些內容視為匿名內部類別宣告；外部類別則為 **enum** 宣告的類別。

enum 宣告中的建構子不是一般的建構子，因為初始化 **enum** 是被禁止的（宣告 **public enum** 建構子會造成編譯錯誤）。每一個 **enum** 常數都可以有參數列。當 **enum** 被初始化時，每個 **enum** 的參數列會對應到相對應的建構子。可以用來將建構子多載 (**overload**)，以接收不同的參數。

下面的程式宣告列舉型別 **TrafficSignal**，具有三種可能的值（例如，**STOP**），每個值對應不同字串代表不同顏色（例如，**red**）。建構子會自動將三種指定的顏色值 (**t**) 指定給 **enum** 欄位 (**light**)。

```
public class Test {
    public enum TrafficSignal {
        STOP("red"), CAUTION("yellow"), GO("green");

        private final String light;

        private TrafficSignal(String t) { light = t; }

        public String format(String message) {
            return message + " " + light;
        }
    }

    public static void main(String[] args) {
        System.out.println(TrafficSignal.GO.format("The light is"));
        System.out.println(TrafficSignal.STOP.format("The light is"));
    }
}
```



注意 -- 回傳結果：

```
The light is green
The light is red
```

使用泛型 (Generics) 和集合框架 (Collections Framework)

單元重點

當完成這個單元後，您將能夠：

- 描述什麼是集合 (Collections)
- 描述實作集合框架的核心介面目的及概要
- 了解 **Map** 介面
- 了解舊式集合類別
- 實作 **Comparable** 和 **Comparator** 介面產生原始和自訂排序
- 使用泛型集合
- 了解泛型類別參數的使用
- 重構 (refactor) 既存的非泛型程式
- 撰寫程式來依序存取集合元素
- 了解加強型 **for** 迴圈

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.

集合 API

集合 (**collection**) 是用來管理一群物件的物件。集合內的物件稱為集合元素 (**elements**)。基本上，集合可以處理多種型別的物件，這些型別的物件都屬於一個特定的型別（也就是說他們具有共同的父類別）。

集合 API 包含的介面如下：

- **Collection** - 一群稱為元素的物件；實作上可決定是否有特定的順序或是可有重複元件。
- **Set** - 無順序的集合；不允許元件重複
- **List** - 有順序的集合；可允許元件重複

在 **Java SE 5.0** 前的版本，集合維護了一群 **Object** 類別的物件，使得任何物件都可以放在集合內，但在使用集合中的物件之前必須先正確的強制轉型。到了 **Java SE 5.0**，您可以使用泛型集合的機制，指定存放於集合的型別，取用時無須做強制轉換。泛型集合會在「泛型」一節中有較詳細的討論。

圖 10-1 是集合 API 中基本介面及實作類別的 UML 圖。

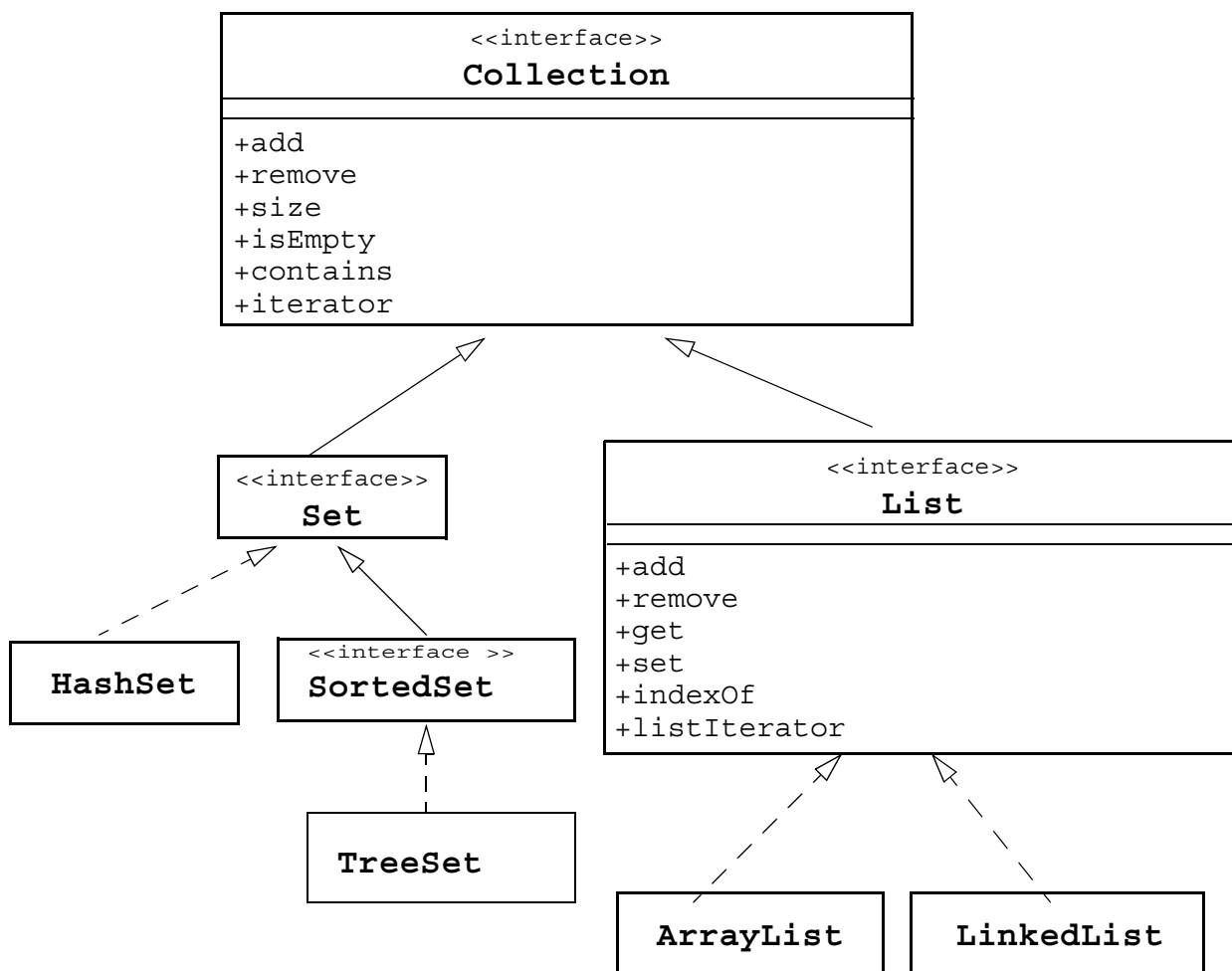


圖 10-1 集合介面的類別層級

HashSet 為提供實作 Set 介面的類別之一。SortedSet 介面繼承自 Set 介面。實作 SortedSet 介面的類別會對其元件排序。TreeSet 實作 SortedSet 介面。ArrayList 和 LinkedList 類別則實作了 List 介面。



注意 -- 上述的說明只是簡化後的集合 API (完整的集合 API 包含更多方法, 介面和中介抽象類別)。詳細資訊請參考: [Introduction to the Collections Framework URL](http://developer.java.sun.com/developer/onlineTraining/collections/) 連結為:

<http://developer.java.sun.com/developer/onlineTraining/collections/>

集合的實作

在集合框架中有幾個通用的核心介面 (**Set**, **List**, **Map** 和 **Deque**)。表 10-1 展示了幾個具體的實作類別。

表 10-1 通用的集合實作

	雜湊表 (Hash Table)	可變長度陣 列	平衡樹 (Balanced Tree)	鏈結串列 (Linked List)	雜湊表 + 鏈結串 列
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

下面幾節將介紹 **HashSet** 和 **ArrayList** 的使用範例。

一個 **Set** 的範例

程式 10-1 的範例中，程式宣告了一個 **Set** 型別的變數 (**set**)，但其所指派的初始物件為 **HashSet**。在 **set** 中加入幾個元素後，輸出 **set** 到標準輸出裝置。程式 10-1 的第 10 和第 11 行試著加入重複的值到 **set**。但因為重複的值無法加入 **set**，所以 **add** 方法回傳 **false**。

程式 10-1 **SetExample** 程式

```

1  import java.util.*;
2  public class SetExample {
3      public static void main(String[] args) {
4          Set set = new HashSet();
5          set.add("one");
6          set.add("second");
7          set.add("3rd");
8          set.add(new Integer(4));
9          set.add(new Float(5.0F));
10         set.add("second");           // 重複，不可加
11         set.add(new Integer(4));    // 重複，不可加
12         System.out.println(set);
13     }
14 }
15
16
```

上述程式輸出結果為：

```
[one, second, 5.0, 3rd, 4]
```

要注意的是元件輸出的順序與元件加入的順序不一樣。



注意 -- 在第 13 行，程式輸出 **set** 物件到標準輸出裝置。這樣也可以輸出的原因是 **HashSet** 覆寫了 **toString** 方法，然後產生以逗號分隔的元素字串表示格式，並且在前後加上括弧。

一個 List 的範例

程式 10-2 的範例中，程式宣告了一個 **List** 型別的變數 (**list**)，並指派一個新的 **ArrayList** 物件，然後在 **list** 中加入元素，最後將 **list** 輸出到標準輸出裝置。因為 **list** 允許重複元件，程式 10-2 第 10 和 11 行的 **add** 方法回傳 **true**。

程式 10-2 ListExample Program

```
1  import java.util.*;
2  public class ListExample {
3      public static void main(String[] args) {
4          List list = new ArrayList();
5          list.add("one");
6          list.add("second");
7          list.add("3rd");
8          list.add(new Integer(4));
9          list.add(new Float(5.0F));
10         list.add("second");           // 重複，可加
11         list.add(new Integer(4));     // 重複，可加
12         System.out.println(list);
13     }
14 }
```

上述程式的輸出結果為：

```
[one, second, 3rd, 4, 5.0, second, 4]
```

元素的順序與當初加入的順序一樣。



注意 -- 集合 API 包含很多實作 **Collection**, **Set**, 和 **List** 介面的類別。您可以細讀這些 API 文件熟悉這些類別。有些類別甚至實作綜合行為，例如 **LinkedHashMap**，使用雜湊函數實作快速搜尋，且內部使用雙向鏈結串列元件，所以可以依序存取集合中的元件。

Map 介面

Map 有時稱之為關連式 (associative) 陣列。Map 物件記錄了 key 與 value 的對應關係。就定義來說 Map 物件不允許重複的 key 或是 null 的 key 值，每個 key 最多只能對應一個 value。

Map 介面提供三個方法讓 `map` 內容可以被視為集合：

- **entrySet** - 回傳一個 **Set**，包含成對的 **key** 與 **value**。
- **keySet** - 回傳一個 **Set**，包含該 **map** 所有的 **keys**。
- **values** - 回傳一個 **Collection**，包含該 **map** 所有的 **values**。

圖 10-2 展示 Map 介面，子介面和幾個較常用的實作類別。

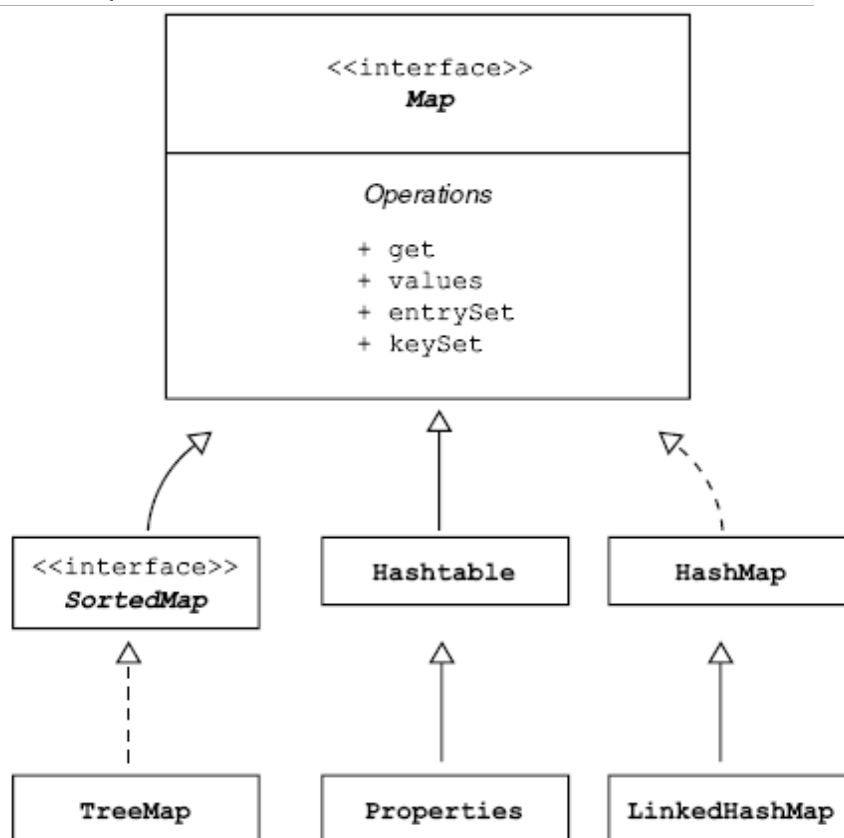


圖 10-2 Map 介面 API

Map 介面並沒有繼承 Collection 介面，因為它使用的是對映機制，和一般集合物件不太一樣。SortedMap 介面繼承 Map 介面。有些類別實作 Map 介面，如 HashMap, TreeMap, IdentityHashMap, 和 WeakHashMap。上述的 Map 集合中，Iterator 會列出所有元素時，其順序會因為實作類別而有所不同。

一個 Map 的範例

程式 10-3 中宣告了一個 **Map** 型別的變數 **map**，並指定一個新的 **HashMap** 物件。然後使用 **put** 方法加入幾個元件。為了證明 **map** 不允許有重複鍵值，程式試著加入已存在的 **key** 及其對應的 **value**。結果是該 **key** 所對應的 **value** 已被新加入的 **value** 所取代。接著程式也使用集合取出 **keySet**, **values**, 和 **entrySet** 集合中的內容。

程式 10-3 MapExample 程式

```

1  import java.util.*;
2  public class MapExample {
3      public static void main(String args[]) {
4          Map map = new HashMap();
5          map.put("one", "1st");
6          map.put("second", new Integer(2));
7          map.put("third", "3rd");
8          // 覆寫前面的指定
9          map.put("third", "III");
10         // 回傳 key 集合
11         Set set1 = map.keySet();
12         // 回傳 value 的集合
13         Collection collection = map.values();
14         // 回傳成對 key 與 value 的集合
15         Set set2 = map.entrySet();
16         System.out.println(set1 + "\n" + collection + "\n" + set2);
17     }
18 }
```

程式輸出結果為：

```

[one, third, second]
[1st, III, 2]
[one=1st, three=III, second=2]
```

舊式集合類別

JDK 1.0和1.1版本的集合類別仍存在於目前的版本中且具有相同的介面，但已經被重新歸類，整合至新的集合 API。

- **Vector** 類別實作 **List** 介面。
- **Stack** 類別是 **Vector** 的延伸，加了典型的堆疊操作：**push**, **pop**, 和 **peek**。
- **Hashtable** 類別實作 **Map**。
- **Properties** 類別是 **Hashtable** 的延伸，只使用 **String** 做為 **key** 與 **value**。
- 上述的每個集合都有 **elements** 方法，回傳一個 **Enumeration** 物件。**Enumeration** 是一個介面，它與 **Iterator** 介面類似但不相容。例如 **Iterator** 介面的 **hasNext** 方法，在 **Enumeration** 介面中是 **hasMoreElements**。

集合排序

Comparable 和 **Comparator** 介面對於集合排序非常有用。**Comparable** 介面提供實作類別使用原生的排序機制。**Comparator** 介面用於指定順序關係。它可以用於覆寫自然排序機制。這些介面對於集合排序非常有用。

Comparable 介面

Comparable 介面為 **java.lang** 套件內的一員。當您宣告一個類別，JVM 無法得知您要如何對該類別的物件排序。因此您可以透過實作 **Comparable** 介面，提供物件排序功能。只要這些物件有實作 **Comparable** 介面，您就可以將集合中的物件排序。

Java 類別中實作 **Comparable** 介面的有 **Byte**, **Long**, **String**, **Date**, 和 **Float**。這些類別在實作上使用數值比較而 **String** 類別則使用字母順序，**Date** 類別按照年月日順序。傳一個包含字串物件的 **ArrayList** 到 **Collections** 類別的靜態 **sort** 方法，會將 **ArrayList** 元件依照字母順序排序，若是包含 **Date** 物件則依照年月日順序排序，若是 **Integer** 物件則是依照數值大小排序。若要撰寫自定的 **Comparable** 型別，則必須實作 **Comparable** 介面的 **compareTo** 方法。程式 10-4 展示了如何實作 **Comparable** 介面。**Student** 類別實作 **Comparable** 介面，所以此類別物件可以互相比較。

程式 10-4 **Comparable** 介面實作範例

```

1  import java.util.*;
2  class Student implements Comparable {
3      String firstName, lastName;
4      int studentID=0;
5      double GPA=0.0;
6      public Student(String firstName, String lastName, int StudentID,
7          double GPA) {
8          if (firstName == null || lastName == null || StudentID == 0
9              || GPA == 0.0) {throw new IllegalArgumentException();}
10         this.firstName = firstName;
11         this.lastName = lastName;
12         this.studentID = studentID;
13         this.GPA = GPA;
14     }
15     public String firstName() { return firstName; }
16     public String lastName() { return lastName; }
17     public int studentID() { return studentID; }
18     public double GPA() { return GPA; }
19     // 實作 compareTo 方法.
20     public int compareTo(Object o) {

```

```

21     double f = GPA-((Student)o).GPA;
22     if (f == 0.0)
23         return 0;    // 0 表示相等
24     else if (f<0.0)
25         return -1;   // 負值表示小於或之前
26     else
27         return 1;    // 正值表示大於或之後
28 }
29 }

```

程式 10-5 **StudentList** 程式用來測試程式 10-4 **Comparable** 界面的實作。**StudentList** 程式產生 4 個 **Student** 物件然後輸出。因為 **Student** 物件被加入 **TreeSet**, **TreeSet** 是個排序的 **set**, 存放的物件都會被排序。

程式 10-5 **StudentList** 程式

```

1  import java.util.*;
2  public class StudentList {
3      public static void main(String[] args) {
4          TreeSet studentSet = new TreeSet();
5          studentSet.add(new Student("Mike", "Hauffmamn",101,4.0));
6          studentSet.add(new Student("John", "Lynn",102,2.8 ));
7          studentSet.add(new Student("Jim", "Max",103, 3.6));
8          studentSet.add(new Student("Kelly", "Grant",104,2.3));
9          Object[] studentArray = studentSet.toArray();
10         Student s;
11         for(Object obj : studentArray){
12             s = (Student) obj;
13             System.out.println("Name = %s %s ID = %d GPA =%f",
14                 s.firstName(), s.lastName(), s.studentID(), s.GPA());
15             = "+s.studentID+" GPA = "+s.GPA);
16         }
17     }
18 }

```

StudentList 程式輸出結果為:

```

Name = Kelly Grant ID = 0 GPA = 2.3
Name = John Lynn ID = 0 GPA = 2.8
Name = Jim Max ID = 0 GPA = 3.6
Name = Mike Hauffmamn ID = 0 GPA = 4.0

```

由結果觀察到學生是依照 **GPA** 來排序。這是因為 **TreeSet** 在排序元件時, 原本是用原生排序, 但在這個例子中則使用 **compareTo** 方法比較物件。

有些集合，例如 **TreeSet** 是有排序的。實作 **TreeSet** 需要知道如何比較元件順序。假如元件已經有原生排序，**TreeSet** 便可使用原生排序。否則您要撰寫一個。例如，**TreeSet** 有個建構子接收 **Comparator** 為參數：

```
TreeSet(Comparator comparator)
```

這個建構子會產生一個新的樹狀 **set**，依照指定的 **Comparator** 來排序。下面將會進一步討論 **Comparator** 介面的使用。

Comparator 介面

Comparator 介面於排序功能提供很大的彈性。例如，以上個 **Student** 類別為例，學生的排序是依照 **GPA**。沒辦法使用學生名字或是其他條件來排序。本節將會展示使用 **Comparator** 介面來排序的彈性。

Comparator 介面是 **java.util** 套件的成員。它是用於自定排序取代原生排序。例如，它可以不使用原生排序來排序物件。也可以用於排序未繼承 **Comparable** 介面的物件。

要撰寫自定的 **Comparator**，您需要實作介面的 **compare** 方法：

```
int compare(Object o1, Object o2)
```

這個方法會比較兩個物件的順序。如果第一個物件小於第二個物件則傳回負數，如果兩個相等傳回零，如果第一個物件大於第二個物件則傳回正數。程式 10-6 展示新版的 **Student** 類別。

程式 10-6 **Student** 類別

```

1  class Student {
2      String firstName, lastName;
3      int studentID=0;
4      double GPA=0.0;
5      public Student(String firstName, String lastName,
6          int StudentID, double GPA) {
7          if (firstName == null || lastName == null || StudentID == 0 ||
8              GPA == 0.0) throw new IllegalArgumentException();
9          this.firstName = firstName;
10         this.lastName = lastName;
11         this.studentID = studentID;
12         this.GPA = GPA;
13     }
14     public String firstName() { return firstName; }
15     public String lastName() { return lastName; }

```

```

16     public int studentID() { return studentID; }
17     public double GPA() { return GPA; }
18 }

```

可以產生幾個類別依照學生的 `firstName` 或 `lastName` 或 `studentID` 或 `GPA` 來排序。程式 10-7 的 `NameComp` 實作 `Comparator` 介面依學生的 `firstName` 來排序。

程式 10-7 `Comparator` 介面實作範例

```

1  import java.util.*;
2  public class NameComp implements Comparator {
3      public int compare(Object o1, Object o2) {
4          return
5              (((Student)o1).firstName.compareTo(((Student)o2).firstName));
6      }
7  }

```

程式 10-8 的 `GradeComp` 類別實作 `Comparator` 介面並依學生的 `GPA` 來排序。

程式 10-8 另一個 `Comparator` 介面實作範例

```

1  import java.util.*;
2  public class GradeComp implements Comparator {
3      public int compare(Object o1, Object o2) {
4          if (((Student)o1).GPA == ((Student)o2).GPA)
5              return 0;
6          else if (((Student)o1).GPA < ((Student)o2).GPA)
7              return -1;
8          else
9              return 1;
10     }
11 }

```

程式 10-9 的 `ComparatorTest` 類別測試 `NameComp` `Comparator`。注意在第 4 行，新的 `Comparator` 會傳給 `TreeSet`。

程式 10-9 `ComparatorTest` 程式

```

1  import java.util.*;
2  public class ComparatorTest {
3      public static void main(String[] args) {
4          Comparator c = new NameComp();
5          TreeSet studentSet = new TreeSet(c);
6          studentSet.add(new Student("Mike", "Hauffmamn", 101, 4.0));
7          studentSet.add(new Student("John", "Lynn", 102, 2.8 ));

```

```
8      studentSet.add(new Student("Jim", "Max",103, 3.6));
9      studentSet.add(new Student("Kelly", "Grant",104,2.3));
10     Object[] studentArray = studentSet.toArray();
11     Student s;
12     for(Object obj : studentArray){
13         s = (Student) obj;
14         System.out.println("Name = %s %s ID = %d GPA =%f",
15             s.firstName(), s.lastName(), s.studentID(), s.GPA());
16         = "+s.studentID+" GPA = "+s.GPA);
17     }
18 }
19 }
```

程式輸出結果為：

```
Name = Jim Max ID = 0 GPA = 3.6
Name = John Lynn ID = 0 GPA = 2.8
Name = Kelly Grant ID = 0 GPA = 2.3
Name = Mike Hauffmamn ID = 0 GPA = 4.0
```

如果程式 10-9 變更為使用 **GradeComp** 物件，學生就會依 **GPA** 來排序。

泛型 (Generics)

集合類別使用 **Object** 型別而可以允許存取不同的型別。當您取用時則必須強制轉型。但這樣的行為不符合型別安全 (**type-safe**) 的要求。

雖然目前的集合架構不支援同質集合（集合內的物件都是同一種型別，例如 **Date** 物件），但沒有任何機制可以防止不同物件被加入到集合內。而且取用時幾乎都得強制轉換。解決這個問題的方法就是使用泛型 (**generics**) 功能。**Java SE 5.0** 引入泛型，它提供編譯器集合所存取的物件型別的相關資訊。因此型別檢查在執行時期已經自動解決，也不需要再強制轉換型別。再加上自動的 **unboxing** 功能，泛型可以讓程式更簡單明瞭。程式 10-10 為未使用泛型前的程式：

程式 10-10 未使用泛型集合

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

程式 10-10 中將物件由 **list** 取出後，您需要強制轉換成 **Integer** 類別。執行時期程式還需要做型別檢查。若使用泛型之後，**ArrayList** 的宣告要變成 **ArrayList<Integer>**，告訴編譯器集合內物件的型別。當取出物件後，不用再強制轉換型別。程式 10-11 為使用泛型後的程式：

程式 10-11 使用泛型集合

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

泛型 **API** 非常適合自動 **unboxing** 功能。程式 10-12 為使用自動解包裝的範例程式：

程式 10-12U 於集合使用自動解包裝

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

程式 10-12 使用泛型宣告方式來宣告與初始化內含 **Integer** 物件的 **ArrayList** 實例。其結果是若加入非 **Integer** 型別物件，編譯時就會產生錯誤。



注意 --Java SE 5.0 預設會啟用泛型機制。使用 **javac** 命令列加上 **-source 1.4** 可以解除此項功能。

泛型 Set 範例

在下面的範例中，程式宣告一個 **Set<String>** 型別的變數 (**set**)，並指派一新的 **HashSet<String>** 物件。然後加入幾個 **String** 物件，輸出結果到標準輸出裝置。第 8 行會造成編譯錯誤（因為 **Integer** 不是 **String**）。程式 10-13 展示使用泛型的 **Set**。可以和程式 10-1 使用非泛型 **Set** 作比較。

程式 10-13 泛型 **Set** 範例

```

1  import java.util.*;
2  public class GenSetExample {
3      public static void main(String[] args) {
4          Set<String> set = new HashSet<String>();
5          set.add("one");
6          set.add("second");
7          set.add("3rd");
8          // 這行會有編譯錯誤
9          set.add(new Integer(4));
10         // 重複，不可以加
11         set.add("second");
12         System.out.println(set);
13     }
14 }
```

泛型 Map 範例

程式 10-14 `MapAcctRepository` 類別展示了泛型集合較實際的用法。程式中要儲存 `Account` 物件。它宣告一個 `HashMap<String, Account>` 型別的變數 (`accounts`)，定義兩個方法 `put()` 和 `get()` 來將元件由 `map` 存入或取出。

程式 10-14 泛型 `Map` 實作

```
1  public class MapAcctRepository {
2      HashMap<String, Account> accounts;
3
4      public MapAcctRepository() {
5          accounts = new HashMap<String, Account> ();
6      }
7
8      public Account get(String locator) {
9          Account acct = accounts.get(locator);
10         return acct;
11     }
12
13     public void put(Account account) {
14         String locator = account.getCustomer();
15         accounts.put(locator, account);
16     }
17 }
```


泛型：了解型別參數

本節將詳細說明泛型類別中型別參數的使用（包括泛型類別，建構子和方法宣告）。表 10-2 為非泛型 `ArrayList` 類別（Java SE 5.0 之前）和泛型 `ArrayList` 類別（Java SE 5.0 之後）比較表。

表 10-2 比較非泛型和泛型 `ArrayList` 類別

分類	非泛型類別	泛型類別
類別宣告	<code>public class ArrayList extends AbstractList implements List</code>	<code>public class ArrayList<E> extends AbstractList<E> implements List <E></code>
建構子宣告	<code>public ArrayList (int capacity)</code>	<code>public ArrayList (int capacity)</code>
方法宣告	<code>public void add(Object o)</code> <code>public Object get(int index)</code>	<code>public void add(E o)</code> <code>public E get(int index)</code>
變數宣告範例	<code>ArrayList list1;</code> <code>ArrayList list2;</code>	<code>ArrayList <String> a3;</code> <code>ArrayList <Date> a4;</code>
實例宣告範例	<code>list1 = new ArrayList(10);</code> <code>list2 = new ArrayList(10);</code>	<code>a3= new ArrayList<String> (10);</code> <code>a4= new ArrayList<Date> (10);</code>

兩者間主要的差別在於參數 `E` 的引入。下面的 `ArrayList` 宣告和初始，以 `String` 代替變數 `E`。

```
ArrayList <String> a3 = new ArrayList <String> (10);
```

下面的 `ArrayList` 宣告和初始，以 `Date` 代替變數 `E`。

```
ArrayList <Date> a3 = new ArrayList <Date> (10);
```

圖 10-3 展示泛型集合 API 的基本介面與實作類別的 UML 圖。

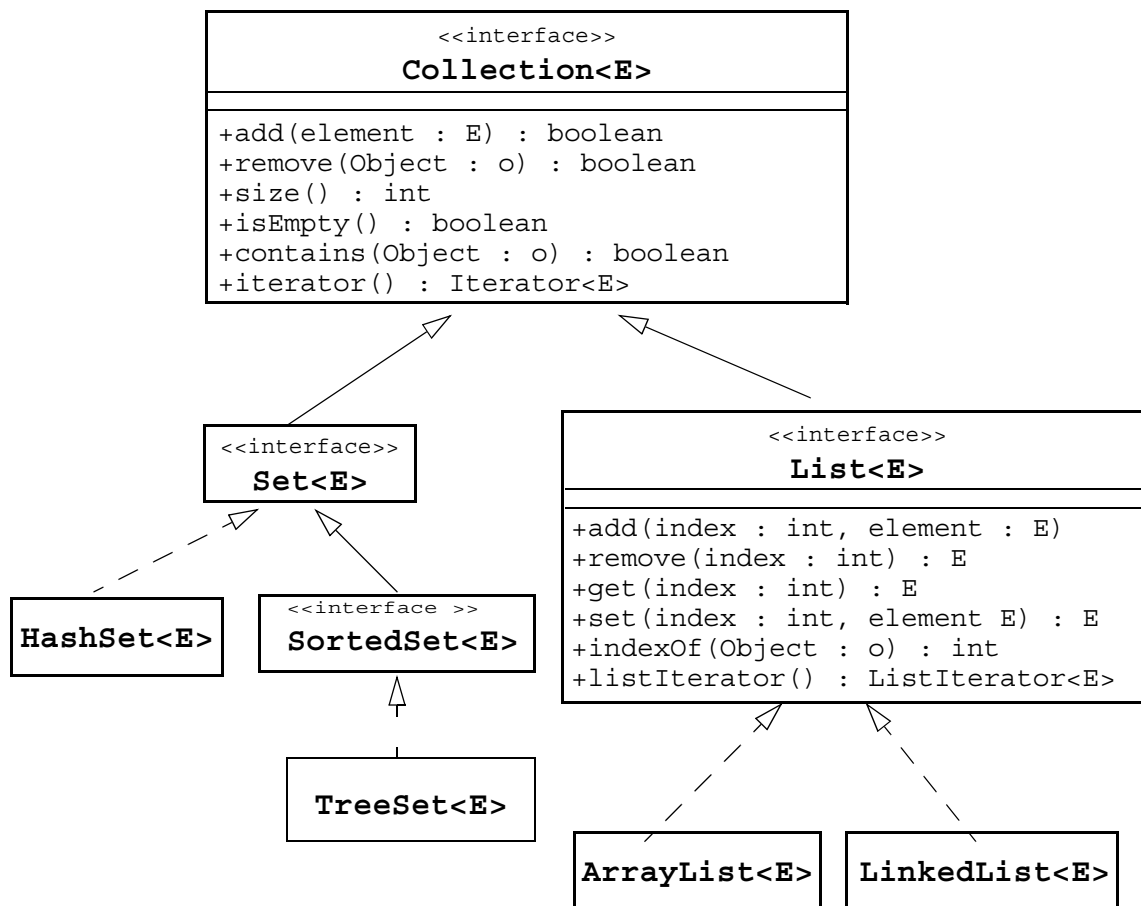


圖 10-3 泛型集合 API

注意 -- 每個介面的型別變數 **E** 代表集合內元件的型別。



通常型別參數會用大寫字母表示，**Java** 函式庫中的使用慣例為：

- **E** 代表集合內元件的型別。
- **K** 和 **V** 表成對鍵 - 值。
- **T** 代表其它型別。

萬用型別參數

下面討論將使用圖 10-4 **Account** 類別的繼承層級。.

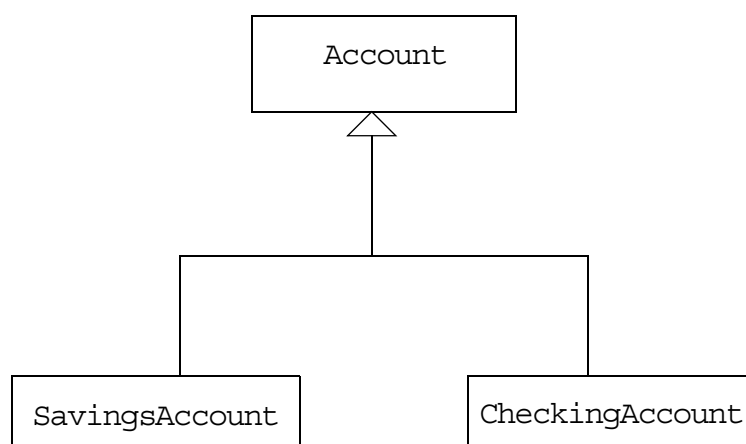


圖 10-4 **Account** 類別與其子類別

本節將介紹萬用型別參數。

型別安全保證

請參閱程式 10-15 的程式範例。

程式 10-15 討論型別安全的程式

```

List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
lc.add(new CheckingAccount("Fred")); // 沒問題
lc.add(new SavingsAccount("Fred")); // 編譯錯誤！
// 因此...
CheckingAccount ca = lc.get(0); // 安全，不須型別轉換
  
```

如果說泛型集合讓不適當的型別不可以加入集合，那麼它也可以保證由集合取出的物件一定與參數型別一樣。

不變性 (Invariance) 的挑戰

請參閱程式 10-16 的程式碼。

程式 10-16 指定不同型別集合的不變性

```

List<Account> la;
  
```

泛型：了解型別參數

```
List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
List<SavingsAccount> ls = new ArrayList<SavingsAccount>();
// 假如下列是可能的話...
la = lc;
la.add(new CheckingAccount("Fred"));
// 那下列也是可能的...
la = ls;
la.add(new CheckingAccount("Fred"));
// 所以...
SavingsAccount sa = ls.get(0); // 阿!!
```

事實上 `la=lc;` 是不合法的，所以即使 `CheckingAccount` 是一種 `Account` 但是 `ArrayList<CheckingAccount>` 不是 `ArrayList<Account>`。

為了有效保證型別安全，將一集合指派給另一內含不同型別的集合是不可以的，即使內含的型別是前一集合內含型別的子類別也一樣。

第一眼看的時候會覺得泛型集合有些缺乏彈性，且與傳統多型相較之下有點怪怪的。

Covariance Response

請參閱程式 10-17 程式碼。

程式 10-17 使用 **Covariance** 型別

```
void printIds(List<? extends Account> lea) {
    for (Account a : lea) {
        System.out.println(a.getAccountId());
    }
}

List<CheckingAccount> lc = new ArrayList<CheckingAccount>();
List<SavingsAccount> ls = new ArrayList<SavingsAccount>();

printIds(lc);
printIds(ls);
// 但是...
List<? extends Object> leo = lc; //OK
leo.add(new CheckingAccount("Fred")); // 編譯錯誤！
```

萬用字元的使用給予使用泛型集合一定程度的彈性。`printIds` 方法參數的宣告使用到萬用字元。萬用字元可以解釋為“任何 `Account` 家族物件”。上層的邊界 (`Account`) 表示集合內的元件可以安全的指派為 `Account` 變數。因此兩個同為內含 `Account` 子類別元件的集合可以傳給 `printIds` 方法。

Covariance Response 的設計用於讀取集合而非寫入集合。有一不變的基本原則，就是不能加入元素到使用萬用字元以及 **extends** 關鍵字集合。

泛型：重構已存在的非泛型程式

使用泛型集合，您可以不指定型別參數，稱為原生型別。這個特性可以提供與非泛型程式的相容性。在編譯時期，所有的泛型資訊會從程式中移除，剩下原生型別。如此一來，使得泛型與非泛型程式之間可以互通。在執行時期 `ArrayList<String>` 和 `ArrayList<Integer>` 都解釋成 `ArrayList` 的原生型別。

使用 **Java SE 5.0** 或是更新的版本編譯舊版非泛型的程式會有警告訊息。程式 10-18 展示編譯時期的警告訊息。

程式 10-18 `GenericWarning` 類別產生警告訊息

```
1  import java.util.*;
2  public class GenericWarning {
3      public static void main(String[] args) {
4          List list = new ArrayList();
5          list.add(0, new Integer(42));
6          int total = ((Integer)list.get(0)).intValue();
7      }
8  }
```

假如您使用下列命令編譯 `GenericWarning` 類別：

```
javac GenericWarning.java
```

請注意下面的警告訊息：

```
Note: GenericWarning.java uses unchecked or unsafe
operations.
```

```
Note: Recompile with -Xlint:unchecked for details.
```

替代方案為使用下列的命令：

```
javac -Xlint:unchecked GenericWarning.java
```

請注意下列的警告訊息：

```
GenericWarning.java:5: warning: [unchecked] unchecked call
to add(int,E) as a member of the raw type java.util.List
list.add(0, new Integer(42));
      ^
```

```
1 warning
```

雖然可以編譯也可以忽略警告訊息，但您還是應該修正程式。要解決這個警告訊息，要將第 4 行程式改為：

```
List<Integer> list = new ArrayList<Integer>();
```

迭代器 (Iterator)

您可以使用迭代器依次存取整個集合。基本的 **Iterator** 介面具有往前掃描任何集合的功能。以 **Set** 的例子來講，順序是無法決定的，但以 **List** 來說，就可以有順序性的往前存取 **list** 內的元件。**List** 物件也支援 **ListIterator**，可以往後存取，也可以加入元素和修改元素。

注意 -- 只要 **Set** 的實例是某些具有排序性的類別，則仍然具有順序性。例如，**TreeSet** 實例，它實作 **SortedSet**，因此具有順序性。

程式 10-19 展示迭代器的使用方式。

程式 10-19 Using Iterators

```

1  List list<Student> = new ArrayList<Student>();
2  // 加入元素
3  Iterator<Student> elements = list.iterator();
4  while (elements.hasNext()) {
5      System.out.println(elements.next());
6  }
```

圖 10-5 展示集合 API 中泛型 **Iterator** 介面的 UML 圖。

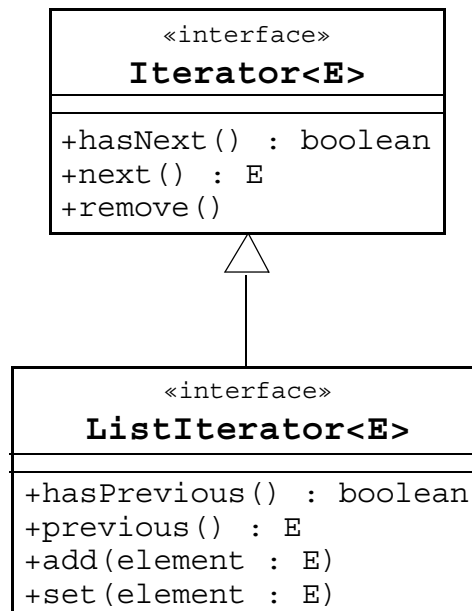


圖 10-5 UML 圖：泛型 **Iterator** 介面

remove 方法可以將目前的元素在存取過程移除。假如集合不支援移除便會丟出 **UnsupportedOperationException** 例外。

當使用 **ListIterator** 時，通常只會單一方向存取集合：不是使用 **next** 往前存取，就是使用 **previous** 往後存取。假如您呼叫 **next** 後又呼叫 **previous**，會取得同一個物件；呼叫 **previous** 又呼叫 **next** 後會得到同樣的結果，。

Set 方法會更改目前集合迭代器所指到的元素。**Add** 方法會加入新元素到迭代器索引之後的位置。因此假如您在 **add** 之後呼叫 **previous**，便會回傳剛加入的新元素。但是如果是呼叫 **next** 便無任何效果。假如集合不支援 **set** 或 **add** 則會丟出 **UnsupportedOperationException** 例外。

加強版的 for 迴圈

程式 10-20 展示了使用迭代器結合傳統的 **for** 迴圈存取集合內的元件的方式。

程式 10-20 使用迭代器和傳統的 **for** 迴圈

```
1  public void deleteAll(Collection<NameList> c) {
2      for (Iterator<NameList> i=c.iterator(); i.hasNext();)
3      {
4          NameList nl = i.next();
5          nl.deleteItem();
6      }
```

程式 10-20 中，**deleteAll** 方法於 **for** 迴圈中使用變數 **i** 次。這可能會是造成程式容易出錯。替代方案為使用加強版的 **for** 迴圈。加強版的 **for** 迴圈可以讓程式存取集合元件更簡單，易懂也更安全。加強版的 **for** 迴圈不需使用迭代器。程式 10-21 展示使用強版 **for** 迴圈的 **deleteAll** 方法。

程式 10-21 使用加強版 **for** 迴圈存取集合內元件

```
1  public void deleteAll(Collection<NameList> c) {
2      for (NameList nl: c) {
3          nl.deleteItem();
4      }
5  }
```

加強版的 **for** 迴圈功能，可以讓巢狀迴圈比以往更簡單容易和明瞭。其中一個原因是，加強版的 **for** 迴圈減少變數的模糊性。程式 10-22 包含了加強版的巢狀 **for** 迴圈。

程式 10-22 加強版的巢狀 **for** 迴圈

```
1 List<Subject> subjects=...;
2 List<Teacher> teachers=...;
3 List<Course> courseList = new ArrayList<Course>();
4 for (Subject subj: subjects) {
5     for (Teacher tchr: teachers) {
6         courseList.add(new Course(subj, tchr));
7     }
8 }
```


資料輸出輸入

單元重點

當完成這個單元後，你將能夠：

- 運用命令列參數及系統屬性來撰寫程式
- 了解 **Properties** 類別
- 善用文件的節點以及資料流
- 序列化及反序列化物件
- 區分並適當地使用資料流中的 **Reader/Writer**

其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, First Edition*. O' Reilly Media. 2003.

命令列 參數

當 Java 應用程式在命令列的視窗開始執行時，你可以在此程式中提供零個或是多個命令列參數。命令列參數允許使用者取得程式的組態資訊，這些參數都是字串，可能是單獨的文字，如 **arg1**，或是雙引號裏的字串，如 **"another arg"**。

在主程式類別名稱之後的參數會依序儲存在一個字串陣列中，並傳遞到主程式裏。以程式 11-1 作為例子。

程式 11-1 測試命令列參數

```
1 public class TestArgs {
2     public static void main(String[] args) {
3         for (int i = 0; i < args.length; i++) {
4             System.out.println("args[" + i + "] is '" + args[i] + "'");
5         }
6     }
7 }
```

此應用程式印出了每個命令列參數傳遞到 **TestArgs** 程式中的結果。

```
ava TestArgs arg1 arg2 "another arg"
args[0] is 'arg1'
args[1] is 'arg2'
args[2] is 'another arg'
```



注意 -- 當應用程式需要非字串類型的命令列參數時，如數字類型，並須先將字串參數用包覆類別轉換成相對應的基本型別，像是 **Integer.parseInt** 函式，可將字串參數轉換成相對應的整數型態。

系統屬性

系統屬性是另一種將應用程式參數化的機制。屬性是一種屬性名稱及其內容的對應關係；這二者皆為字串。而 **Properties** 類別正是用來表示這一類的對應關係。**System.getProperties()** 會回傳系統的 **Properties** 物件。而 **System.getProperties(String)** 則會用字串回傳以所傳入參數為名的屬性值。另外還有一個函式 **System.getProperties(String,String)**，讓你可以在以第一個參數為名稱的屬性值不存在時，把第二個參數設為預設值。



注意 -- 每一個 java 虛擬機器的實作，都必需要提供一組預設的屬性。（參見 **System.getProperties** 函式的文件以獲得更多資訊）。某些特別的虛擬機器廠商，提供了更多的屬性。

其它還有些在包覆類別裏的靜態轉型函式：**Boolean.getBoolean(String)**，**Integer.getInteger(String)**，和 **Long.getLong(String)**。這些傳入的字串是指屬性的名稱。如果此屬性不存在的話，則會依狀況分別傳回 **false** 或 **null**。

Properties 類別

Properties 類別包含了屬性名稱及內容之間的對應關係。它有兩個擷取屬性值的主要方法，`getProperty(String)` 及 `getProperty(String,String)`，後者提供額外的功能，就是當之前定義屬性名稱的值不存在時，可以回傳某個預設值。

你可以使用 `propertyNames` 方法，然後一個一個列出所屬性的名稱。然後利用屬性名稱呼叫 `getProperty`，你可以得到所有對應屬性名稱的值。

最後，屬性的集合可以透過 I/O 資料流的 `store` 和 `load` 方法存取。

程式 11-2 列出當程式執行時，所有屬性的集合

```

1  import java.util.Properties;
2
3  public class TestProperties {
4      public static void main(String[] args) {
5          Properties props = System.getProperties();
6          props.list(System.out);
7      }
8  }
```

第 5 行的程式取得系統的屬性，而第六行利用 **Properties** 類別的 `list` 函式，印出所有的屬性。

java -DmyProp=theValue TestProperties

下面的結果是從輸出結果中擷取出來的一部份。

```

java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\jse\jdk1.6.0\jre\bin
java.vm.version=1.6.0-b105
java.vm.vendor=Sun Microsystems Inc.
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=US
myProp=theValue
```

輸出入資料流的基礎理論

資料流是指資料從某一來源流向另一目的地。基本上你寫的程式是資料流的一端，而其它地方（如：檔案）則可為另一端。來源端以及目標端可分別稱為輸入流及輸出流。你可以從輸入流讀取，但是不能寫入；相反的，你可以寫入資料至輸出流，但是不可讀取資料。表 11-1 為基礎的資料流類別。

表 11-1 基本資料流類別

資料流	位元組資料流	字元資料流
來源資料流	InputStream	Reader
目標資料流	OutputStream	Writer

資料流中的資料

Java 支援兩種在資料流裏的型別，位元組資料流和 Unicode 字元資料流。基本上而言，**stream** 一詞指的是位元組資料流而 **reader** 和 **writer** 等詞則是指字元資料流。

更仔細來說，字元輸入流是由 **Reader** 的子類別來實作，而字元輸出流則是由 **Writer** 的子類別來實作。至於位元組輸入流，則由 **InputStream** 的子類別實作，而位元組輸出流則由 **OutputStream** 的子類別實作。

位元組資料流 (Byte Streams)

以下將敘述位元組串流的基本觀念。

InputStream 類別的方法

以下三個方法可從資料輸入流讀取資料：

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

第一個方法回傳一個整數，代表讀取的位元組資料，或是表示檔案結尾的 -1。其它兩個方法則是從資料流中把讀取到的資料放入一個位元組陣列，並傳回已經讀取的位元組數量。在第三個方法裏所使用的兩個整數參數，是用來指定在陣列中必須被填入的範圍。



注意 -- 為了增進程式效能，儘可能在可行的範圍內使用最大區塊 (**block**) 讀取資料，或是使用緩衝資料流 (**buffered streams**)。.

```
void close()
```

當您使用完資料流之後，請務必關閉它。如果您使用到多個資料流時，可以用過濾器資料流 (**filter streams**) 來作關閉最外層的資料流，如此一來也會關閉其它相連的資料流。(審按：這裏指的是 **Input Streams** 常透過 **decorator** 樣式串接在一起，關閉最外層的資料流，則串在後面的資料流的 **close()** 方法也會跟著被呼叫)

```
int available()
```

此方法是用來得知在資料流中還有多少位元組的資料可被讀取。在呼叫此方法後，在實際的讀取過程中，可能會傳回更多的位元組。

```
long skip(long n)
```

此程式會在資料流裏，跳過指定長度的位元組。

```
boolean markSupported()
void mark(int readlimit)
void reset()
```

如果系統支援的話，您可以用上述方法在資料流中執行將資料放回 (push-back) 的動作。當 `mark()` 及 `reset()` 可以在特定資料流上運作時，`markSupport()` 會傳回 `True`。而 `mark(int)` 則是會在目前資料流的讀取位置留下註記，並保留一塊不小於指定參數的記憶體空間。在 `mark(int)` 裏的參數，指定了當呼叫 `reset()` 時可以再次被讀取的空間大小。在接下來的 `reset()` 被呼叫後，使用 `reset()` 則回傳一資料流並指向你剛註記的讀取位置。當已讀取過標示的緩衝區時，使用 `reset()` 則不會有任何的反應。

OutputStream 類別的方法

下列三個方法可寫入資料至輸出流：

```
void write(int)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

跟輸入類似，儘可能在可行的範圍內使用最大區塊 (block) 讀取資料

```
void close()
```

當你已經使用完資料輸出流時，記得關閉它。如果您使用到多個資料流時，可以用過濾器資料流 (filter streams) 來作關閉最外層的資料流，如此一來也會關閉其它相連的資料流。

你結束資料輸出流時，`close` 可用來表示關閉資料流。同樣的，當資料是一層層字串時，當你結束了最上層的程式，剩下 `stream` 的也會同時的結束。

```
void flush()
```

有時候，輸出流會在真正寫入之前先累積資料。`Flush()` 可協助你強制將資料寫出。

字元資料流 (Character Streams)

以下的章節將討論字元資料流的基本觀念。

Reader 類別的方法

下列三個函式可從 **reader** 讀取字元資料：

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

第一個函式可回傳一個整數，代表從資料流讀取的 **Unicode** 字元，-1 表示檔案結尾。其它兩個方法則是從資料流中把讀到的資料放入一個字元陣列，並傳回已讀取的字元數量。在第三個方法裏所使用的兩個整數參數，是用來指定在陣列中必須被填入的範圍。



注意 -- 為了效率，儘可能在可行的範圍內使用最大區塊 (**block**) 讀取資料。

```
void close()
boolean ready()
long skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

這些方法都和輸入流的方法類似。

Writer 類別的方法

下列方法可將資料寫入至 **writer**:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

和輸出流一樣，也可使用 **close** 和 **flush** 方法。

```
void close()
void flush()
```

節點資料流 (Node Streams)

在 JDK 中，有三種基本的節點類型（表 11-2）：

- 檔案
- 記憶體（如陣列，字串物件）
- 資料管線(Pipes)(執行程序或是執行緒間的資料通道；一端是負責傳送而另一端則是負責接收)。

我們可以產生新的節點資料流物件，不過同時也需要處理裝置驅動程式的原生函式。這些原生函式本身是不具備跨平台特性的。表 11-2 為節點的種類。

表 11-2 節點的種類。

類型	字元資料流	位元組資料流
File	FileReader FileWriter	FileInputStream FileOutputStream
Memory: array	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
Memory: string	StringReader StringWriter	N/A
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream

一個簡單的範例

程式 11-3 以第一個命令列參數為讀取的檔案名稱，然後用第二個命令列參數為輸出的檔案名稱，將原來的檔案複製一份。下面這行命令列示範了程式如何被執行：

```
java TestNodeStreams file1 file2
```

程式 11-3 The TestNodeStreams Program

```
1 import java.io.*;
2
3 public class TestNodeStreams {
4     public static void main(String[] args) {
```

```
5      try {
6          FileReader input = new FileReader(args[0]);
7          try {
8              FileWriter output = new FileWriter(args[1]);
9              try {
10                 char[]    buffer = new char[128];
11                 int        charsRead;
12
13                 // read the first buffer
14                 charsRead = input.read(buffer);
15                 while ( charsRead != -1 ) {
16                     // write buffer to the output file
17                     output.write(buffer, 0, charsRead);
18
19                     // read the next buffer
20                     charsRead = input.read(buffer);
21                 }
22
23                 } finally {
24                     output.close();}
25             } finally {
26                 input.close();}
27         } catch (IOException e) {
28             e.printStackTrace();
29         }
30     }
31 }
```

這個例子雖簡單，但是處理這些緩衝記憶體卻很枯燥乏味，且容易出錯。不過，倒是有一些類別可以幫你從資料流裏一次讀取一行並顯示出來。像是 **BufferedReader** 就是其中的一種。

緩衝記憶體資料流 (Buffered Streams)

程式 11-4 所呈現的功能與的程式 11-3 相同，不過 11-4 的例子使用 **BufferedReader** 以及 **BufferedWriter**。

程式 11-4 **TestBufferedStreams** 程式

```

1  import java.io.*;
2  public class TestBufferedStreams {
3      public static void main(String[] args) {
4          try {
5              FileReader input = new FileReader(args[0]);
6              BufferedReader bufInput = new BufferedReader(input);
7              try {
8                  FileWriter output = new FileWriter(args[1]);
9                  BufferedWriter bufOutput= new BufferedWriter(output);
10                 try {
11                     String line;
12                     // read the first line
13                     line = bufInput.readLine();
14                     while ( line != null ) {
15                         // write the line out to the output file
16                         bufOutput.write(line, 0, line.length());
17                         bufOutput.newLine();
18                         // read the next line
19                         line = bufInput.readLine();
20                     }
21                 } finally {
22                     bufOutput.close();
23                 }
24             } finally {
25                 bufInput.close();
26             }
27         } catch (IOException e) {
28             e.printStackTrace();
29         }
30     }
31 }

```

此程式的流程與前一個相同，不同的是這個例子運用了 **readLine** 方法（第 14 及 20 行），每次讀取一行文字資料並將傳回的字串設給變數 **line**，以提升資料讀取的效率。第 7 行的程式將 **FileReader** 的物件跟 **BufferedStream** 物件 串接起來。利用在這串接關係最外層的資料流物件 (**bufInput**)，來操控最內層的資料流物件 (**input**)

輸出入資料流的串接

程式通常很少只會使用一個資料流物件。相反的，一個程式會串連一系列的資料流來處理資料。圖 11-1 示範了輸入資料流的例子；在這個例子中，**FileSteam** 的物件利用資料緩衝區增加了程式效率，接下來再透過資料輸入流 (**DataInputStream**) 轉回 **Java** 的基本類型。

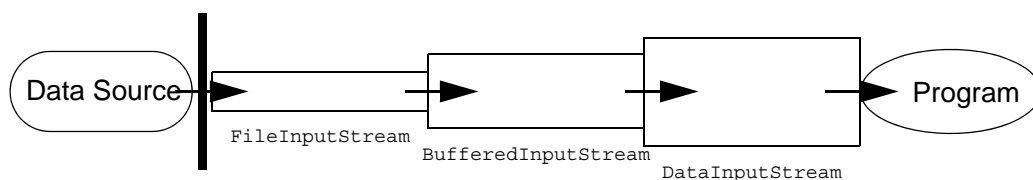


圖 11-1 輸入資料流串接範例

圖 11-2 示範了輸出資料流的過程，從資料被寫入，被放入緩衝區以及最後寫入檔案中。

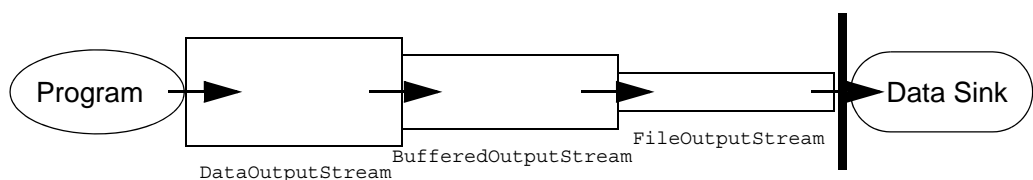


圖 11-2 輸出資料流串接範例

處理資料流 (Processing Streams)

處理資料流是將其他資料流的資料做某種的轉換，也稱之為過濾性資料流 (filter streams)。過濾性輸入流就是透過已存在的輸入流產生連結關係而來。當連結關係建立後，當你試著從過濾性輸入流的物件讀取時，它可提供連結關係中其他串流物件所讀取的字元。因此可讓你將原始資料轉換成對其他應用程式有用的形式。表 11-3 列出在 `java.io` 套件內所有內建的處理資料流。

表 11-3 依類型排序的處理資料流類型

類型	字元資料流	位元組資料流
緩衝型	<code>BufferedReader</code> <code>BufferedWriter</code>	<code>BufferedInputStream</code> <code>BufferedOutputStream</code>
過濾型	<code>FilterReader</code> <code>FilterWriter</code>	<code>FilterInputStream</code> <code>FilterOutputStream</code>
位元組與字元組間的轉換	<code>InputStreamReader</code> <code>OutputStreamWriter</code>	
* 物件序列化		<code>ObjectInputStream</code> <code>ObjectOutputStream</code>
資料轉換		<code>DataInputStream</code> <code>DataOutputStream</code>
記錄行數	<code>LineNumberReader</code>	<code>LineNumberInputStream</code>
資料預覽	<code>PushbackReader</code>	<code>PushbackInputStream</code>
輸出型	<code>PrintWriter</code>	<code>PrintStream</code>



注意 -- `FilterStreamXyz` 屬於抽象類別，並沒有辦法直接使用。你要先繼承寫出子類別來實作您的處理資料流。

要建立一個新的資料流並不難，下節將會有詳細的介紹。



注意 -- 物件序列化將在後面的章節討論。

基本的位元組串流類別

圖 11-3 列出 `java.io` 套件內輸入位元資料流的類別架構。部份常用的資料流類別會在下節討論之。

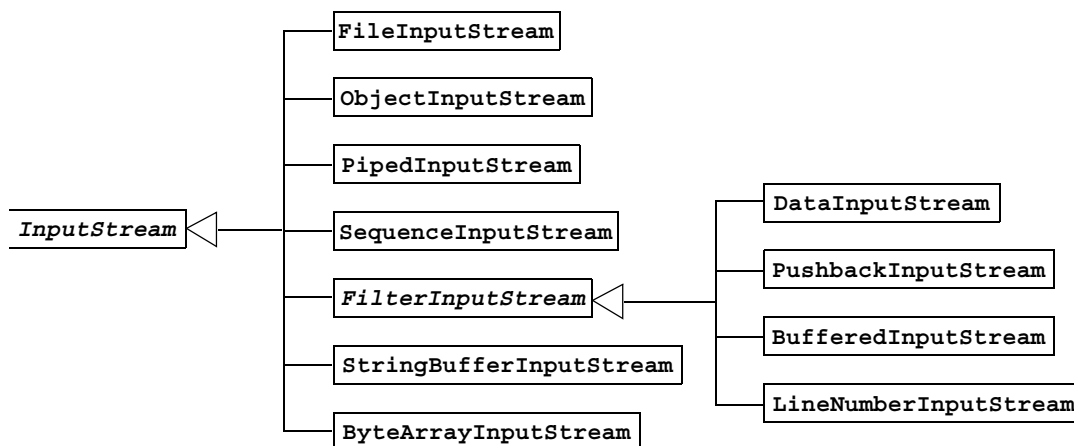


圖 11-3 `InputStream` 的類別架構

圖 11-4 列出 `java.io` 套件內輸出位元資料流的類別架構。

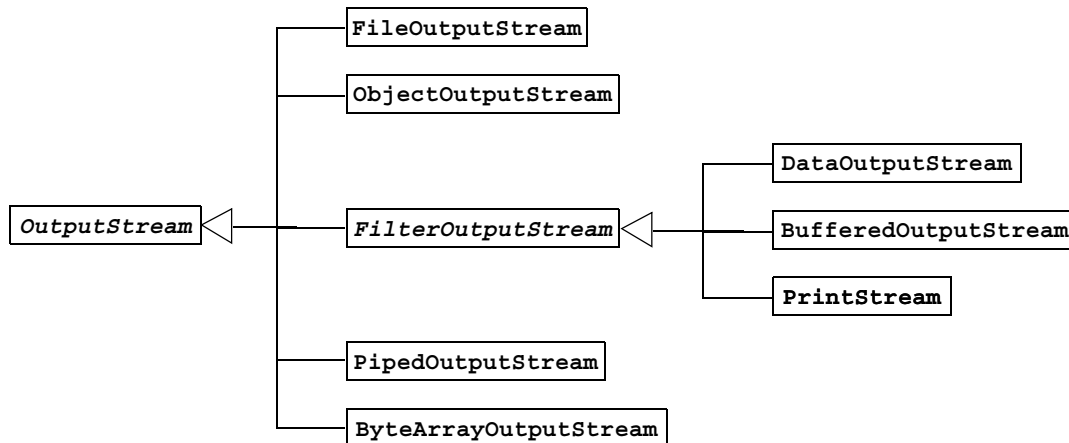


圖 11-4 `OutputStream` 的類別架構

FileInputStream 以及 FileOutputStream 類別

FileInputStream 以及 **FileOutputStream** 就如同其名字一樣，利用檔案來作為連結點的節點資料流。它們的建構子可讓你指定並連結到一個實體檔案的路徑。要建立一個 **FileInputStream**，相關的檔案要已經存在並且是可讀的。如果你要建立一個 **FileOutputStream**，若輸出檔案已存在則會被重新覆寫。

```
FileInputStream infile
    = new FileInputStream("myfile.dat");

FileOutputStream outfile
    = new FileOutputStream("results.dat");
```

BufferedInputStream 和 BufferedOutputStream 類別

使用 **BufferedInputStream** 和 **BufferedOutputStream** 以增加輸出入執行上的效能。

PipelineInputStream 和 PipeOutputStream 類別

利用管線資料流讓執行緒之間能彼此溝通，在某執行緒內的 **PipelineInputStream** 物件會從另一個執行緒內 **PipeOutputStream** 物件接受資料。管線資料流必需有輸入端也必需有輸出端才會有效。

DataInputStream 和 DataOutputStream 類別

這兩個類別稱為過濾性資料流。可以讀取或寫入各種型別的資料，如 **Java** 的基本型別和其它特別的格式。以下針對不同的基本型態提供不同的方法。

DataInputStream 的函式

如下，

```
byte readByte()  
long readLong()  
double readDouble()
```

DataOutputStream 的函式

如下，

```
void writeByte(byte)  
void writeLong(long)  
void writeDouble(double)
```

在 `DataInputStream` 的方法跟 `DataOutputStream` 的方法是相互對應的。這些資料流有可以讀取及寫入字串的方法，但請你不要使用，這些方法都已經廢棄不用 (`deprecated`)，取而代之的是 `readers` 及 `writers`，將會在稍後的章節討論。

ObjectInputStream 以及 ObjectOutputStream 類別

這兩個類別可將物件分別從資料流讀取和寫入至資料流。

把物件寫入至資料流中主要是將此物件的所有欄位值存入。如果欄位值是物件本身的話，則這些物件也應該被寫入至資料流。



注意 -- 如果物件的欄位是被設定為 `transient` 或是 `static`，他們的值則不會被寫入至資料流。這將會在之後的章節討論。

從資料流讀取物件包含了讀取物件的型別，建立物件以及將讀取到的資料放入物件。

物件永續化可以存在以檔案為主的資料流裏。

Java API 提供了一個標準的機制可以將寫入物件以及讀取物件的動作完全自動化。



注意 -- 如果此資料流屬於網路連線資料流，則物件資料在傳送前會先序列化，而接收端在接收後也會將物件反序列化。

圖 11-5 提供所有可以串聯一起的輸入流以及 **reader** 類別的整體圖。.

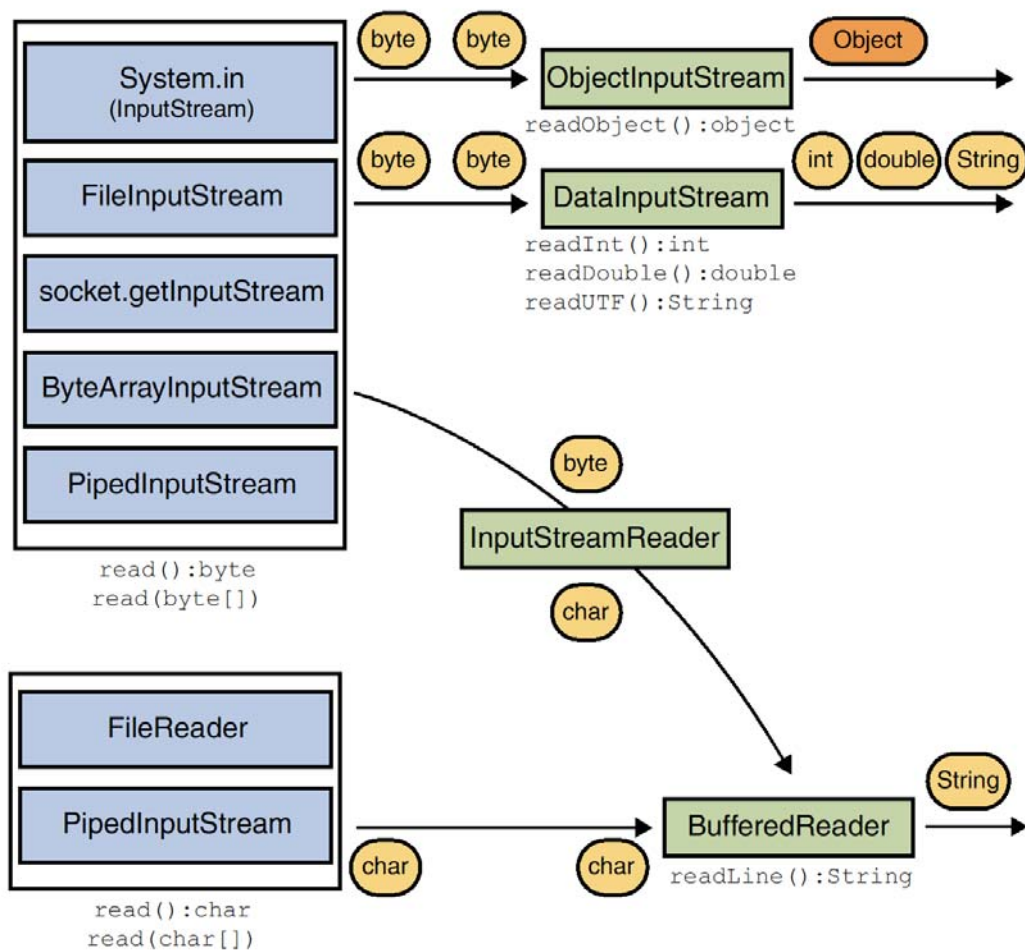


圖 11-5 輸入流的整體圖

圖 11-6 提供所有可以串聯一起的輸出流以及 **writer** 類別的整體圖。.

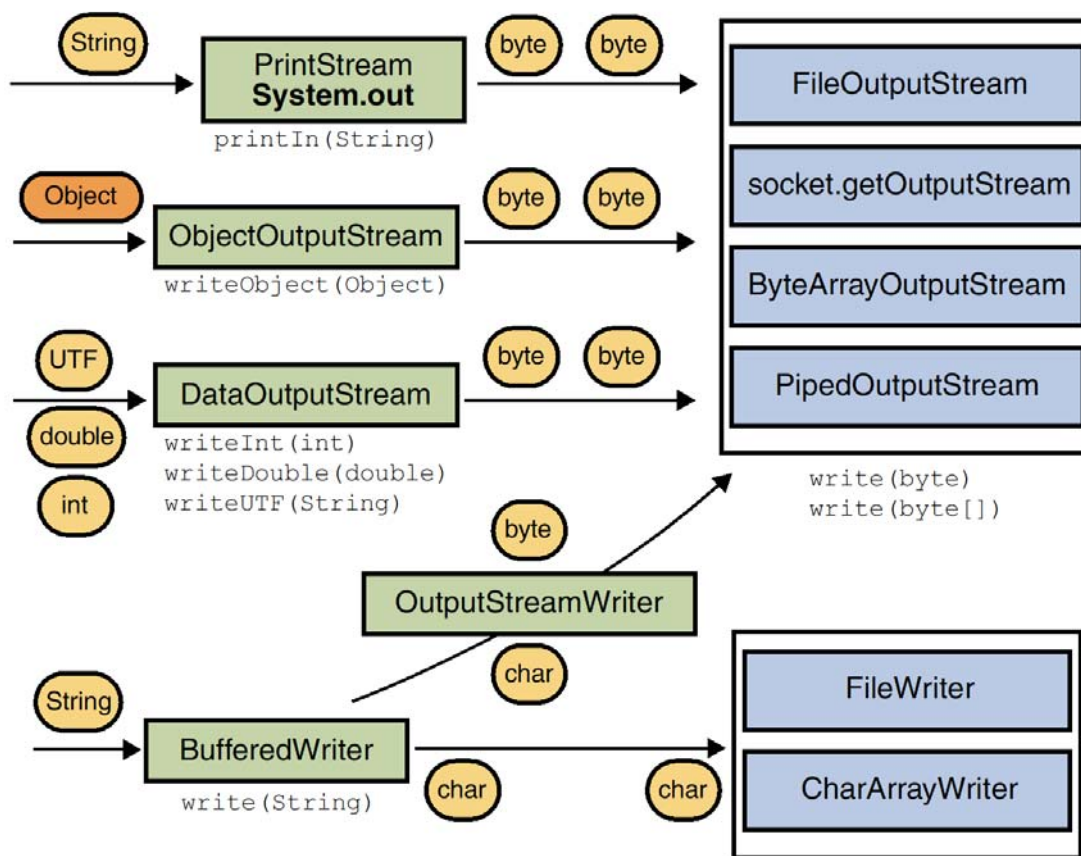


圖 11-6 輸出流的整體圖

序列化

將一個物件寫入到某種永久性儲存裝置稱之為永續化 (persistence)。

當你可以將一個物件儲存在一個磁碟或磁帶或是將它傳送到另一台電腦的記憶體時，我們會稱這個物件是可永續化 (persistent-capable) 的。那些不可永續化的物件只能存在於 Java 虛擬機器執行的時候。

物件序列化是一種把物件以一連串的位元組記錄下來，並可以在未來需要的時候，將這一連串的位元組重建出相同於先前物件資訊的機制。

當某一個類別的物件需要可以被序列化時，此類別必需實作 `java.io.Serializable` 介面。`Serializable` 是一個標示的介面，也就是說它不包含任何函式的定義，其目的只是用來標示有實作 `Serializable` 介面的物件具有序列化的能力。

物件序列化及物件圖 (Object Graphs)

物件被序列化時，只有成員變數的內容會被保留；但方法及建構式都並不會被序列化。當成員變數參考了某個物件時，如果被參考物件的類別可被序列化時，則其內部的成員變數也會一併被永續化。這個代表物件成員變數的樹狀圖組織成物件序列化的架構。

有些物件的類別並無法被序列化，因為這些物件所包含了短暫性而不可序列化的資料。例如 `java.io.FileInputStream` 及 `java.lang.Thread` 類別。如果一個可序列化的物件包含了一個無法序列化的元素，則整個序列化的運作將會中止而且會丟出一個 `NotSerializableException`。

倘若物件圖包含了一個無法序列化的物件參考，只要此物件的參考用關鍵字 `transient` 註明為短暫性，則此物件仍可進行序列化。

程式 11-5 示範了如何實作序列化介面。

程式 11-5 序列化的範例

```
1 public class MyClass implements Serializable {
2     public transient Thread myThread;
3     private String customerID;
4     private int total;
5 }
```

物件欄位的修飾詞（如 `public`、`protected`、`default` 和 `private`）對序列化的資料並沒有任何效用。資料是以位元組的方式寫入至資料流並以 Unicode 字串的方式存在檔案系統上。關鍵字 `transient` 能防止資料被序列化。

```

1  public class MyClass implements Serializable {
2      public transient Thread myThread;
3      private transient String customerID;
4      private int total;
5  }
```



注意 -- 儲存在靜態 (`static`) 欄位內的值並無法被序列化永續儲存。當物件在反序列化靜態欄位內的值後，原本靜態欄位的值將會存放在相對應的類別成員變數。

寫入及讀取物件資料流

從資料流中讀取一個物件，或是將一個物件寫入資料流中的過程是很簡單的。這個單元提供一些讀取跟寫入物件資料流的例子。

寫入物件資訊

程式 11-6 示範了將 `java.util.Date` 中的物件寫入檔案中。

程式 11-6 `SerializeDate` 類別

```

1  import java.io.*;
2  import java.util.Date;
3
4  public class SerializeDate {
5
6      SerializeDate() {
7          Date d = new Date ();
8
9          try {
10             FileOutputStream f =
11                 new FileOutputStream ("date.ser");
12             ObjectOutputStream s =
13                 new ObjectOutputStream (f);
14             s.writeObject (d);
15             s.close ();
16         } catch (IOException e) {
17             e.printStackTrace ();
18         }
```

```

19     }
20
21     public static void main (String args[]) {
22         new SerializeDate();
23     }
24 }

```

序列化的動作從第十四行開始，也就是當 `writeObject()` 被呼叫時。

讀取物件資訊

從檔案中讀取物件和將寫入物件的方式類似，但是有一點要注意——在 `readObject()` 回傳的資料流物件卻是以 **Object** 型式出現，所以在使用資料流相關的方法前，必需先被轉換至適當的類別名稱。程式 11-7 示範了如何從資料流中反序列化物件的資料。

程式 11-7 `DeSerializeDate` 類別

```

1  import java.io.*;
2  import java.util.Date;
3
4  public class DeSerializeDate {
5
6      DeSerializeDate () {
7          Date d = null;
8
9          try {
10             FileInputStream f =
11                 new FileInputStream ("date.ser");
12             ObjectInputStream s =
13                 new ObjectInputStream (f);
14             d = (Date) s.readObject ();
15             s.close ();
16         } catch (Exception e) {
17             e.printStackTrace ();
18         }
19
20         System.out.println(
21             "Deserialized Date object from date.ser");
22         System.out.println("Date: "+d);
23     }
24
25     public static void main (String args[]) {
26         new DeSerializeDate();
27     }

```

```
28 }
```

反序列化的動作從第十四行開始，也就是當 `readObject()` 被呼叫時。

基本的字元資料流類別

圖 11-7 示範了 `java.io` 套件內 `Reader` 字元資料流類別的層級。常用的部份會在下節討論。

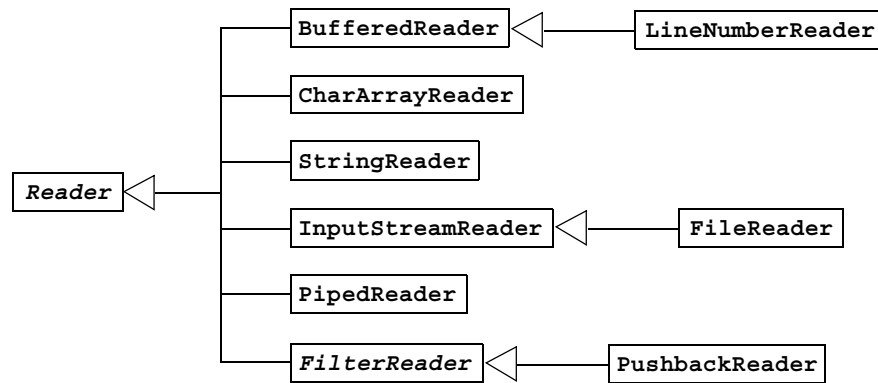


圖 11-7 Reader 類別的層級

圖 11-4 示範了 `java.io` 套件內 `Writer` 字元資料流類別的層級。

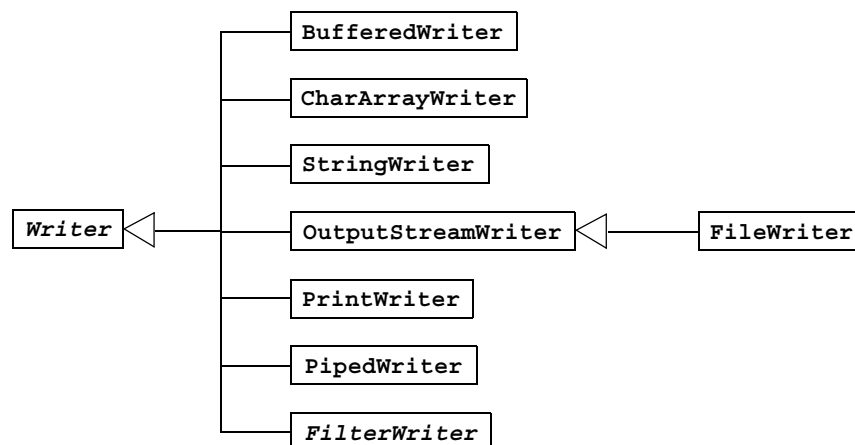


圖 11-8 Writer 類別的層級

InputStreamReader 以及 OutputStreamWriter 相關方法

`InputStreamReader` 和 `OutputStreamWriter` 是 `reader` 和 `writer` 中最重要類別。這些類別是介於位元資料流和文字 `reader` 和 `writers`。

在產生 `InputStreamReader` 或 `OutputStreamWriter` 物件時，轉換的原則是將 16 位元的 `Unicode` 及其他特定平台的表示格式之間作轉換。

位元組及字元間的轉換

基本上，如果你產生一個 `reader` 或 `writer` 來讀取資料流，則預設的轉換原則是針對預設平台的字元編碼和 `Unicode` 之間的轉換。在英語國家，位元的資料使用國際標準組織編碼 (ISO)8859-1。

並且還可以用系統支援語系編碼來指定其它的位元編碼方式。

如果你有安裝開發文件，你可以在下面的網址找到一連串的已支援語系編碼：<http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html>

利用這種編碼轉換的定義，`Java` 可以善用本機系統的字元編碼，同時又可以透過內部的萬國編碼來達到跨平台的特性。

使用其他字元組的編碼

如果你需要從非本機的編碼系統讀取字元（例如透過網路連線，從不同機器上讀取資料），你可建立 `InputStreamReader` 物件並配合明確的字元編碼，例如：

```
InputStreamReader ir
    = new InputStreamReader(System.in, "ISO-8859-1");
```

注意 -- 如果你是透過網路存取字元資料，則可套用此種格式。如果你沒有套用的話，你的程式將會像在本機上一樣，一直試著邊讀取邊轉換至可能不正確的格式。`ISO8859-1` 是 `Latin-1` 對應到 `ASCII` 的編碼定義。



FileReader 和 FileWriter 類別

FileReader 和 **FileWriter** 類別是節點資料流，有如 **Unicode** 的 **FileInputStream** 和 **FileOutputStream** 類別。

BufferedInputStream 和 BufferedOutputStream 類別

用 **BufferedInputStream** 和 **BufferedOutputStream** 類別來過濾字元資料流以增進輸出入操作的效率。

StringReader 和 StringWriter 類別

StringReader 和 **StringWriter** 類別是讀取或寫入 **Java** 字串物件的節點資料流。

假設你寫了一系列的報表類別及方法，這些方法的參數可以接一個 **Writer** 的參數（表示報表文字的目的地）。因為方法呼叫是透過多型，所以程式可以傳入一個 **FileWriter** 或是 **StringWriter** 物件而不用在意其真正型別。你可以利用 **FileWriter** 物件將報表寫入檔案。你也可以使用 **StringWriter** 將報表放入至記憶體的 **String** 物件以顯示在圖形介面的文字區塊。不管是哪種參數，報表本身的程式仍維持不變。

PipeReader 和 PipeWriter 類別

為了執行緒間的彼此溝通，你可以使用管線資料流。**PipedReader** 物件會在其中之一的執行緒接收從另一個執行緒裏 **PipedReader** 傳送出來的資料。要使用管線資料流，一定要有二個管線資料流物件分別在接收端以及傳送端。

主控台 (Console) 輸出入以及檔案輸出入

學習目標

在完成此單元的介紹後，您將可以：

- 解釋主控台輸出入
- 解釋檔案以及檔案輸出入的概念

其他資源

其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, First Edition*. O' Reilly Media. 2003.

主控台輸出入

大部份的應用程式都需要和使用者互動。這些互動有時是在主控台 (Console) 裡以文字輸出入的方式進行。(將鍵盤做為標準輸入並將終端機螢幕做為標準輸出)。

JDK 在 `java.lang.System` 類別裡提供了三個公開的靜態變數，來負責主控台的輸出入：

- 變數 `System.out` 是 `PrintStream` 物件，在被 Java 應用程式呼叫時，指的就是螢幕 (預設)。
- 變數 `System.in` 是 `InputStream` 物件，指的就是使用者的鍵盤 (預設)。
- 變數 `System.err` 是 `PrintStream` 物件，在被 Java 應用程式呼叫時，指的就是螢幕 (預設)。

您也可以使用靜態方法：`System.setOut`、`System.setIn` 以及 `System.setErr` 將資料導向至其他串流 (stream)。舉例來說，您可以將標準錯誤導向檔案串流。

將資料寫至標準輸出

您可以使用 `System.out.println(String)` 方法，將訊息寫至標準輸出。`PrintStream` 方法會將傳入的字串參數輸出至主控台，並在文字結尾加上換行的字元。下列方法也提供輸出其他資料型態，像是：基本資料型別，字元陣列，以及任何的物件。這些方法在輸出的結尾都會加上換行的字元。

```
void println(boolean)
void println(char)
void println(double)
void println(float)
void println(int)
void println(long)
void println(char[])
void println(Object)
```

也有一組類似 `println` 的多載方法，方法名稱為 `print`，該方法並不會在文字結尾加入換行的字元。

從標準輸入讀取資料

程式 12-1 的範例程式，展示了使用程式讀取主控台標準輸入的字串資訊：

程式 12-1 鍵盤輸入程式

```

1  import java.io.*;
2
3  public class KeyboardInput {
4      public static void main (String[] args) {
5          String s;
6          // 建立 buffered reader 讀取
7          // 從鍵盤輸入的每一行資料
8          InputStreamReader ir
9              = new InputStreamReader(System.in);
10         BufferedReader in = new BufferedReader(ir);
11         System.out.println("Unix: Type ctrl-d to exit." +
12             "\nWindows: Type ctrl-z to exit");
13     }

```

在程式第 5 行宣告字串變數 **s**，程式會記錄從標準輸入進來的每一列資料。第 8 至 10 行使用兩個物件圍住 **System.in**，是為了讀取從標準輸入串流 (stream) 而來的位元資料。**InputStreamReader(ir)** 讀取字元，並將原本的位元組字元轉換成 **Unicode** 字元。而 **BufferedReader(in)** 所提供的 **readLine** 方法讓程式可以一次從標準輸入讀取一整列的資料。



注意 -- 在 **UNIX** 平台使用 **Control-d** 字元做為標示檔案的結束。在微軟視窗環境中，在使用者按下 **enter** 鍵後，再接著 **Control-z** 表示檔案結尾。

下列為鍵盤輸入資料後續處理的程式碼：

```

14         try {
15             讀取每一列輸入的文字並隨即顯示 (echo)
16             s = in.readLine();
17             while ( s != null ) {
18                 System.out.println("Read: " + s);
19                 s = in.readLine();
20             }

```

第 16 行從標準輸入取得的第一列文字。而 **while** 迴圈（第 17 至 20 行）會反覆進行並列出目前取得的文字列，再準備讀取下一列的文字。我們可以將程式碼寫的再更簡潔一點（但比較不容易理解），像是：

```

while ( (s = in.readLine()) != null ) {
    System.out.println("Read: " + s);
}

```

```
}
```

由於 `readLine` 方法可能會產生輸出入例外，所以您必須將此置於 `try-catch` 區塊內。

```
21
22         // 關閉 buffered reader.
23         in.close();
```

第 23 行關閉輸入串流，釋放之前建立的串流物件所關連的系統資源。

```
24         } catch (IOException e) { // 捕捉任何產生的例外
25             e.printStackTrace();
26         }
27     }
28 }
```

最後，程式會處理所有其他可能會產生的輸出入例外。



注意 — 在第 23 行呼叫 `close` 方法應該是要在 `finally` 區塊裡執行。之所以沒有如此撰寫範例程式，是為了讓程式更為精簡。

簡易的格式化輸出

Java 程式語言 1.5 版本提供類似 C 語言 **printf** 函式的功能。它讓程式能夠將欲輸出的資料格式化。這使得程式設計師可以使用過去的程式碼。

您可以使用 **printf**，就像是用 C 以及 C++ 的語法一樣

```
System.out.printf("%s %5d %f%n",name,id,salary);
```

此格式化的功能也可以應用於字串類別的 **format** 方法。下列的程式所產生輸出結果和上述程式碼是一樣的：

```
String s = String.format("%s %5d %f%n",name,id,salary);  
System.out.print(s);
```

表 12-1 列示部份常用的格式化程式碼。

表 12-1 常用的格式化程式碼

程式碼	描述
%s	格式化字串參數值，通常會呼叫該物件的 toString 方法
%d %o %x	格式化整數，像是 10 進位、8 進位以及 16 進位的數值
%f %g	格式化浮點數。%g 則是代表使用科學符號
%n	在字串或串流加入換行符號
%%	在字串或串流加入百分比 (%) 符號

您可以使用 %n，來表示換行，可代替過去所使用的 \n，這方式適用於每個作業平台。

注意 -- 有關更多 **printf** 的功能以及格式化程式碼，請參考 **java.util.Formatter** 類別的 API 文件。



簡易格式化輸入

Scanner 類別提供格式化輸入的功能。它是 **java.util** 套件的一部份，提供方法可取得基本資料型別以及字串，並會等待使用者的輸入。請見程式 12-2 的範例程式。

程式 12-2 範例程式 – **Scanner** 類別

```
1  import java.io.*;
2  import java.util.Scanner;
3  public class ScanTest {
4      public static void main(String [] args) {
5          Scanner s = new Scanner(System.in);
6          String param = s.next();
7          System.out.println("the param 1" + param);
8          int value = s.nextInt();
9          System.out.println("second param" + value);
10         s.close();
11     }
12 }
```

在程式 12-2，第 5 行建立 **Scanner** 參考，並將主控台輸入做為參數。第 6 行使用 **Scanner** 的 **next** 方法，從指定的輸入取得字串值。第 8 行使用 **Scanner** 的 **nextInt** 方法，從指定的輸入取得整數值。第 10 行關閉使用者的輸入。



注意 — **Scanner** 類別可以將格式化輸入的資料拆解成幾個不同的基本資料型別以及字串。您可以使用正規表示式 (**regular expressions**) 來檢視串流。更多相關的資料，請見下列連結：

<http://java.sun.com/docs/books/tutorial/essential/io/scanning.html>

檔案以及檔案輸出入

Java 技術包含許多輸出入串流，這些串流已在前個章節描述過了。這個章節介紹幾個簡單的範例，從檔案讀取資料，以及將資料寫入至檔案，都是著重在字元資料的處理，包括：

- 建立檔案物件
- 操作檔案物件
- 讀取及寫入檔案串流

建立一個新的檔案物件

File 類別提供許多功能，包含檔案操作及取得檔案的相關資訊。在 **Java** 技術裡，目錄好比是另一個檔案。您可以建立檔案物件用來表示目錄，接著再用來定義其他的檔案，請見下列第三個的示範。

- ```
File myFile;
myFile = new File("myfile.txt");
```
- ```
myFile = new File("MyDocs", "myfile.txt");
```
- ```
File myDir = new File("MyDocs");
myFile = new File(myDir, "myfile.txt");
```

您所選擇使用的建構子通常會根據是否存取其他的檔案物件而定。舉例來說，如果您的應用程式只用到一個檔案，可以使用第一個建構子。然而，如果您要在同一個目錄裡使用好幾個檔案，使用第二個或第三個建構子可能會方便點。

**File** 類別所定義操作檔案的方法，都是可跨平台使用的。然而，它並不允許您存取檔案內容。



---

注意 — 您可以把檔案物件做為建構 **FileReader** 以及 **FileWriter** 物件時的參數，代替原本使用的字串。這樣一來您就不需依賴本機檔案系統的規定，一般都會建議使用此方式。

---



## 檔案測試以及工具

在您建立檔案物件後，您可以使用接下來章節所提供的方法，取得檔案的相關資訊。

### 檔案名稱

下列方法可回傳有關檔案名稱的資訊：

- `String getName()`
- `String getPath()`
- `String getAbsolutePath()`
- `String getParent()`
- `boolean renameTo(File newName)`

### 目錄工具

下列方法提供目錄工具：

- `boolean mkdir()`
- `String[] list()`

### 一般檔案資訊以及工具

下列方法可回傳一般的檔案資訊：

- `long lastModified()`
- `long length()`
- `boolean delete()`

### 檔案測試

下列方法可回傳檔案的相關屬性：

- `boolean exists()`
- `boolean canWrite()`
- `boolean canRead()`
- `boolean isFile()`
- `boolean isDirectory()`

- `boolean isAbsolute()`
- `boolean isHidden()`

## 檔案串流輸出入

Java SE 開發工具提供下列兩種方式，從檔案讀取資料：

- 使用 `FileReader` 類別來讀取字元
- 使用 `BufferedReader` 類別的 `readLine` 方法

Java SE 開發工具提供下列兩種方式，將資料寫入檔案：

- 使用 `FileWriter` 類別來寫入字元
- 使用 `PrintWriter` 類別的 `Print` 以及 `println` 方法

### 檔案輸入範例

程式 12-3 從檔案讀取資料，隨即將讀入的資料列出至標準輸出，也就是將檔案內容輸出到螢幕。

程式 12-3 從檔案讀取資料

```
1 import java.io.*;
2 public class ReadFile {
3 public static void main (String [] args) {
4 // 建立檔案
5 File file = new File(args[0]);
6
7 try {
8 // 檔案輸入範例
9 // 從檔案讀取每一列資料
10 BufferedReader in
11 = new BufferedReader(new FileReader(file));
12 String s;
13 }
```

第 5 行是將命令列取得的第一個參數值，建立一個新的檔案物件。第 10 以及第 11 行在建立的 `buffer reader` 時使用了 `file reader`。如果檔案不存在，則程式會產生 `FileNotFoundException` 例外。

程式 12-4 從檔案讀取資料，並將讀入的資料列出至標準輸出，也就是將檔案內容輸出至螢幕。

程式 12-4 列出檔案內容

```

14 try {
15 // 從檔案讀取每一列的資料
16 s = in.readLine();
17 while (s != null) {
18 System.out.println("Read: " + s);
19 s = in.readLine();
20 }
21 } finally {
22 // 關閉緩衝讀取物件
23 in.close();
24 }
25
26 } catch (FileNotFoundException e1) {
27 // 當檔案不存在時
28 System.err.println("File not found: " + file);
29
30 } catch (IOException e2) {
31 // 捕捉所有其他的 IO 例外
32 e2.printStackTrace();
33 }
34
35 }
36 }

```

在程式 12-4 裡，在第 17 至 20 行的 **while** 迴圈程式碼和鍵盤輸入程式（在 12-4 頁的程式 12-1）是一樣的；使用 **buffer reader** 物件讀取輸入的每一行資料，並隨即將資料輸出至標準輸出。

第 23 行關閉 **buffer reader** 物件，它會一併關閉關連的 **file reader** 物件。

在第 26 至 28 行的例外處理程式碼是負責捕捉，由 **file reader** 建構子所產生的例外狀況。而第 30 至 33 行處理所有可能產生的其他輸出入例外（針對 **readLine** 以及 **close** 方法）。

## 檔案輸出範例程式

程式 12-5 讀取從鍵盤輸入的資料，並隨即將資料寫入檔案。

程式 12-5 檔案輸出範例

```

1 import java.io.*;
2
3 public class WriteFile {
4 public static void main (String[] args) {
5 // 建立檔案
6 File file = new File(args[0]);
7
8 try {
9 // 建立 buffer reader 物件
10 InputStreamReader isr
11 = new InputStreamReader(System.in);
12 BufferedReader in
13 = new BufferedReader(isr);
14 // 在此檔案建立 print writer 物件
15 PrintWriter out
16 = new PrintWriter(new FileWriter(file));
17 String s;

```

像 12-11 頁的程式 12-4，程式 12-5 的第 6 行根據取得的第一個參數值建立檔案物件。第 10 至 11 行的程式碼從位元串流 (**System.in**) 建立位元讀取串流。第 12 至 13 行程式碼為標準輸入建立 **buffer reader**。第 15 至 16 行程式碼使用第 6 行建立的檔案建立 **print writer**。

```

18 System.out.print("Enter file text. ");
19 System.out.println("[Type ctrl-d to stop.]");
20
21 // 讀取每一列輸入的文字
22 while ((s = in.readLine()) != null) {
23 out.println(s);
24 }
25

```

第 18 及 19 行提示使用者輸入資料以及需使用 **Control-d** 停止輸入。程式碼第 22 至 24 行會從輸入串流讀取資料，並一次輸出至檔案。



注意 — 在這個範例中必須使用 **Control-d** 字元（用來表示檔案結尾），而不能使用 **Control-c**，因為 **Control-c**，會在程式結束檔案串流之前，就中斷 JVM 的執行。

```

26 // 關閉 buffered reader
27 in.close();
28 out.close();
29
30 } catch (IOException e) {
31 // 捕捉所有其他的 IO 例外

```

```
32 e.printStackTrace();
33 }
34 }
35 }
```

第 27 以及 28 行程式碼關閉輸入以及輸出串流。第 30 至 33 行程式碼負責處理所以其他可能產生的輸出入例外。



# 使用網路技術實作多層架構應用程式

---

## 學習目標

在完成此單元的介紹後，您應該將可以：

- 開發建立網路連線的程式
- 使用 **ServerSocket** 以及 **Socket** 類別實作 **TCP/IP** 的客戶端以及伺服器端
- 描述以及使用 **URL** 類別

## 其他資源



其他資源 -- 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O' Reilly Media. 2005.



## 網路

接下來的章節將透過 **socket** 來介紹網路概念。

### Sockets

**Socket** 是一個特有的程式模型，指 **Processes** 之間通訊連結的端點。由於這是已廣泛使用的程式模型，**Socket** 這個名詞也廣為其他程式語言模型所採用，包括 **Java** 技術。

當在網路上進行 **Process** 之間的溝通時，**Java** 技術使用的是串流模型 (**stream model**)。Socket 使用了兩種串流：一個是輸入串流以及另一個是輸出串流。當一個程序在網路上送出資料至另外一個程序時，是將資料寫到和 **socket** 連結的輸出串流。而另一程序則是使用和 **socket** 連結的輸入串流讀取從該程序所送出的資料。

在建立網路連線之後，把串流連結至連線 (**connection**)，就像連結至另外一個串流一樣的作法。

### 建立連線 (Connection)

為了建立連線，其中一台機器上必須執行一個程式來等待連線，而另外一台機器必須試著和這台機器建立連線。這跟電話系統很類似，一方必須撥出電話，而另一方就是等待電話撥入。

我們在這個單元將會描述 TCP/IP 網路連線。圖 13-1 是一個網路連線的範例。

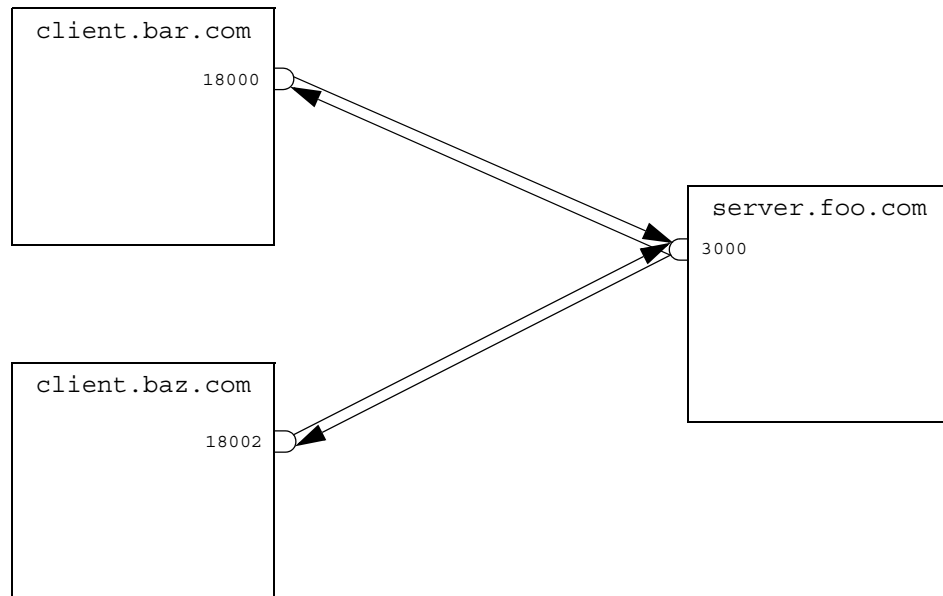


圖 13-1 一個網路連線的範例

伺服器是使用相同的通訊埠號和多個客戶端連線的。一個連線 (**connect**) 定義了四組數字：客戶端 IP 位址，客戶端通訊埠號 (**port number**)，伺服器 IP 位址以及伺服器通訊埠號。這種機制讓伺服器能夠處理來自同一個客戶端的多個連線，因為客戶端在每個連線一建立時，會配置不同的通訊埠號，客戶端通訊埠的配置是由作業系統控管。程式設計師並不會指定客戶端通訊埠，只會指定伺服器通訊埠。像電話系統在這樣的架構下就運作的很好。呼叫者（客戶端）必須要知道欲通話對象的電話號碼以及分機，但是客戶端本身並不需要使用自己的電話號碼（IP 位址）以及分機 (**port**)（雖然在底層系統會使用這個資訊）。

# Java 與網路技術

這個章節是描述 **Java** 技術應用在網路上的相關概念。

## 定址連線

當您要撥打電話時，您得先知道電話的號碼。同樣的，當您想要建立網路連接時，您就必須要知道遠端機器的位址或名稱。除此之外，網路連線還需要一個通訊埠號 (**port number**)，您可以將其想成電話的分機。在您連接到適當的電腦之後，還必須要知道連線的目的地。所以，像是您可以用特定的分機，跟會計部門通話；同樣的，您得也必須使用特定的通訊埠號和會計系統溝通。

## 通訊埠號

在 **TCP/IP** 系統使用 16 個位元的數字來表示通訊埠號 (**ports**)，其數值範圍為 0 至 65535。實務上，小於 1024 的通訊埠號都已經明文規定保留給預設的系統服務，您應該避免使用它們，除非您要使用的正是這些服務之一（像是 **telnet**、**Simple Mail Transport Protocol [SMTP] mail**、**ftp** 以及等等）。當伺服器通訊埠號被程式設計師指定用來進行特定的服務時，客戶端的本機作業系統也會指定目前尚未使用的通訊埠號。

客戶端以及伺服器兩者都必須先同意所要使用的通訊埠號。如果任何一方的系統不同意通訊埠號，則就無法進行連線。



注意 — 客戶端使用連接至伺服器的通訊埠號，照道理來說應該要和伺服器提供的相同。然而，如果伺服器提供服務使用 **port 2000**，而客戶端的通訊埠號可能並不是 **port 2000**，但客戶端本應該要使用 **port 2000** 連線至伺服器。這就是為什麼 **Socket** 物件針對取得埠號資料，提供了兩個方法，一個是 **getPort()** 方法（回傳的是此 **Socket** 連接的遠端通訊埠的號碼）以及另一個 **getLocalPort()**（回傳的是此 **Socket** 連接的本機通訊埠的號碼）。

## Java 網路模型

在 Java 程式語言裏，TCP/IP socket 連線機制實現在 `java.net` 套件類別中。圖 13-2 說明伺服器端以及客戶端間發生的動作。

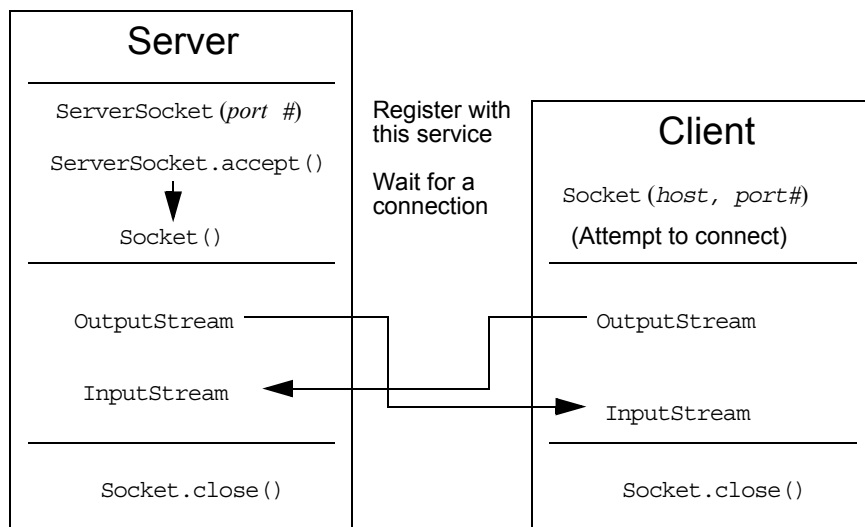


圖 13-2 TCP/IP socket 連線

在圖 13-2:

- 伺服器會分派一個通訊埠號。當客戶端要求連線時，伺服器會使用 `accept()` 方法開啟 socket 連線。
- 客戶端使用本機的通訊埠號建立連線。
- 客戶端以及伺服器兩者之間的溝通，使用 `InputStream` 以及 `OutputStream`。

## 迷你 TCP/IP 伺服器

TCP/IP 的伺服器應用程式根據 Java 程式語言提供的 **ServerSocket** 以及 **Socket** 網路類別。**ServerSocket** 類別負責主要建立伺服器連線的工作。

```

1 import java.net.*;
2 import java.io.*;
3
4 public class SimpleServer {
5 public static void main(String args[]) {
6 ServerSocket s = null;
7
8 // 在 port5432 登記您的服務
9 try {
10 s = new ServerSocket(5432);
11 } catch (IOException e) {
12 e.printStackTrace();
13 }
14
15 // 在迴圈中不斷執行監聽 / 接受
16 while (true) {
17 try {
18 // 在此處等待以及監聽連線
19 Socket s1 = s.accept();
20
21 // 取得和 Socket 連接的 outputStream
22 OutputStream slout = s1.getOutputStream();
23 BufferedWriter bw = new BufferedWriter(
24 new OutputStreamWriter(slout));
25
26 // 送出您的字串！
27 bw.write("Hello Net World!\n");
28
29 // 關閉連線，但不是 server socket
30 bw.close();
31 s1.close();
32 } catch (IOException e) {
33 e.printStackTrace();
34 } // try-catch 結束
35 } // while(true) 結束
36 } // 主方法結束
37 } // SimpleServer 程式結束

```

## 迷你 TCP/IP 客戶端

TCP/IP 的客戶端應用程式使用 **Socket** 類別。同樣的，大部份需要建立連線的工作都是由 **Socket** 類別所負責。客戶端所連接的伺服器，就是上一些所提到的“迷你 TCP/IP 伺服器”，會列出從伺服器送來的資訊到主控台 (Console)。

```
1 import java.net.*;
2 import java.io.*;
3
4 public class SimpleClient {
5 public static void main(String args[]) {
6 try {
7 // 於 port 5432, 開啟至伺服器的連線
8 // localhost 使用 d here
9 Socket s1 = new Socket("127.0.0.1", 5432);
10
11 // 從 socket 取得 inputStream
12 InputStream is = s1.getInputStream();
13 InputStreamReader irs = new InputStreamReader(is);
14 BufferedReader br = new BufferedReader(irs);
15 // 讀取輸入的資料並出至螢幕
16 System.out.println(br.readLine());
17
18 // 完成後, 關閉串流以及連接
19 br.close();
20 s1.close();
21 } catch (ConnectException connExc) {
22 System.err.println("Could not connect.");
23 } catch (IOException e) {
24 // 忽略
25 } //try-catch 結束
26 } // 主程式結束
27 } // SimpleClient 程式結束
```

## URL 類別

Java 程式可以使用 URL 類別連接至指定的 URL 位址存取內容。這個類別是 `java.net` 套件裡的類別。URL 類別提供了幾個連線和在 Internet 搜尋資訊的功能。

URL 類別可以用來建立以及解析 URL 位址，開啟指定 URL 位址的連線，以及從連線讀取和寫入資料。

### . 從 URL 存取資訊

下列步驟通常和連線至 URL 目的地以及存取 URL 目的地資料相關。

- 建立 URL 物件
- 解析 URL
- 連接至 URL 目的地
- 讀取或寫入內容至 URL 連線
- 關閉 URL 連線

下列章節將會進一步描述這些步驟。

#### 建立 URL

URL 類別提供了幾個建立 URL 物件的建構子。根據給予所指定的資料，這些建構方法都可以用來建立 URL。

最簡單的方法就是使用字串表示絕對 URL 位址。舉例來說：

```
URL sunAddress = new
URL("http://www.sun.com/products/index.jsp");
```

另外一個可供選擇的方式是指定相對路徑，而不是使用絕對路徑。

```
URL baseAddress = new URL("http://www.sun.com/");
```

```
URL sunAddress = new
URL(baseAddress, "products/index.jsp");
```

使用這個方式的優點就是可再使用相同的 `baseAddress` 存取另外一個頁面，舉例來說：

```
URL sunAddress = new
URL(baseAddress, "software/index.jsp");
```

另外一個建立 **URL** 物件的方式，就是已得知所要使用的通訊協定，機器名稱以及檔案名稱。假如使用通訊協定的通訊埠和預設的通訊埠是不一樣的話，還有另外一個建構 **URL** 方式，可以於參數指定所要使用的通訊埠。

### 解析 URL

**URL** 類別可以用來搜尋和 **URL** 的相關資訊。舉例來說，表 13-1 藉由下列的 **URL**，展示部份 **URL** 類別的方法以及使用這些方法時所會回傳的資訊：

<http://onesearch.sun.com/search/onesearch/index.jsp?qt=JAVA&charset=UTF-8>

表 13-1 解析 **URL** 的範例

| 方法                                 | 回傳                                                                  |
|------------------------------------|---------------------------------------------------------------------|
| <code>String getProtocol()</code>  | http                                                                |
| <code>String getAuthority()</code> | onesearch.sun.com                                                   |
| <code>String getHost()</code>      | onesearch.sun.com                                                   |
| <code>int getPort()</code>         | -1                                                                  |
| <code>int getDefaultPort()</code>  | 80                                                                  |
| <code>String getPath()</code>      | /search/onesearch/index.jsp                                         |
| <code>String getQuery()</code>     | qt=JAVA&charset=UTF-8                                               |
| <code>String getFile()</code>      | /search/onesearch/index.jsp?qt=JAVA&charset=UTF-8                   |
| <code>String getRef()</code>       | null                                                                |
| <code>Object getContent()</code>   | sun.net.www.protocol.http.HttpURLConnection\$HttpInputStream@addbf1 |
| <code>String getUserInfo()</code>  | null                                                                |



注意 --**URL** 並不需要具備表 13-1 的所有元件。若遇到這種狀況，方法將會回傳 **null** 值，見表 13-1。

`getPort()` 方法回傳 -1，這是因為該 **URL** 並沒有特別定義通訊埠。在這樣的情況下，**HTTP** 通訊協定所回傳的是預設埠號會是 80。





注意 -- 解析 URL 是一個選擇性的步驟，讓您從特定的 URL 來搜尋 URL 元件相關資訊。

## 連接至 URL 位址

在建立 URL 之後，URL 類別的 `openConnection()` 方法可以用來建立連線至 URL 位址。

```
URLConnection openConnection();
```

`openConnection` 方法建立了 `URLConnection` 的實例，它是用來代表連至 URL 的通訊連結。

`URLConnection` 類別可以用來搜尋以及管理一些連線的參數，像是連線逾時，內容長度以及類型，存取的時間，通訊協定標頭欄位，權限以及使用者快取。

在使用 `openConnection` 方法建立連接之後，在 `URLConnection` 的 `connect()` 方法是用來初始化連線。

## 讀取及寫入內容至連線

您可以使用 `URLConnection` 的 `getInputStream` 方法以及 `getOutputStream` 方法來讀取以及寫入連線，或者是使用在 URL 類別的 `openStream` 方法來開啟連線以及回傳串流。

在建立串流之後，讀取以及寫入資料至串流的方式，類似讀取及寫入資料至其他串流，例如檔案，記憶體以及 `pipes`。

下列範例使用 `openStream` 方法來開啟連接串流，並且讀取資料。程式 13-1 展示建立連接以及讀取內容的程式範例。

### 程式 13-1 URLEExample 類別

```
1 import java.net.*;
2 import java.io.*;
3
4 public class URLEExample {
5 public static void main(String[] args) throws Exception {
6 String oneLine;
7 URL url_address = new URL("http://www.sun.com/");
8 BufferedReader br= new BufferedReader(
```

```

9 new InputStreamReader(url_address.openStream()));
10 while ((oneLine = br.readLine()) != null) {
11 System.out.println(oneLine);
12 }
13 br.close();
14 }
15 }

```

**readLine()** 方法從開啟的串流讀取每一行的資料，並將資料儲存於區域變數 **oneLine**。在第 11 行的 **println** 指令，列出每一行的 HTML 程式。

下列輸出是擷取列在螢幕上的部份資料。

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
 <!-- BEGIN A0 COMPONENT V.2 -->
 <html lang="en-US">
 <head>
 <meta http-equiv="content-type" content="text/html; charset=UTF-8">
 <title> Sun Microsystems </title>
 <meta name="keywords" content="sun microsystems, sun, java, java
computing, solaris, sparc, unix, jini, computer systems, 伺服器, mission
critical, RAS, high availability, cluster, workgroup 伺服器, desktop,
workstation, storage, backup solutions, network computer, network
computing, hardware, software, service, consulting, support, training,
compiler, jdk, technical computing, scientific computing, high
performance, enterprise computing, staroffice, starportal, sun ray">
 <meta name="description" content="Sun Microsystems, Inc. The Network
Is The Computer[tm].">
 <meta http-equiv="content-language" content="en-US">
 <meta name="date" content="2006-03-14">
 <link rel="shortcut icon" href="/favicon.ico" type="image/x-icon">

```



注意 -- 考下列網址

<http://java.sun.com/docs/books/tutorial/networking/urls/readingWriting.html> 輸出至 **URLConnection** 物件的範例程式。

## 關閉連結

和其他的串流相同，被開啟的連接串流，在完成讀取或寫入串流的動作之後，應該要立即關閉。





# 實作多執行緒應用程式

---

## 學習目標

在完成此單元的介紹後，您將可以：

- 定義執行緒
- 用 **Java** 程式建立另一個執行緒，並在此執行緒中控制程式及使用資料
- 控制執行緒的執行以及撰寫跨平台的執行緒程式
- 介紹幾個建立執行緒的方法
- 使用 **synchronized** 以防止資料錯誤
- 說明執行緒之間的互動
- 使用 **wait** 以及 **notify** 來進行執行緒之間的溝通

## 其他資源



其他資源 — 在下列參考的資源中，提供了本章節所探討主題的相關資訊及細節：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O’ Reilly Media. 2005.
- Goetz, Peierls, Bloch, Bowbeer, Holmes, Lea. *Java Concurrency in Practice, First Edition*. Addison-Wesley Professional. 2006.

## 執行緒

若以簡單的觀點來解釋電腦系統運作的話，一個電腦系統首先必須具備一個 **CPU** 來執行運算，記憶體則用來存放 **CPU** 所要執行的程式及存放程式執行時所會使用的資料。在這個觀點裡，同一時間只會有一個工作在執行。但以更完整的觀點來看的話，現今所有的電腦系統環境中，幾乎都能夠同時執行多個工作。

您不需要知道多工作業是如何實現的，只需從程式設計的觀點來探討其行為。執行一個以上的工作和擁有一台以上的電腦的道理是類似的。在這個單元中，一個執行緒，或說是「執行環境 (Execution Context)」，裏面包裝了一個虛擬的 **CPU**，和其所擁有的程式及資料。而 `java.lang.Thread` 類別是讓您能夠建立以及管理執行緒的機制。



- 執行緒主要由三個部份組成（見圖 14-1 的說明）：
- ● 虛擬 CPU
- ● CPU 執行所需的程式
- ● 程式執行時所使用到的資料

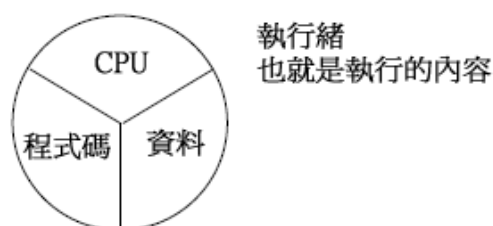


圖 14-1 執行緒

一個程式可以被多個執行緒所共用，且和資料無關。兩個執行緒所使用的程式是來自於相同的類別時，這兩個執行緒就共用了一個程式。

同樣地，資料也可以為多個執行緒所共用，而和程式無關。當兩個執行緒都擁有存取共同物件的權限時，就能夠存取相同的資料。

在 Java 程式裡，虛擬 CPU 封裝於 `java.lang.Thread` 類別的實例 (instance)。當執行緒被建立時，定義執行環境所用到的程式和資料會作為參數值，傳遞至執行緒的建構子。

## 建立執行緒

本節將說明如何建立執行緒，以及您如何將程式以及資料，做為建立執行緒的參數值，提供執行緒執行時使用。

建立執行緒，需要一個 **Runnable** 實例 (instance) 做為執行緒建構子的參數值。而 **Runnable** 的實例是來自於實現 **Runnable** 介面的類別（此介面定義了 **public void run()** 方法）。

範例程式如下所示：

```
1 public class ThreadTester {
2 public static void main(String args[]) {
3 HelloRunner r = new HelloRunner();
4 Thread t = new Thread(r);
5 t.start();
6 }
7 }
8
9 class HelloRunner implements Runnable {
10 public void run() {
11 int i = 0;
12 while (true) {
13 System.out.println("Hello " + i++);
14 if (i == 50) {
15 break;
16 }
17 }
18 }
19 }
```

首先，主方法會建立類別 **HelloRunner** 的實例 **r**。實例 **r** 擁有自己的資料，在這個例子是整數 **i**。因為實例 **r**，會做為傳遞至執行緒類別建構子的參數值，**r** 的整數 **i** 會在執行緒執行時所使用。執行緒總是從定義於 **Runnable** 實例的 **run** 方法開啟執行。



多執行緒程式環境讓您在建立多個執行緒時，可以使用同一個實現 **Runnable** 的實例。像是，您可以這麼做：

```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```

這個例子裡，這兩個執行緒分享相同的資料以及程式碼。

總而言之，執行緒就是執行緒物件的實例。執行緒會從 **Runnable** 實例的 **run** 方法開始執行。而執行緒所使用的資料，是來自於 **Runnable** 的實例所指定的資料，這個實例會做為參數值，傳入執行緒的建構子（圖 14-2）。

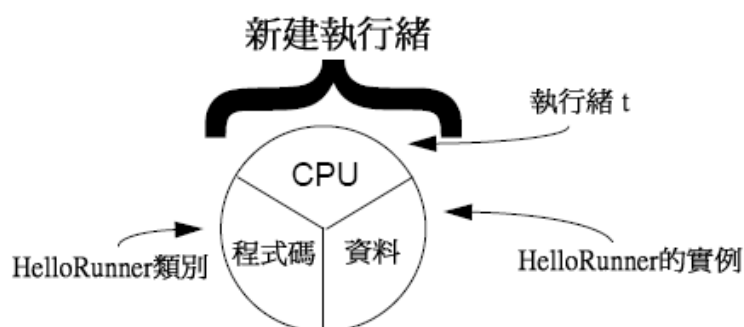


圖 14-2 建立執行緒

## 啟動執行緒

一個新建立的執行緒，並不會自動執行。您必須呼叫它的 **start** 方法。舉例來說，您可以參考前一個範例程式的第 5 行：

```
t.start();
```

呼叫 **start** 時，虛擬 CPU 會使執行緒進入可執行 (**runnable**) 狀態，代表系統的排程器可以為該執行緒安排執行。但這並不是代表執行緒會被馬上執行。

## 執行緒排程

通常在 Java 執行緒排程屬於優先權式多工 (preemptive) 型，而非時間間隔型 (time-sliced 指每一個執行緒都享有一定的 CPU 執行時間)。常有人以為優先權式多工 (preemptive) 的進行方式就是以時間間隔的型式來達成。

優先權式多工排程模式中會有許多可執行的執行緒，但是只會有一個執行緒是正在執行的。當輪到該執行緒所分派執行的時間，就會轉為可執行狀態，除非有另一個更高優先權的執行緒出現。

執行緒可能會因為一些原因，終止可執行 (runnable) 的狀態，例如被凍結 (blocked)。執行緒的程式可以呼叫 `Thread.sleep()`，來刻意要求執行緒暫停一段固定的時間。這麼做很可能是因為執行緒必須等待要存取的資料，所以不能繼續執行，直到該資料能夠使用為止。

所有可執行的執行緒會根據優先權，置於可執行池 (runnable pool) 裡。當凍結的執行緒變成可執行時，它會被放入適當的可執行池。

執行緒物件在其生命週期會經歷幾種不同的狀態，見圖 14-3。

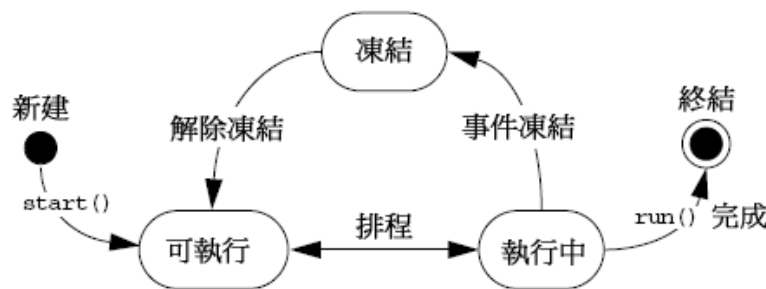


圖 14-3 執行緒基本狀態圖

雖然執行緒變成可執行的狀態，但是它並不是馬上就會被執行。一個 CPU 在同一時間一次只能執行一個動作。接下來我們將描述 CPU 資源是如何被配置給多個執行緒使用。

我們知道 Java 技術的執行緒並不一定是時間間隔 (time-sliced) 型式，因此您可以在執行緒的程式裡給予其他執行緒執行的機會。您可以使用 `sleep` 方法並指定間隔時間來達成這個目的，見程式 14-1。

程式 14-1 執行緒排程範例

```

1 public class Runner implements Runnable {
2 public void run() {
3 while (true) {
4 // 撰寫所需要的程式碼
5 // ...

```

```

6 // 給予其他執行緒執行的機會
7 try {
8 Thread.sleep(10);
9 } catch (InterruptedException e) {
10 // 此執行緒的 sleep
11 // 另外一個執行緒中斷
12 }
13 }
14 }
15 }

```

程式 14-1 列出如何使用 **try** 以及 **catch** 區塊。除了 **Thread.sleep()**，還有其他方法可以用來中斷執行緒一段時間。執行緒可以呼叫另外一個執行緒的 **interrupt** 方法，這會使得被暫停的執行緒會丟出 **InterruptedException** 例外。

**sleep** 是執行緒類別的靜態方法，會作用在目前正在執行的執行緒，此方法為 **Thread.sleep(x)**。**sleep** 方法的參數值是以毫秒 (**ms**) 為單位，會將執行緒轉為非作用中 (**inactive**)。**sleep** 還有幾個多載方法，像是您可以指定暫停的時間單位到奈秒 (**nano second**) 的準確度。（然而，實際上這幾個方法還是要視作業系統時間器以及排程的支援）。被暫停的執行緒除非是它接收到中斷訊號才會提早執行，否則會一直等到指定的暫停時間到了才會繼續。

## 終結執行緒

已經執行完成或被終結 (**terminated**) 的執行緒，無法再次被執行。

您可以設一個旗標 (**flag**) 來標記何時應該要離開 **run** 方法，以終結執行緒。

```

1 public class Runner implements Runnable {
2 private boolean timeToQuit=false;
3
4 public void run() {
5 while (! timeToQuit) {
6 // do work until we are told to quit
7 }
8 // clean up before run() ends
9 }
10
11 public void stopRunning() {
12 timeToQuit=true;
13 }
14 }

```

```

1 public class ThreadController {
2 private Runner r = new Runner();

```

```
3 private Thread t = new Thread(r);
4
5 public void startThread() {
6 t.start();
7 }
8
9 public void stopThread() {
10 // 使用指定的 Runner 實例
11 r.stopRunning();
12 }
13 }
```

在下列這一段程式碼，您可以使用類別 `java.lang.Thread` 的靜態方法 `currentThread`，來取得現正在執行的執行緒的參考 (reference)。例如：

```
1 public class NameRunner implements Runnable {
2 public void run() {
3 while (true) {
4 // 撰寫所需要的程式碼
5 }
6 // 列印目前執行緒的名稱
7 System.out.println("Thread " + Thread.currentThread()
8 .getName() + " completed");
9 }
10 }
```

## 執行緒的基本控制

本節描述如何控制執行緒。

### 測試執行緒

執行緒會在一個未知的狀態，可使用 **isAlive** 方法取得執行緒是否仍是存在的。此名稱 **Alive** 並不是表示執行緒正在執行，此方法回傳 **true** 的意義是該執行緒已經被啟動，但尚未完成。

### 存取執行緒的優先順序

**getPriority** 方法可用來取得執行緒的優先順序，而 **setPriority** 方法可用來設定執行緒的優先順序。**Priority** 是一個整數值。執行緒類別包含了下列常數：

```
Thread.MIN_PRIORITY = 1
Thread.NORM_PRIORITY = 5
Thread.MAX_PRIORITY = 10
```

執行緒所預設的優先順序權是 **NORM\_PRIORITY**。

### 暫時停止執行緒

有方法可以暫時停止執行緒的執行，在暫停執行緒之後，還可以再繼續執行，就像沒有發生任何事情一樣。只會感覺此執行緒的執行效能慢了點。

**Thread.sleep()** 方法

**sleep** 方法可將執行緒暫停一段時間，回想一下之前的說明，被暫停的執行緒並不一定會在 **sleep** 時間結束後就立即重新執行。這是因為某些其他的執行緒在那時正在執行，或是排程尚未安排執行。除非發生下列事件之一：

- 具有較高的優先執行權的執行緒被喚起
- 為了某些理由需要凍結 (**block**) 目前正在執行的執行緒

### join 方法

**join** 方法會使得目前正在執行的執行緒轉為等待 (**wait**)，直到呼叫 **join** 方法的執行緒結束。例如：

```
1 public static void main(String[] args) {
2 Thread t = new Thread(new Runner());
3 t.start();
4 ...
5 // Do stuff in parallel with the other thread for a while
6 ...
7 // 在此等待 timer 執行緒結束
8 try {
9 t.join();
10 } catch (InterruptedException e) {
11 // t 提早返回
12 }
13 ...
14 // 繼續執行該執行緒未完成的動作
15 ...
16 }
```

`join` 方法有好幾種的多載方法可供選擇，像是可以讓您指定執行緒等待的時間。您可以在呼叫 `join` 方法時，指定暫停的時間，單位是毫秒。例如：

```
void join(long millisec);
```

在這個範例中 `join` 方法並不是將讓正在執行的執行緒暫停一段時間，而是等到呼叫 `join` 方法的執行緒結束。

您也可以呼叫 `join` 方法時，指定暫停的時間單位到奈秒 (nano second) 的準確度。例如：

```
void join(long millisec, int nanosec);
```



---

注意 -- 雖然跟 `sleep` 方法有點類似，但這些 `join` 方法時間的準確性也會根據作業系統以及其排程機制有關。

---

跟 `sleep` 方法一樣，`join` 也是會對中斷產生反應，離開時會產生 `InterruptedException` 例外。

### Thread.yield() 方法

使用 `Thread.yield()` 方法是讓其他可執行的執行緒有機會可以執行。如果有其他的可執行的執行緒，`Yield` 會將目前正在執行的執行緒置於可執行池 (runnable pool)，讓給其他可執行的執行緒。但沒有其他可執行的執行緒，`yield` 不會有任何的效果。

**sleep** 方法是讓給低優先權的執行緒有執行的機會，而 **yield** 方法則是給另外其他的可執行緒有執行的機會。

## 另外一個建立執行緒的方法

至此，您已經看到如何使用另一個類別實現 **Runnable** 介面來建立執行緒的方法。事實上，將類別實現 **Runnable** 介面並不是唯一建立執行緒的方法，您還可以用繼承 **Thread** 類別的方式，來建立執行緒，而並不是使用實現 **Runnable** 介面的方式。

```
1 public class MyThread extends Thread {
2 public void run() {
3 while (true) {
4 // 撰寫所需要的程式碼
5 try {
6 Thread.sleep(100);
7 } catch (InterruptedException e) {
8 // 中斷 sleep
9 }
10 }
11 }
12
13 public static void main(String args[]) {
14 Thread t = new MyThread();
15 t.start();
16 }
17 }
```



## 選擇建立執行緒的方法

我們已經知道好幾個建立執行緒的方法，但您要如何決定使用哪一個呢？每一個方法都有它的優點，在本節會對此做詳細的描述。

下面是實作 **Runnable** 介面方式的優點：

從物件導向的觀點來說，執行緒類別是虛擬 **CPU** 的封裝，它應該只能被繼承，讓您更改或繼承在 **CPU** 裡的行為。由於上述理由，也為了在執行緒裡區別 **CPU**，程式以及資料，本課程將使用實作 **Runnable** 的方式。

- 因為 Java 技術只允許單一繼承，那麼您就不能再繼承其他的類別，像是 **Applet**，如果您已經繼承執行緒類別，某些情況下，這個限制會讓您只能選擇使用實現 **Runnable** 的這個方法。

- 因為有時候您會遇到一定要實現 **Runnable** 的情況，如果為了一致性的考量，你很可能久而久之就會習慣用這個方式了。

而直接繼承執行緒的優點是程式會比較簡單點。



注意 -- 儘管這兩個方式都可以用來建立執行緒，當您選擇使用繼承 **Thread** 的方式時，應該多考慮一下。您會想要繼承 **Thread**，主要應該是您想要更改以及延伸執行緒的行為，而非單純只是想實現 **run** 方法。

## 使用 `synchronized` 關鍵字

當多個執行緒共用資料時，可能會使得資料產生錯亂，例如，某一個執行緒正在更改共用資料，但在完成修改之前，另一個執行緒就讀取這份正在修改的共用資料。同步化機制 (**synchronization mechanism**) 就是為了避免產生這種類型的錯誤。接下來的章節將會更進一步的詳細的介紹這個問題。

這個章節會介紹使用 `synchronized` 關鍵字，此為 **Java** 程式語言的程式提供了一個方式，讓程式設計師可以控制執行緒共用資料。

### 問題

將一個類別想像成堆疊 (**stack**)。這個類別應該會是：

```
1 public class MyStack {
2 int idx = 0;
3 char [] data = new char[6];
4
5 public void push(char c) {
6 data[idx] = c;
7 idx++;
8 }
9
10 public char pop() {
11 idx--;
12 return data[idx];
13 }
14 }
```

此類別並不考慮處理超過存取 **stack** 範圍的問題，要注意 **stack** 容量是有限的。然而，這個問題跟要討論的主題並無相關，所以先暫不考慮此情況。

這個範例程式是透過索引 (**index**) 的遞增，遞減來決定資料存放的位置。

想像現在有兩個執行緒都擁有此類別所建立的實例參考 (**reference**)。其中一個執行緒正把資料放入堆疊，而另一個執行緒正將資料自堆疊裡移除。原則上來說，加入以及移除資料的動作都會成功的被執行，然而，這可能會導致問題。

假設執行緒 **a** 正在加入字元，而執行緒 **b** 正在移除字元。執行緒 **a** 才剛完成存放字元至堆疊，但還尚未遞增 **index** 指標。為了某些理由，執行權被別的執行緒搶走了。在此時，此物件的資料會產生不一致的狀況，如下所示：

```
buffer |p|q|r| | | |
idx = 2 ^
```

明確地來說，資料一致性的要求為：**idx** 應該為 3，要不就是還不能新增字元資料。

如果執行緒 **a** 接著繼續執行，可能還不會造成任何影響，但是假定負責移除字元的執行緒 **b** 正在等待機會執行，而執行緒 **b** 得到移除字元的執行機會了。

這會有個資料不一致的狀況在進入到 **pop** 方法，並進一步遞減索引值 (**index values**)。

```
buffer |p|q|r| | | |
idx = 1 ^
```

實際上已經導致忽略字元 **r**。在此之後，所回傳的值會是字元 **q**。至此，看起來都像是從來沒有放置字母 **r** 的樣子，所以會很難去陳述有著這麼一個問題。我們接著看原本執行緒 **a** 繼續執行的狀況。

輪到執行緒 **a** 執行 **push** 方法剩下的程式碼，接著它會遞增索引值。得到的結果如下：

```
buffer |p|q|r| | | |
idx = 2 ^
```

這個結構表示 **q** 是有效的資料，但存放 **r** 的格子應該是空的。換句話說，**q** 被讀取並被放置於 **stack** 兩次，以及字母 **r** 永遠不會出現了。

我們透過這個簡單範例來說明多個執行緒存取共同資料所會發生的狀況。您需要一個在執行緒開始存取共同資料進行特定工作前，確保共用資料一致性狀態的機制，

其中一個方法是阻止執行緒在結束程式之前切換到另一個執行緒。這個方法常用在一般低階機器語言，但它並不適合在多人使用的系統上。

在 **Java** 中是使用另一個方式，就是提供更精細的控制機制，讓執行緒在存取資料時，不用擔心是否會有其他執行緒會中途插進來執行。

## 物件鎖定旗標 (Object Lock Flag)

在 Java 技術裡，每個物件都有一個自己的旗標 (flag)。您可以把這個標記想成表示是否鎖定的旗標 (lock flag)。而關鍵字 **synchronized** 可以跟這個旗標互動，讓程式可以獨佔共用資料。修改下列程式片段：

```
public class MyStack {
 ...
 public void push(char c) {
 synchronized(this) {
 data[idx] = c;
 idx++;
 }
 }
 ...
}
```

當執行緒執行到 **synchronized** 敘述時，將物件做為參數傳遞給 **synchronized** 敘述，檢查物件的鎖定旗標，並在繼續下一個動作之前試著去取得物件的鎖定旗標（見圖 14-4）..

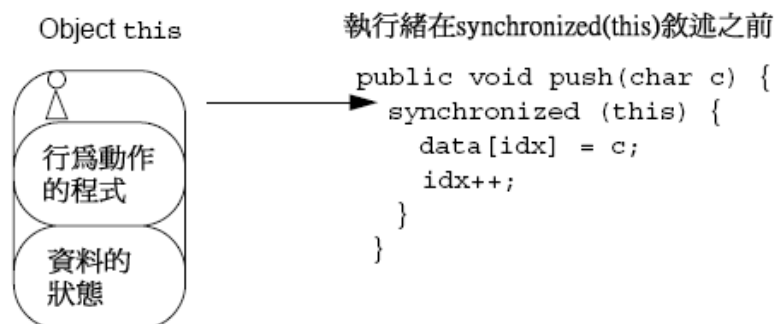


圖 14-4 在執行緒之前使用 synchronized 敘述

圖 14-5 是在執行緒之後使用 **synchronized** 敘述的範例，.

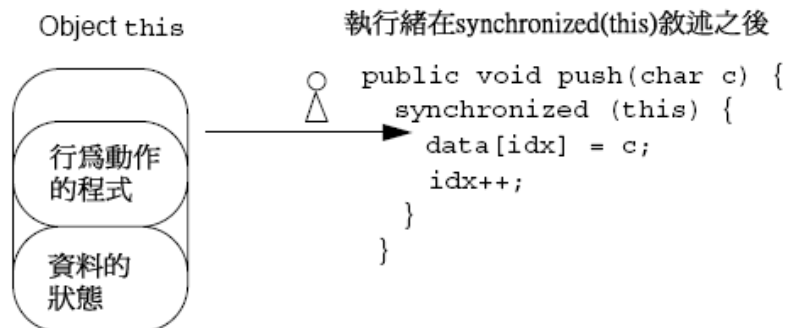


圖 14-5 於執行緒之後使用 synchronized 敘述

您應該會發現，即使如此還是無法徹底保護資料。如果 `pop` 方法並沒有使用 `synchronized` 保護共用資料，而 `pop` 方法被另一個執行緒所執行，仍會有因不一致存取資料的行為造成損毀資料的風險。所以所有存取共用資料的方法都必須使用 `synchronize` 來保護共用資料的存取，才能達到保護資料一致性的目的。

圖 14-6 說明 `pop` 方法使用 `synchronized`，讓執行的執行緒拿到物件的鎖定旗標，而同時若有另一個執行緒試著執行物件的 `pop` 方法時所會發生的狀況。

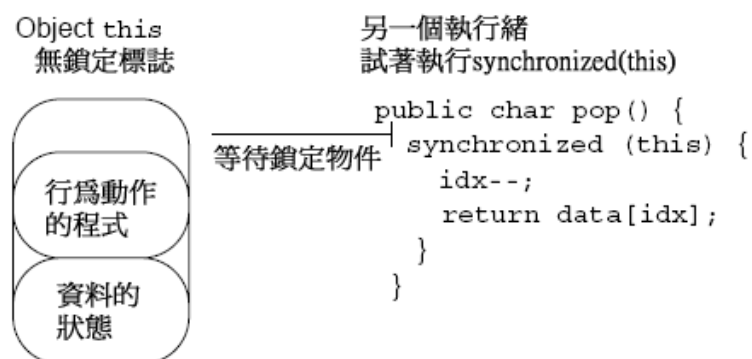


圖 14-6 執行緒試著執行同步化區塊

當執行緒試著執行 `synchronized(this)` 敘述，它會試著取得物件的鎖定旗標。但因為得不到旗標，於是執行緒無法繼續執行。此執行緒轉等待池 (waiting pool)，等待物件的鎖定的解除。當旗標回到物件時，等著取得旗標的執行緒，在取得旗標後就可繼續執行。

## 釋放鎖定旗標

執行緒若沒取得物件鎖定旗標，就得等到取得旗標後才能繼續執行。因此，當持有旗標的執行緒不再需要旗標時，就要歸還旗標，這點相當重要。

鎖定旗標會被自動送返所屬物件，當持有鎖定旗標的執行緒，完成 `synchronized` 區塊的執行時，就會釋放鎖定旗標。Java 會讓鎖定 (lock) 自動返回所屬的物件，即使在 `synchronized` 區塊裡遇到例外狀況，`break` 敘述，或是回傳敘述改變程式執行程序。同理，如果執行緒在執行巢狀區塊的程式對同一個物件使用 `synchronized`，會在離開最外層的區塊時，才會釋放物件的標記，而不是在最裡層的區塊。

比起其他系統，這些規定會使得開發人員在使用 `synchronized` 區塊時方便的多。

## 使用 `synchronized` 機制

只要所有試著存取 `synchronized` 區塊所保護的資料，`synchronized` 機制就會發揮作用。

您應該將需要保護的資料用 `synchronized` 區塊標示起來，就像私有的一樣。如果您不願做此設定，那麼原本需要保護的資料就可以被外部類別定義的程式碼所存取；像這樣的狀況會使其它的開發人員忽略您的保護，並可能導致資料在執行時期被破壞。

如果 `synchronized` 區塊包含了整個方法所有敘述，則也可將 `synchronized` 關鍵字直接置於方法的最前面。所以下列的兩個程式碼片斷的效果是一樣的：

```
:public void push(char c) {
 synchronized(this) {
 // push 方法的程式碼
 }
}

public synchronized void push(char c) {
 // push 方法的程式碼
}
```

何時要使用上述的那一種寫法比較好？

如果您在方法宣告使用 `synchronized` 修飾字，那麼整個方法都會變成同步化區塊，那會導致鎖定旗標的時間會比原本所需要的時間久。

然而，這麼做卻可以讓使用者得知方法是否具備 `synchronized` 的特性。而且使用 `javadoc` 工具所產生的文件，也會明列方法是否為 `synchronized`。這類資訊對避免造成死結 (`deadlock`) 是很重要的資訊。（我們將會在下一節提到 `deadlock`）。`Javadoc` 文件產生器會將 `synchronized` 修飾字置於文件檔案裡，但是 `synchronized(this)` 則不會有此效果，您只能在方法區塊內找尋它。

## 執行緒狀態

`synchronization` 是一個特別的執行緒狀態。圖 14-7 說明新的執行緒狀態轉換圖

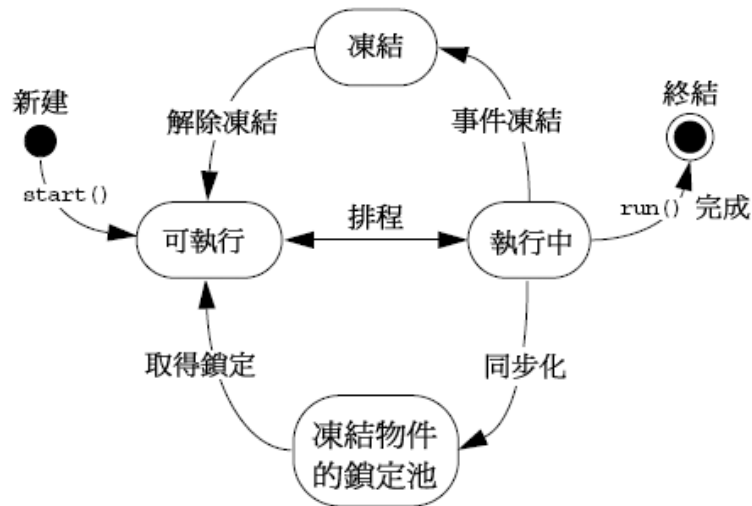


圖 14-7 使用同步化機制後的執行緒狀態圖

## 死結 (Deadlock)

在程式有多個執行緒相互競爭取獲得存取資料的權利。在這樣的狀況下，有可能會發生死結 (deadlock)。當 A 執行緒正等待被 B 執行緒鎖定的資料，但 B 執行緒也在等待被 A 執行緒鎖定的資料。在這個狀況下，這兩個執行緒都在互相等待另一個執行緒解除在 `synchronized` 區塊的鎖定。但因為這兩個執行緒都無法繼續進行，所以也無法結束區塊以解除鎖定。

Java 無法偵測也無法試圖避免這個狀況。確保避免發生死結是開發人員的責任。避免死結的經驗法則是：如果您用 `synchronized` 取得多個物件的鎖定，需對鎖定物件的次序做全面性的規定，並在程式裡遵循鎖定的次序，再依相反的順序釋放鎖定的物件。



## 執行緒互動

通常不同的執行緒所負責執行的任務，之間是沒有關連的。然而，有時執行的任務之間，會需要以某些方式互動，這必須會透過程式來進行任務之間的溝通。

### 情境

把您自己和計程車司機想成兩個執行緒。您需要計程車載您到目的地，而計程車司機則是要載運乘客賺取車費。所以兩個人都有自己想要達成的任務。

### 問題

您在搭乘計程車時可能想舒服的休息一下，直到司機通知您到達目的地。但如果您需要每隔兩秒就得問司機“到達目的地了沒？”，對您和司機來說都會很煩。在等待下一位乘客時，司機會想在車內打個盹，直到有乘客要搭乘計程車。但司機並不需要在休息時，每隔 5 分鐘就起來看有沒有乘客。所以，我們希望這兩個執行緒在完成工作後，可以儘可能抽空休息一下。

### 解決方案

您以及司機彼此之間需要一些溝通的方式。當您正走向計程車站牌，而司機正在車子內休息。當您告訴司機想要搭乘計程車，司機就會醒來，並開始駕駛，那您就可以開始休息。在到達目的地後，司機會通知您下車。司機就會再度等待以及打個小盹直到下個客戶光臨。

### wait 以及 notify 方法

`java.lang.Object` 類別針對執行緒之間的溝通，提供了兩個方法：**wait** 以及 **notify**。如果某一個執行緒在一個約定好的物件 `x`，呼叫 **wait** 方法。那麼此執行緒會暫停執行，直到另一個執行緒在同一個物件 `x` 上呼叫 **notify** 方法。

在之前的情節中，當司機在車子裡等待，就是計程車司機執行緒呼叫 `cab.wait()`，而當您需要使用計程車，那麼就是你這個執行緒執行 `cab.notify()`。

執行緒必須要獲得物件的鎖定，才能呼叫此物件的 **wait** 方法或是 **notify** 方法。換句話說，必須是 **synchronized** 所作用的物件，才能呼叫此物件的 **wait** 以及 **notify** 方法。在這個範例，您要在一開始就指出 **synchronized(cab)**，才允許呼叫 **cab.wait()** 或 **cab.notify()**。

### Pool Story

當一個執行緒執行的 **synchronized** 區塊程式碼中，呼叫鎖定物件的 **wait** 方法，那麼此執行緒會為了這個物件，被放入等待池 (**wait pool**)。此外，執行緒一旦呼叫物件的 **wait** 方法，會自動地釋放該物件的鎖定標記。有幾個不同的 **wait** 方法可供呼叫。

```
wait()
wait(long timeout)
wait(long timeout, int nanos)
```

若呼叫鎖定物件的 **notify** 方法，會將執行緒會從物件的等待池移到鎖定池 (**lock pool**)，執行緒會在此等待，直到物件鎖定標記被釋放。**notifyAll** 方法會將等待取得此物件鎖定標記的所有執行緒，從等待池移出，並進入鎖定池。只有在鎖定池的其中一個執行緒可以獲得該物件的鎖定標記，而這個執行緒可以繼續執行，自呼叫 **wait()** 方法之後的程式。

許多的系統都實作了 **wait-notify** 機制，執行緒會喚起等待最久的執行緒。然而，**java** 並不保證這種運作方式。

當您使用 **notify** 方法時，不用在意是否有其他正在等待的執行緒。如果在執行某個物件的 **notify** 方法時，並沒有執行緒凍結在等待池等待取得物件的鎖定標記，是不會有任何作用的。

## 執行緒狀態

等待池 (wait pool) 也是個特別的執行緒狀態。圖 14-8 說明最完整的執行緒狀態轉換圖。

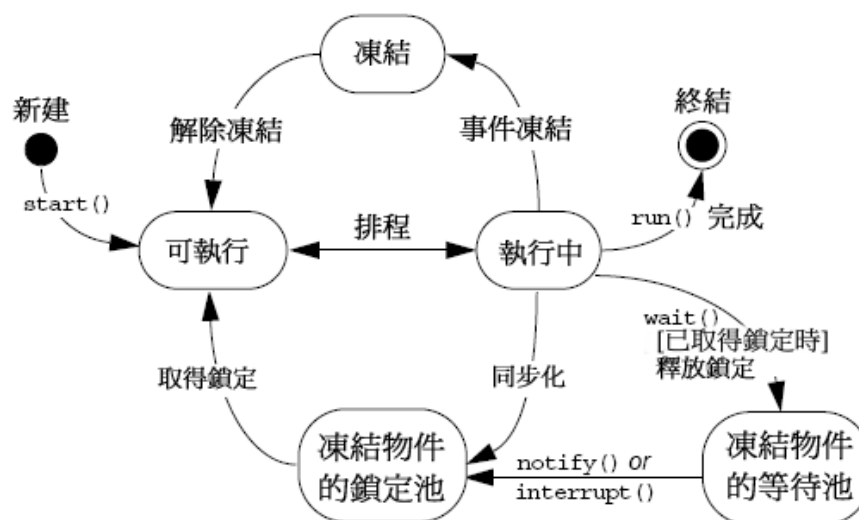


圖 14-8 使用 wait 以及 notify 的執行緒狀態圖

## 同步監控 (Monitor) 模型

試想兩個執行緒之間若需要存取共同資料，會讓情況變的更複雜。您必須要確保執行緒並不會讓共用資料發生不一致的狀態，因為可能會有另外的執行緒想存取這些資料。您也必須確保您的程式不會產生死結 (**deadlock**)，若有其他執行緒正在等待鎖定資料，執行緒並不會適時的釋放鎖定。

在計程車範例中，程式會根據約定好的物件 - 計程車，對這個物件執行 **wait** 以及 **notify** 方法。如果有人打算搭乘的是巴士，您需要 **notify** 另一個 **bus** 物件。要記得所有在同一個等待池的執行緒必須要能接受等待池控制物件的通知。絕對不要撰寫程式碼去干涉此行為。

## 綜合應用

本節將介紹一個執行緒互動的範例，並展示使用 `wait` 以及 `notify` 方法來解決傳統的生產者與消費者問題 (producer-consumer problem)。

我們首先介紹 `stack` 物件的架構，以及執行緒存取 `stack` 物件的細部程式。然後查看 `stack` 的細部程式，以及用來保護 `stack` 的資料的機制，根據 `stack` 的狀態實現執行緒的溝通方式。

此範例的類別名稱是 `SyncStack`，為了核心類別 `java.util.Stack` 做區別，此類別提供下列公開的 API：

```
public synchronized void push(char c);
public synchronized char pop();
```

## Producer 執行緒

**producer** 執行緒會產生新的字元，並放置於堆疊 (**stack**)。

程式 14-2 列出 **producer** 類別

```
1 package mod14;
2 public class Producer implements Runnable {
3 private SyncStack theStack;
4 private int num;
5 private static int counter = 1;
6
7 public Producer (SyncStack s) {
8 theStack = s;
9 num = counter++;
10 }
11
12 public void run() {
13 char c;
14 for (int i = 0; i < 200; i++) {
15 c = (char) (Math.random() * 26 + 'A');
16 theStack.push(c);
17 System.out.println("Producer" + num + ": " + c);
18 try {
19 Thread.sleep((int) (Math.random() * 300));
20 } catch (InterruptedException e) {
21 // 省略
22 }
23 }
24 } // run 方法結束
25 } // Producer 類別結束
```

這個範例產生 200 個隨機的大寫字元，並逐一將它們放入至堆疊 (**stack**)，每一次產生資料的時間間隔會是 0 至 300 毫秒，用隨機的方式決定。每一次放入 **stack** 的字元都會顯示至主控台上。

## Consumer 執行緒

**consumer** 執行緒會將字元從堆疊 (**stack**) 移除。

## 程式 14-3 列出 Consumer 類別

## 程式 14-3 Consumer 類別

```

1 package mod14;
2
3 public class Consumer implements Runnable {
4 private SyncStack theStack;
5 private int num;
6 private static int counter = 1;
7
8 public Consumer (SyncStack s) {
9 theStack = s;
10 num = counter++;
11 }
12
13 public void run() {
14 char c;
15 for (int i = 0; i < 200; i++) {
16 c = theStack.pop();
17 System.out.println("Consumer" + num + ": " + c);
18 try {
19 Thread.sleep((int)(Math.random() * 300));
20 } catch (InterruptedException e) {
21 // ignore it
22 }
23 }
24 } // run 方法結束
25
26 } //Consumer 類別結束

```

這個範例將收集的 200 個字元，逐一將它們移出堆疊 (**stack**)，每一次移出動作的時間間隔是 0 至 300 毫秒，用隨機的方式決定。每一次移出 **stack** 的字元都會顯示至主控台上。

現在來考慮 **stack** 類別的建構子。您可以建立沒有容量限制的 **stack** 物件，像是 **ArrayList** 類別。在這個設計裡，您的執行緒只需要依據 **stack** 是否為空來進行溝通。

## SyncStack 類別

新建立 **SyncStack** 物件的緩衝區 (**buffer**) 應該是空的。您可以使用下列程式來建構您的類別。

```
public class SyncStack {

 private List<Character> buffer
 = new ArrayList<Character>(400);

 public synchronized char pop() {
 //pop 程式在此實作...
 }

 public synchronized void push(char c) {
 // push 程式在此實作...
 }
}
```

此範例並沒有建構子，但在設計時，加入建構子會是比較好的設計，省略它只是為了讓程式更為精簡。

### pop 方法

現在來看 **push** 以及 **pop** 方法，它們必須要設定為同步化才能保護共同的資料 (**shared buffer**)。除此之外，如果在執行 **pop** 方法時，發現 **stack** 是空的，那麼必須將正在執行的執行緒轉為等待狀態；若在 **pop** 方法的 **stack** 不再是空的，就可以通知正在等待的執行緒。程式 14-4 列出 **pop** 方法的程式碼。

程式 14-4 **pop** 方法

```
1 public synchronized char pop() {
2 char c;
3 while (buffer.size() == 0) {
4 try {
5 this.wait();
6 } catch (InterruptedException e) {
7 // ignore it...
8 }
9 }
10 c = buffer.remove(buffer.size()-1);
11 return c;
12 }
```



當 **stack** 是空的時，就不會有資料從 **stack** 物件移除，所以執行緒必須等到 **stack** 不再是空的，才能試著從 **stack** 移出資料。

要把 **wait** 方法放在 **try-catch** 區塊裡，因為中斷呼叫 (**interrupt call**)，會使得執行緒結束等待。在這個範例中，也是必須要在迴圈裡呼叫 **wait**。如果為了某些理由（像是中斷）喚起執行緒，但發現 **stack** 仍舊是空的，那麼執行緒必須再次進入等待狀況。

**pop** 方法為了 **stack**，而使用宣告 **synchronized**，這是為了兩個理由。首先，從 **stack** 物件移除字元，會影響到共同的資料緩衝區。第二個，必須在 **synchronized** 區塊裡，才能呼叫 **this.wait()** 方法，**stack** 物件在這裡用 **this** 表示。

**push** 方法使用 **this.notify()**，從 **stack** 物件的等待池釋放執行緒。在釋放執行緒之後，它可能可以取得 **stack** 的鎖定以及繼續執行 **pop** 方法，再從 **stack** 的緩衝區 (**buffer**) 移除字元。



注意 -- 在 **pop** 方法裡，**wait** 方法會在從 **stack** 移出字元之前被呼叫。這是因為要有資料，才能繼續執行移除的動作。

您應該也要考慮錯誤檢查。您可能會注意到沒有防止 **stack** 超出範圍的程式碼。這並不是必要的，因為只有一個方法可以從 **stack** 移除字元，那就是要經由 **pop** 方法，以及這個方法會發現 **stack** 是空的時候，讓執行緒進入等待狀態。所以，這個錯誤檢查並不是必要的。

## push 方法

**push** 方法和 **pop** 方法是類似的，也是會對共用資料造成影響，所以也要宣告 **synchronized**。除此之外，因為 **push** 方法將一個字元加入緩衝區，所以它要負責通知正在等待不再是空的 **stack** 的執行緒。

### 程式 14-5 展示 push 方法

#### 程式 14-5 push 方法

```

1 public synchronized void push(char c) {
2 this.notify();
3 buffer.add(c);
4 }
5 }
6

```

呼叫 `this.notify()` 方法，所釋放的執行緒，是因為空的 `stack` 而呼叫 `wait` 方法轉為等待。在呼叫 `notify` 之前，共用的資料還沒有被更動資料。要離開 `synchronized` 區塊，才能釋放 `stack` 物件的鎖定，所以當 `push` 方法正改變 `stack` 資料時，這個執行緒也正在等待獲得資料的鎖定。

將所有的程式片斷組合起來，程式 14-6 列出完成的 `SyncStack` 類別的程式碼。

程式 14-6 `SyncStack` 類別

```
1 package mod13;
2
3 import java.util.*;
4
5 public class SyncStack {
6 private List<Character> buffer
7 = new ArrayList<Character>(400);
8
9 public synchronized char pop() {
10 char c;
11 while (buffer.size() == 0) {
12 try {
13 this.wait();
14 } catch (InterruptedException e) {
15 // ignore it...
16 }
17 }
18 c = buffer.remove(buffer.size()-1);
19 return c;
20 }
21
22 public synchronized void push(char c) {
23 this.notify();
24 buffer.add(c);
25 }
26 }
27
28
29
30
31
```

## SyncTest 範例

您必須將 **producer**、**consumer** 以及 **stack** 的程式設計成完整的類別。測試程式會將這些類別集中起來測試。特別注意到 **SyncTest** 只有建立一個 **stack** 物件，並將其給所有的執行緒所共用。程式 14-7 列出 **SyncTest** 類別的程式碼。

程式 14-7 SyncTest 類別

```

1 package mod14;
2
3 public class SyncTest {
4 public static void main(String[] args) {
5 SyncStack stack = new SyncStack();
6 Producer p1 = new Producer(stack);
7 Thread prodT1 = new Thread (p1);
8 prodT1.start();
9 Producer p2 = new Producer(stack);
10 Thread prodT2 = new Thread (p2);
11 prodT2.start();
12 Consumer c1 = new Consumer(stack);
13 Thread consT1 = new Thread (c1);
14 consT1.start();
15 Consumer c2 = new Consumer(stack);
16 Thread consT2 = new Thread (c2);
17 consT2.start();
18 }
19 }
```

下列是執行 **mod14.SyncTest** 程式的部份輸出結果。請注意每一次執行緒程式執行結果都不一定會相同。

```

Producer2: F
Consumer1: F
Producer2: K
Consumer2: K
Producer2: T
Producer1: N
Producer1: V
Consumer2: V
Consumer1: N
Producer2: V
Producer2: U
Consumer2: U
Consumer2: V
Producer1: F
```

Consumer1: F  
Producer2: M  
Consumer2: M  
Consumer2: T

# 應用 Swing API 建立 Java 圖形使用者介面

---

## 目標

完成這個單元後，您將能夠：

- 描述 JFC Swing 技術
- 知道什麼是 Swing
- 認得 Swing 套件
- 描述 GUI 的基礎構件：程式容器 (container)、元件 (components)、及版面配置管理程式 (layout manager)
- 檢視程式容器的頂層 (top-level) 屬性、多用途屬性、及特殊用途屬性
- 檢視元件
- 檢視版面配置管理程式
- 描述 Swing 的單一執行緒 (single-threaded) 模型
- 運用 Swing 元件建立 GUI

## 其他資源



其他資源 — 對於本模組中敘述之主題，提供以下額外資訊做為參考：

欲運用 JFC / Swing 技術所有的功能，需要更多的練習與研究。進一步學習 JFC / Swing 技術的參考資料如下：

- 在免費的 Java 教學簡介裡內含 JFC / Swing 技術教學簡介，它位於以下 URL：  
<http://java.sun.com/docs/books/tutorial/>
- SwingSet 範例程式碼
- API 說明文件  
從 javax.swing 套件開始學習起，並順著需要的部份，學習其他子套件。
- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O' Reilly Media. 2005.

## 何謂 Java 基礎類別 (Java Foundation Class, JFC)?

JFC，或稱 Java 基礎類別，是一組圖形使用者介面支援套件，起先是 Java SE 平臺的一部份，之後則成為 JDK 的核心 API。因此，JFC 及 JDK 核心 API 間的界限變得沒有那麼明顯，不過在本課程中，JFC 包含以下特色：

- **Swing 元件集** —— **Swing** 是一組加強過的元件集，為原本 AWT 中的元件提供替換元件，及許多更進階之元件。這些元件使您能夠很容易地建立具有許多最新 GUI 應用程式所包含功能的使用者介面。這類元件包括：樹狀結構、表格、進階的文字編輯器、及可拖曳工具列。
- **2D 圖形** —— 運用 **Java 2D API**，你可以執行進階的繪圖功能、複雜的色彩操作、成形及變形（旋轉、切變 (shear)、伸縮…等等）處理、以及將文字當成圖形來處理。本課程中不討論 2D 圖形。
- **可插接式 look-and-feel**。這項特色提供 **Swing** 元件選擇其觀感。同一份程式可以被描繪為 **Microsoft Windows**、**Motif**、與金屬等不同的外觀形式。
- **易用性** —— 政府機構正逐漸要求各部門的電腦程式能輕易的讓殘障人士使用。**Swing** 元件集藉由 **Accessibility API** 幫助此類程式的開發。它提供介面給相關的技術，如：螢幕閱讀器、螢幕放大鏡、可聽的文字朗讀程式（語音處理）等等。
- **拖放 (Drag-and-drop)** —— 以 GUI 為基礎的資料轉移，可在同一個程式的元件中或界於不同程式使用，數年來已成為現代 GUI 系統的特色。就使用者而言，此類型的轉移，可分為兩種形式：剪下與貼上及拖放。**JDK** 可支援這兩類資料轉移工具。
- **國際化** —— **JFC** 類別支援不同的字元集，如：日文、中文、韓文等。藉此讓開發者建立的程式能以不同的語言跟使用者互動。
-

## 何謂 Swing ?

**Swing** 是一組加強過的元件集，為原本 **AWT** 中的元件提供替換元件及許多更進階的元件。這些元件使您能夠透過現代應用程式預設就有的功能，建立使用者介面。包括：樹狀結構、表格、進階的文字編輯器、及可拖曳 (**tear-off**) 的工具列。

**Swing** 尚有一些特別的功能。例如，透過 **Swing** 撰寫程式時，能夠讓它採用各種主機平臺特有的 **look-and-feel**，或是使用特別為 **Java** 程式語言設計的 (**Metal**)**look-and-feel**。事實上，您也可以從頭開始建立你自己的 **look-and-feel**、或是修改現成的 **look-and-feel**，然後插接到您的程式裡。**look-and-feel** 可以在程式中寫定、或是讓使用者和系統管理者在執行時期自行選擇。



---

註釋 --**look-and-feel** 一詞在本模組中會經常出現。**look** 指的是元件的外觀，而 **feel** 指的是元件回應使用者動作（例如滑鼠點擊）的方式。設計新的 **look-and-feel**，意指撰寫所需類別來定義新的外觀及輸入行為。至於如何設計 **look-and-feel**，並不在本單元的範圍之內。

---



## 可插接式 look-and-feel

可插接式 **look-and-feel**，讓開發者建立的 **Java** 程式，在任何平臺上執行，外觀的呈現就如同該平臺開發一般程式一樣。當程式在 **Microsoft Windows** 環境下執行時，就如同它是專門為此 **Windows** 環境而開發；且同一個程式在 **UNIX** 平臺上，也如同它是為 **UNIX** 環境而開發。

開發者可以運用任何他們選用的 **look-and-feel** 類型，建立自訂 **Swing** 元件。這樣可以增強應用程式與跨平臺部署之 **applets** 的一致性。整個應用程式的 **GUI** 可以在執行時期從某一種 **look-and-feel**，切換成另一種完全不同的 **look-and-feel**。

由 **Swing** 元件提供的可插接式 **look-and-feel**，也受惠於 **Swing** 元件的基本架構。下一節將說明 **Swing** 架構並解釋它如何幫助可插接式 **look-and-feel**。

## Swing 架構

Swing 元件是基於模型—視圖—控制程式 (Model-View-Controller“MVC”) 之架構所設計。Swing 架構雖沒有嚴格的遵循 MVC 架構，但其根源仍然來自 MVC。

### 模型—視圖—控制程式之架構

按照 MVC 架構，一個元件可以用三個獨立的部份來塑造。圖 15-1 顯示 MVC 之架構。

- 模型 —— 模型用來定義元件的資料。
- 視圖 —— 視圖代表元件視覺所顯示的樣貌。此樣貌受模型內的資料控制。
- 控制程式 —— 當使用者與元件互動時，控制程式處理元件的反應。這些反應可能包括模型與視圖的更動。

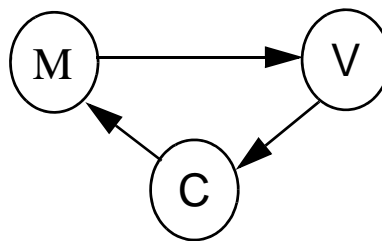


圖 15-1 MVC 架構

理論上，架構中的這三種類型（模型、視圖、控制程式），應該用三種不同的類別類型來表達。但礙於視圖與控制程式間種種的相依性，使得付諸執行有其困難。控制程式的角色與視圖如何實作，有極大的相關性，因為使用者是與視圖進行互動。換句話說，若不顧及視圖如何實作，而獨立撰寫一個通用的控制程式，是很困難的。有關這方面的問題，我們會在下一節詳加討論。

### 可分離的模型架構

Swing 元件採用可分離的模型架構。在此架構下，視圖與控制程式合併成單一的複合物件，因為他們彼此間有緊密的相依性。模型物件則同 MVC 架構一般，被當作是一個獨立的物件。15-7 頁圖 15-2 顯示此可分離的模型架構。

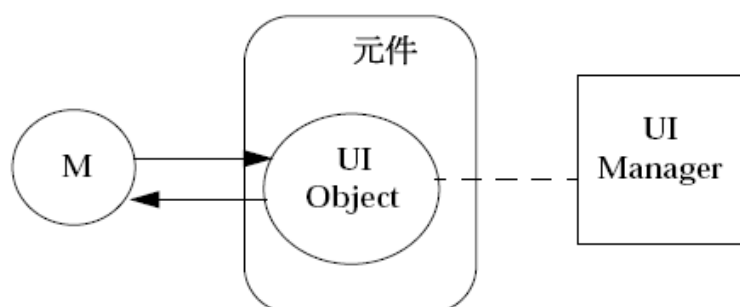


圖 15-2 可分離的模型架構

圖 15-2 裡的 UI 物件稱作 UI 代理 (UI delegate)。此架構下，就有可能將元件的少部份任務，委派給一個獨立的 UI 物件。這件事解釋了可插接式 look-and-feel 如何從 Swing 元件架構受惠。元件把顯示視覺外觀的任務交給獨立的 UI 物件。所以所安裝的 look-and-feel 內含的 UI 物件，會負責處理元件 look-and-feel 的呈現。

## Swing 套件

Swing API 擁有豐富且方便的套件組，使得它功能強大且具彈性。表 15-1 條列出各套件的名稱及其用途。

表 15-1 Swing 套件

| 套件名稱                                  | 用途                                                                                                          |
|---------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <code>javax.swing</code>              | 供應一組「輕型」的元件，例如： <b>JButton</b> 、 <b>JFrame</b> 、 <b>JCheckBox</b> 、及許多其他的元件                                 |
| <code>javax.swing.border</code>       | 供應繪製特定邊框的類別與介面，例如：斜邊 ( <b>bevel</b> )、蝕刻 ( <b>etched</b> )、線狀 ( <b>line</b> )、冰銅狀 ( <b>matte</b> )、及許多其他的邊框 |
| <code>javax.swing.event</code>        | 為 Swing 元件發出的事件提供支援                                                                                         |
| <code>javax.swing.undo</code>         | 允許開發者在應用程式（如文字編輯器）中，支援「取消 / 重做 ( <b>undo/redo</b> )」功能                                                      |
| <code>javax.swing.colorchooser</code> | 含有 <b>JColorChooser</b> 元件會用到的類別與介面                                                                         |
| <code>javax.swing.filechooser</code>  | 含有 <b>JFileChooser</b> 元件會用到的類別與介面                                                                          |
| <code>javax.swing.table</code>        | 提供用來處理 <b>JTable</b> 的類別與介面                                                                                 |
| <code>javax.swing.tree</code>         | 提供用來處理 <b>JTree</b> 的類別與介面                                                                                  |
| <code>javax.swing.plaf</code>         | 供應一個介面與許多抽象化類別， <b>Swing</b> 使用它們來提供可插接式觀感之功能                                                               |
| <code>javax.swing.plaf.basic</code>   | 提供依照「 <b>Basic</b> 」觀感設計的使用者介面物件                                                                            |
| <code>javax.swing.plaf.metal</code>   | 提供依照「 <b>Java</b> 」觀感設計的使用者介面物件                                                                             |
| <code>javax.swing.plaf.multi</code>   | 提供合併兩個以上觀感的使用者介面物件                                                                                          |
| <code>javax.swing.plaf.synth</code>   | 為可更換外層 ( <b>skinnable</b> ) 的觀感（其繪製工作均委派出去），提供使用者介面物件                                                       |

表 15-1 Swing 套件

| 套件名稱                                      | 用途                                                   |
|-------------------------------------------|------------------------------------------------------|
| <code>javax.swing.text</code>             | 供應類別與介面，處理可編輯及不可編輯的文字元件                              |
| <code>javax.swing.text.html</code>        | 為建立 HTML 文字編輯器，提供 <code>HTMLToolkit</code> 類別及其他支援類別 |
| <code>javax.swing.text.html.parser</code> | 為建立 HTML 文字編輯器，提供 <code>HTMLToolkit</code> 類別及其他支援類別 |

## 檢視 Java 技術 GUI 的構成

以下要素構成以 **Swing API** 為基礎的 **GUI**。

- 程式容器

程式容器位居 **GUI 層次結構 (containment hierarchy)** 的頂端。**GUI** 內所有元件都是加在這些程式容器上。**JFrame**、**JDialog**、**JWindow**、及 **JApplet** 均是頂層的程式容器。

- 元件

所有 **GUI** 元件都是從 **JComponent** 類別衍生 (derived) 出來的，例如：**JComboBox**、**JAbstractButton**、及 **JTextComponent**。

- 版面配置管理程式

版面配置管理程式負責展示程式容器裡的元件。**BorderLayout**、**FlowLayout**、**GridLayout** 即是一些版面配置管理程式的例子。尚有許多更精密、複雜的版面配置管理程式，用更多方法來控制 **GUI**。

圖 15-3 示意如何運用元件、程式容器、及版面配置管理程式，構成一個 **Swing** 使用者介面的範例。

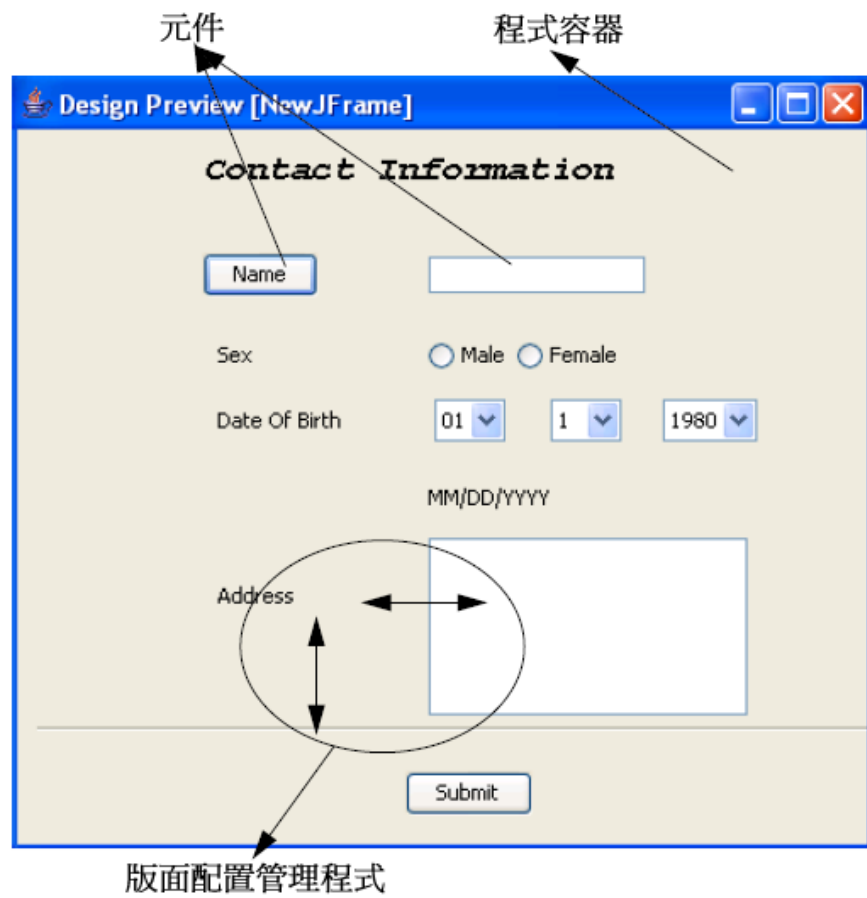


圖 15-3 GUI 之構成

## Swing 程式容器

Swing 程式容器可分為三種主要類型。

- 頂層 (top-level) 程式容器
- 多用途 (general-purpose) 程式容器
- 特殊用途 (special-purpose) 程式容器

### 頂層程式容器

頂層程式容器位居 Swing 層次結構 (containment hierarchy) 的頂端。共有三個頂層 Swing 程式容器：JFrame、JWindow、及 JDialog。此外尚有一個特殊的類別：JApplet，嚴格說來它並不是頂層程式容器，但仍值得在此一提，因為任何使用 Swing 元件的 applet 均應以它作為頂層程式容器。圖 15-4 是頂層程式容器的繼承 (inheritance) 階層架構。

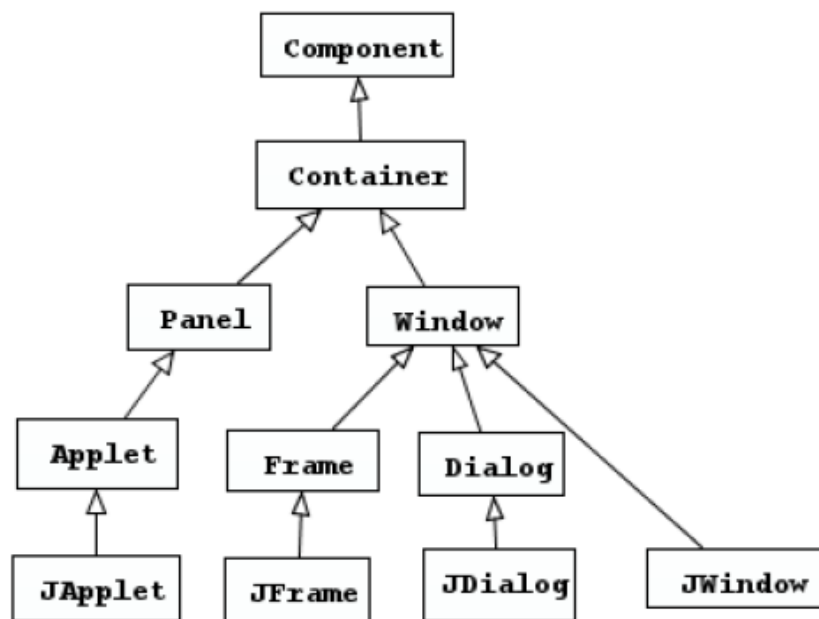


圖 15-4 頂層程式容器階層架構

JApplet、JFrame、JDialog、與 JWindow 類別分別從 Applet、Frame、Dialog、與 Window 類別直接衍生出來。這是要注意的重點，因為除此之外的所有其他 Swing 程式容器與元件，都是從 JComponents 衍生出來的。



表 15-2 含有對各個頂層程式容器的簡介。

表 15-2 頂層程式容器

| 程式容器                                                                                                                                  | 使用者介面                                                                                |
|---------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <p>框架 (Frame) 是用於大部份 GUI 應用程式中，最基本的視窗。它具有邊框與標題列。其他的元件可以加入框架之中。你亦可在框架中繪圖。框架裡還能加入選單。用來建立框架的是 <code>javax.swing.JFrame</code> 類別。</p>    |    |
| <p><code>JDialog</code> 用於建立對話框視窗。API 提供數種不同版本的建構子，來定義對話框。對話框是依附於框架的。對話框可用在向使用者取得輸入資料與確認所有關鍵動作；還可用於對使用者顯示警告訊息、錯誤訊息、問題、及資訊。</p>        |   |
| <p><code>JWindow</code> 程式容器類似 <code>JFrame</code>，但它少了邊框與標題列。沒有視窗的管理服務。</p>                                                          |  |
| <p><code>JApplet</code> 程式容器用於建立在網路瀏覽器上的 UI。通常 <code>JApplet</code> 會嵌在網頁內並可以播放動畫。其他元件與選單都能加入此程式容器。你也可以在 <code>applet</code> 內繪圖。</p> |  |

## 多用途程式容器

多用途程式容器是中介的程式容器，可靈活用於許多情況，例如：**JPanel**、**JScrollPane**、**JToolBar**、**JSplitPane**、及 **JTabbedPane**。所有這些元件都延伸 (extend) 自 **JComponent**。表 15-3 含有對各個多用途程式容器的簡介。

表 15-3 多用途程式容器

| 程式容器                                                                                                                                                                                                | 使用者介面                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <p>面板 (<b>Panel</b>) 是可以加入許多元件的程式容器。它也提供可以繪圖的表面。異於 <b>JFrame</b> 之處，在於它不是頂層程式容器。面板必須被放在頂層程式容器裡。用來建立面板的是 <b>javax.swing.JPanel</b> 類別。它延伸 <b>JComponent</b> 而非 <b>java.awt.Panel</b>。</p>            |    |
| <p>捲軸式窗格 (<b>scrollpane</b>) 在空間受限的時候相當好用。它用於顯示大型的元件或圖像。捲軸式窗格具有兩條捲軸列 (<b>scrollbar</b>)、一個列標頭 (<b>row header</b>)、以及一個行標頭 (<b>column header</b>)。用來建立捲軸式窗格的是 <b>javax.swing.JScrollPane</b> 類別。</p> |  |
| <p>工具列 (<b>toolbar</b>) 是一組有圖示的按鈕，方便存取常用功能。可將它們視為選單內動作的捷徑。用來建立工具列的是 <b>javax.swing.JToolBar</b> 類別。</p>                                                                                             |  |

表 15-3 多用途程式容器

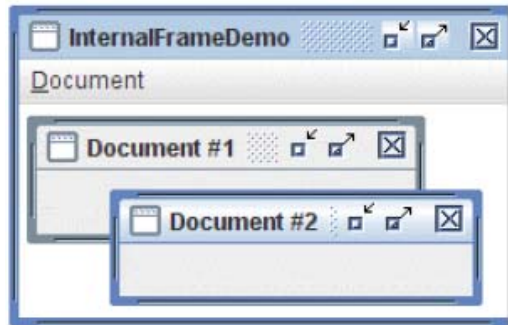
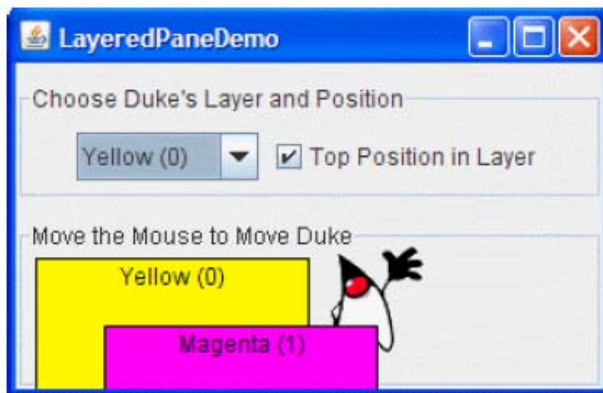
| 程式容器                                                                                                                                      | 使用者介面                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| 分割式窗格 (splitpane) 顯示兩個以上的元件，用分隔器 (divider) 隔開。元件可以左右並列顯示，或是上下並排顯示。各元件佔用的空間可以藉拖曳分隔器來調整。用來建立分割式窗格的是 <code>javax.swing.JSplitPane</code> 類別。 |  |
| 標籤式窗格 (tabbedpane) 在空間有限時也十分有用。數個標籤可共享同一份空間。但同時只能顯示其中一個標籤的內容。使用者必須選取欲顯示之標籤。用來建立標籤式窗格的是 <code>javax.swing.JTabbedPane</code> 類別。           |  |

## 特殊用途程式容器

特殊用途程式容器的例子有 `JInternalFrame` 與 `JLayeredPane`。它們在使用者介面上扮演特定的角色。內部框架 (internal frame) 被設計為在桌面窗格 (desktop pane) 內作業。它們不是像 `JFrame` 那樣的頂層程式容器。`JLayeredPane` 程式容器協助指出元件的深度 (depth)，當元件重疊時對繪製 GUI 相當有用。

表 15-4 含有對特殊用途程式容器的簡介。

表 15-4 特殊用途程式容器

| 程式容器                                                                                                                                                                                                                                                                                       | 使用者介面                                                                               |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
| 內部框架不是頂層程式容器。但它們有類似 <code>JFrame</code> 的特色，如：拖曳、改變大小、圖示化 (iconfying)、及最大化。用來建立內部框架的是 <code>javax.swing.JInternalFrame</code> 類別，它先被加入 <code>JdesktopPane</code> ，接著被加入 <code>JFrame</code> 。如同一般的框架， <code>JInternalFrame</code> 可以加入元件。                                                  |   |
| 階層式窗格 (layered pane) 允許元件加入在指定的深度。深度指定為一個整數值。用來建立階層式窗格的是 <code>javax.swing.JLayeredPane</code> 類別。為了方便起見，你可利用此類別定義的標準階層：<br><code>DEFAULT_LAYER</code> 是最底層、<br><code>PALETTE_LAYER</code> 、<br><code>MODAL_LAYER</code> 、<br><code>POPUP_LAYER</code> 、<br><code>DRAG_LAYER</code> 則是最頂層。 |  |

使用程式容器類別時，你應將下列規則牢記於心：

- GUI 元件只有在層次結構內時才會被顯示出來。所謂層次結構，指的是由元件所構成的樹狀結構，其樹根為一個頂層程式容器。
- 一個 GUI 元件的物件實例 (instance)，在層次結構樹中，只能出現一次。若是一個程式容器裡的元件被加入另一個程式容器，此元件會被移至後來的程式容器裡，並從前一個程式容器中移除。
- 每個頂層程式容器都具有有一個 `contentPane`，通常含有 (直接或間接) 在該頂層程式容器 GUI 內的可見元件。
- 你可以隨意在頂層程式容器內加入選單列。按照慣例，選單列位於頂層程式容器內，內容窗格外。某些觀感，例如 **Mac** 觀感，讓你選擇是否將選單列置於更適合該 **look-and-feel** 的地方，如螢幕的最上方。

這四種程式容器（包括 **JApplet**）均實作（implement）一個特別的介面 **RootPaneContainer**。至於對 **RootPaneContainer** 更深的探討，則已超出本模組的範圍。

## **JFrame** 程式容器的要點

**JFrame** 程式容器是最常使用的頂層 **Swing** 程式容器。**JFrame** 程式容器允許你對「關閉視窗 (**Close Window**)」選單按鈕，設定四項回應動作中的一種。這四項回應動作分別是：

- `DO_NOTHING_ON_CLOSE`
- `HIDE_ON_CLOSE`
- `DISPOSE_ON_CLOSE`
- `EXIT_ON_CLOSE`

你可藉由呼叫 **JFrame** 實例中的 `setDefaultCloseOperation` 方法，來設定一個選項。

## Swing 元件

Swing GUI 使用兩種類別: GUI 類別與非 GUI 支援類別。GUI 類別是可見的, 並由 JComponent 所衍生, 稱之為 J 類別。非 GUI 類別提供服務, 並為 GUI 類別執行不可或缺的功能; 然而, 它們並不產生任何可見的輸出。

Swing 元件主要供應用於文字處理、按鈕、標籤、清單、窗格、複合框 (combo box)、捲軸列、捲軸式窗格、選單、表格、以及樹狀結構的元件。Swing 元件可概括分類如下:

- 按鈕
- 文字元件
- 不可編輯資訊的顯示元件
- 選單
- 格式化的顯示元件
- 其他基本控制項目

## Swing 元件階層架構

圖 15-5 展示了 Swing 元件的階層架構關係。

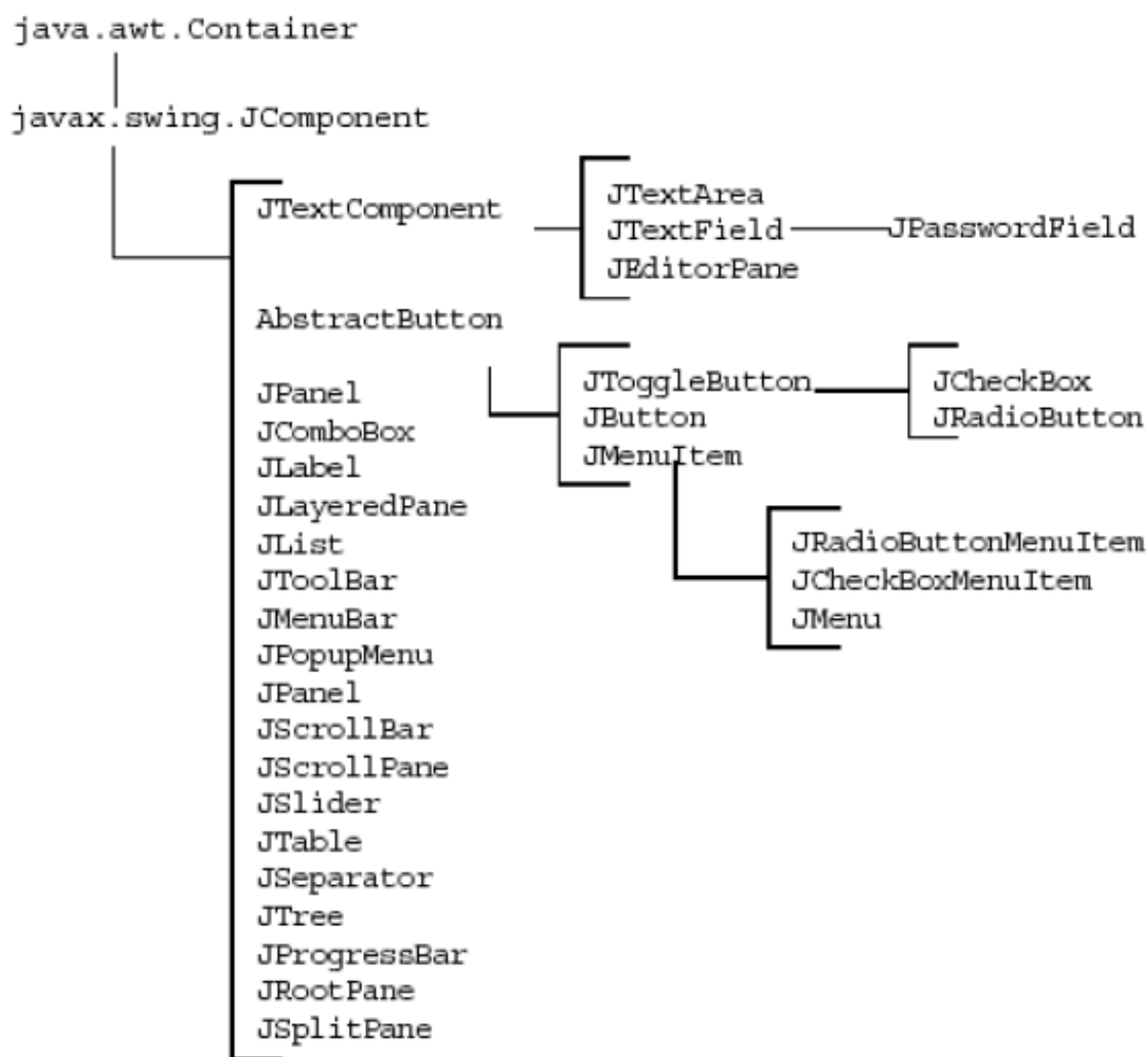


圖 15-5 Swing 元件階層架構

註釋 --Swing 元件的事件處理 (event handling) 類別，是非 GUI 類別的例子。

## 按鈕 (Buttons)

普通的按鈕 (**Button**)、核取框 (**checkbox**)、單選按鈕 (**radio button**) 都歸類於按鈕這一大類。若你使用 **Icon** 物件，定義你想顯示的圖形，那麼建立圖形化的按鈕也很容易。**JCheckBox** 類別提供核取框按鈕的支援。

**JRadioButton** 類別運作的方式是：點選單選按鈕群組中的某一個單選按鈕，就會取消選擇此單選按鈕群組中其他所有的按鈕。表 15-5 說明各元件並顯示它們的使用者介面。

表 15-5 按鈕

| 元件                                                                                                                                                                                                                                                                                                                                | 使用者介面                                                                                 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <b>JButton</b> 物件可以藉由傳遞一個簡單的 <b>String</b> 參數給建構子來建立，此狀況下它會把該字串顯示為它的標籤。點擊 <b>JButton</b> 會產生 <b>ActionEvent</b> 。你或許會想設定按鈕的動作命令 ( <b>action command</b> ) 屬性，讓 <b>ActionEvent</b> 傳達特定的命令字串。若你是用文字來建立 <b>JButton</b> ，標籤文字會被預設為動作命令字串。然而，若你建立的是只有圖形的按鈕，或是預設的動作命令字串（也就是按鈕上的標籤文字）不是你想要的，那麼你可以用 <b>setActionCommand</b> 方法明確的定義動作命令。 |    |
| 核取框類似於 <b>JButton</b> ，它可以藉由傳遞一個簡單的 <b>String</b> 參數給建構子來初始化 ( <b>initialize</b> )，此狀況下它會把該字串顯示為它的標籤。但它們的選取方式通常是大不相同。核取框具有一個 <b>boolean</b> 狀態值，可以是 <b>on(true)</b> 或 <b>off(false)</b> 。點擊核取框會切換它的狀態從 <b>on</b> 變成 <b>off</b> ，或從 <b>off</b> 變成 <b>on</b> 。用來建立核取框的是 <b>javax.swing.JCheckBox</b> 類別。                            |  |
| 單獨一個 <b>JRadioButton</b> 跟 <b>JCheckBox</b> 一樣，每次點選就單純的在 <b>on</b> 跟 <b>off</b> 間切換。要達成單選按鈕間互斥效果，則須將這些按鈕加在同一個 <b>ButtonGroup</b> 實例中。 <b>ButtonGroup</b> 會管理這些按鈕，確保一次只有一個按鈕能被選取。使用 <b>ButtonGroup</b> 類別來建立 <b>ButtonGroup</b> 。                                                                                                  |  |



# 文字元件

Swing 文字元件可概括分為三類：

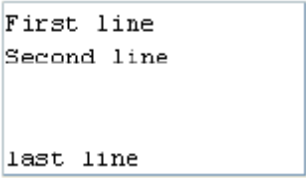
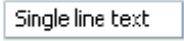
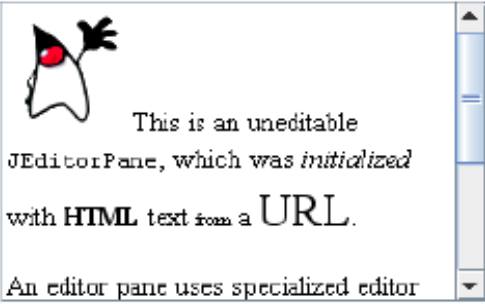
文字控制項目 —— JTextField、JPasswordField （用於使用者輸入）

純文字 (plain text) 區域 —— JTextArea （以純文字顯示文字，也用於多行的使用者輸入）

樣式 (styled) 文字區域 —— JEditorPane、JTextPane （顯示格式化的文字）

表 15-6 說明各元件並顯示它們的使用者介面。

表 15-6 文字元件

| 元件                                                                                                                             | 使用者介面                                                                                 |
|--------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| 文字區域通常用於向使用者收集一行以上的資訊。用來建立文字區域的是 javax.swing.JTextArea 類別。建立文字區域時，你可以指定列數、行數、及初始內容。文字區域僅能顯示純文字。                                |   |
| 文字欄位亦是用來向使用者收集資訊。類似於文字區域，不過它限制只可接收一行文字。用來建立文字欄位的是 javax.swing.JTextField 類別。                                                   |  |
| 編輯器窗格 (editor pane) 屬於樣式文字區域。除了純文字外，編輯器窗格還可以用 RTF 與 HTML 格式顯示及編輯文字。編輯器窗格通常用在顯示 HTML 格式。用來建立編輯器窗格的是 javax.swing.JEditorPane 類別。 |   |

Yu-Wei SHEN (SHEN\_Yu-Wei@yahoo.com.tw) has a non-transferable license to use this Student Guide

表 15-6 文字元件

| 元件                                                                                                                                                        | 使用者介面                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>javax.swing.JTextPane</code> 類別繼承 <code>javax.swing.JEditorPane</code> 類別。除了提供 <code>JEditorPane</code> 所有的特色外， <code>JTextPane</code> 類別還允許嵌入其他元件。 |  A screenshot of a Java Swing window containing a JTextPane. The text inside is "This is an editable JTextPane, another styled text component, which supports embedded components...". Below the text, there is a small icon of a megaphone and the text "...and embedded icons...". At the bottom of the pane, there is a cartoon pig illustration. The pane has a vertical scrollbar on the right side. |
| <code>javax.swing.JPasswordField</code> 是專門用來輸入密碼的文字欄位。為了安全起見，密碼欄位只顯示固定一個字元，如星號「*」。密碼欄位的值會儲存為字元陣列而非字串。如同其他的文字欄位物件，當你按下「Enter」按鍵時，密碼欄位會送出一個動作事件。           |  A screenshot of a Java Swing window containing a JPasswordField. The field is a rectangular box with a thin border, and it contains ten asterisks "*****" to represent masked input.                                                                                                                                                                                                                    |

## 不可編輯資訊的顯示元件

「不可編輯資訊的顯示元件」用來顯示關於元件本身的更多資訊。這類元件可以只當成顯示元件使用。表 15-7 說明部份此類元件。

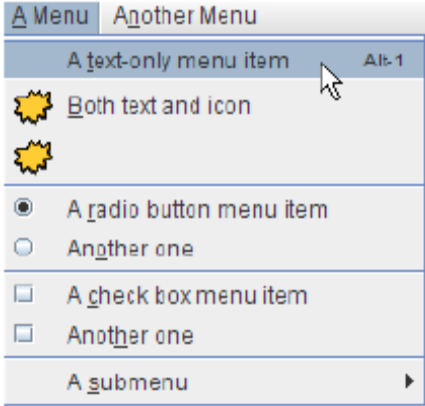
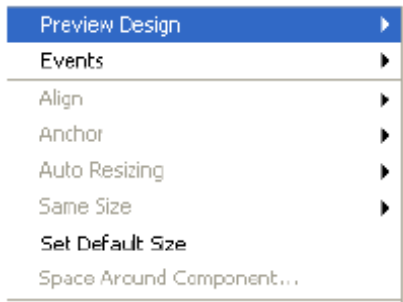
表 15-7 不可編輯的顯示元件

| 元件                                                                                                                                   | 使用者介面                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| <p>標籤 (Label) 用來在螢幕上顯示文字。它們是不可編輯的元件。用來建立標籤的是 <code>javax.swing.JLabel</code> 類別。標籤也能用來顯示圖像。</p>                                      |   |
| <p><code>javax.swing.JToolTip</code> 類別對顯示關於元件的資訊很有幫助。<code>JComponent</code> 提供 <code>setToolTipText</code> 方法，可以讓所有元件設定要顯示的字串。</p> |  |
| <p><code>javax.swing.JProgressBar</code> 類別可以協助執行時間較長的任務，顯示它目前的進度。</p>                                                               |  |

## 選單

選單的作用和清單類似，通常出現在選單列，或是以彈出式 (popup) 選單的方式呈現。選單列可以含有一個以上的選單（稱為下拉式選單），並依作業系統不同位置也不相同，通常在視窗的上方。彈出式選單則出現於當使用者觸發了作業平臺指定的滑鼠按鈕或鍵盤按鍵組合，例如：按下滑鼠右鍵，或是將滑鼠游標移至會彈出選單的元件之上。表 15-8 說明各元件並顯示它們的使用者介面。

表 15-8 選單元件

| 元件                                                                                                                                                                                                     | 使用者介面                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| 選單列含有一個以上的下拉式 (pull down) 選單名稱。點擊這些選單名稱，即可展開選單項目及子選單。用來建立選單的有 <code>javax.swing.JMenu</code> 與 <code>javax.swing.JMenuItem</code> 類別。選單項目能藉由助記鍵 (mnemonics) 與快速鍵 (accelerators) 來選取。選單項目也可能會是核取框或單選按鈕。 |   |
| 彈出式選單不依附選單列上。此類選單有時也稱作背景 (context) 選單。用來建立彈出式選單的是 <code>javax.swing.JPopupMenu</code> 類別。                                                                                                              |  |

## 格式化的顯示元件

「格式化的顯示元件」是 **Swing** 中最複雜的元件種類之一。一些例子如：表格、樹狀結構、色彩選取程式、及檔案選取程式等皆是這類元件。表 15-9 說明各元件並顯示它們的使用者介面。

表 15-9 格式化的顯示元件

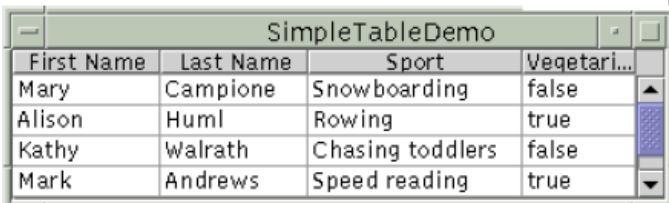
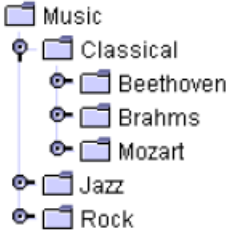
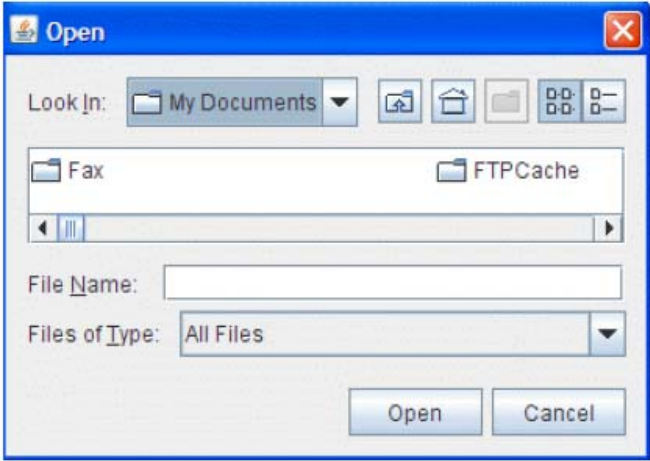

| 元件                                                                                                                                                                 | 使用者介面                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| 表格以格狀的形式來顯示及編輯資訊。用來建立表格的是 <b>javax.swing.JTable</b> 類別。 <b>JTable</b> 不儲存資料，僅顯示從表格模型傳來的資料。                                                                         |    |
| <b>JTree</b> 用在顯示階層式的資訊。建構顯示樣貌的資料是透過 <b>TreeModel</b> 實例所提供。資料也能由元素的集合來提供，例如 <b>Hashtable</b> 或 <b>Vector</b> 。 <b>JTree</b> 實際上不儲存資料，僅顯示 <b>TreeModel</b> 實例裡的資料。 |  |
| <b>JFileChooser</b> 類別允許使用者瀏覽檔案系統並選取檔案。檔案選取程式有數個過濾方法，讓你過濾欲顯示的檔案類型，以及其他方法，讓你自訂檔案選取程式展示檔案的方式。                                                                        |  |

表 15-9 格式化的顯示元件

| 元件                                                                                                                                                                                                                                                              | 使用者介面                                                                              |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <p><b>JColorChooser</b> 允許使用者操作及選擇色彩。有三種不同的方法可以選取色彩。「樣本 (Swatches)」、「色調、飽和、及亮度 (Hue, Saturation, and Brightness~HSB)」、「紅綠藍 (Red, Green, and Blue, RGB)」。<b>JColorChooser</b> 類別提供數個建構子，用於建立色彩選擇窗格。預設的建構子會建立初始色彩為白色的窗格。另有一個建構子能接收初始色彩作為參數。你可以在建構子中指定色彩選擇的模式。</p> |  |

## 其他基本控制項

本節說明其他的 **Swing** 元件，例如常用於 **GUI** 中的：複合框、清單、滑動軸 (slider)、及微調按鈕 (spinner)。**JComboBox** 讓使用者在數個選項中選取其中一項。**JComboBox** 類別具有一個方便的建構子，接受物件陣列作為初始選項。你可分別使用 **addItem** 與 **removeItem** 方法，來加入與移除選項。**JList** 將項目展示為一行以上。你可以藉由滑鼠點選或是鍵盤操縱，選取一個以上的陳列項目。利用 **JSlider**，你可以透過滑鼠點擊及拖曳，輸入數值。當螢幕空間有限時，微調按鈕則是另一個可以替代滑動軸的選擇。

表 15-10 說明各元件並顯示它們的使用者介面。

表 15-10 其他基本控制項

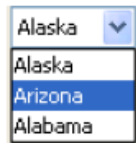
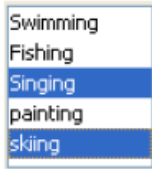

| 元件                                                                                                                             | 使用者介面                                                                                 |
|--------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| <p><b>JComboBox</b> 具有兩種形式：可編輯與不可編輯。預設的形式為不可編輯，它會顯示一個按鈕及下拉式的項目清單；而可編輯的形式則會顯示一個文字編輯欄位及選取按鈕。你可以在文字欄位內鍵入資料值，或使用按鈕來顯示下拉式的項目清單。</p> |  |

表 15-10 其他基本控制項 (Continued)

| 元件                                                                                                                                                                                                                                   | 使用者介面                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| 若項目的數量過多，因而無法全部顯示於指定的螢幕區域內時，清單也具有使用捲軸列的功能。建立清單時尚可將它設定成允許改變大小。在簡單的情況下， <b>JList</b> 類別更是易於使用，並且它視需要能建立自己的 <b>ListModel</b> 。要讓它建立自己的 <b>ListModel</b> ，你必須把資料項目放入 <b>Vector</b> 或是 <b>Objects</b> 陣列，然後用這些資料做為參數呼叫 <b>JList</b> 的建構子。 |   |
| 微調按鈕允許你鍵入資料值。 <b>JSpinner</b> 有三個子元件：向上的箭頭、向下的箭頭、及編輯器。編輯器可以是任意一種 <b>JComponent</b> ，預設為一個具有格式化文字欄位的面板。                                                                                                                               |   |
| 滑動軸的值域是有限的，亦即它有最小值和最大值。若有指定明確數字的需要，滑動軸也可以聯結格式化文字欄位。                                                                                                                                                                                  |  |



## Swing 元件屬性

本節說明 **Swing** 元件之屬性。

### 共通元件屬性

所有 **Swing** 元件都共同使用某些共通屬性，因為它們都延伸自 **JComponent** 類別。表 15-11 列出一部份所有 **Swing** 元件都會從 **JComponent** 繼承來的共通屬性。

表 15-11 共通元件屬性

| 屬性                                        | 方法                                                                   |
|-------------------------------------------|----------------------------------------------------------------------|
| 邊框 (Border)                               | Border getBorder()<br>void setBorder(Border b)                       |
| 背景與前景色彩 (Background and foreground color) | void setBackground(Color bg)<br>void setForeground(Color fg)         |
| 字型 (Font)                                 | void setFont(Font f)                                                 |
| 透明度 (Opaque)                              | void setOpaque(boolean isOpaque)                                     |
| 最大與最小尺寸 (Maximum and minimum size)        | void setMaximumSize(Dimension d)<br>void setMinimumSize(Dimension d) |
| 對齊 (Alignment)                            | void setAlignmentX(float ax)<br>void setAlignmentY(float ay)         |
| 偏好大小 (Preferred size)                     | void setPreferredSize(Dimension ps)                                  |



註釋 — 某些屬性，例如偏好大小，用來指示版面配置管理程式。雖然元件也能提供版面配置管理程式這些指示，但版面配置管理程式不一定須要遵循，而可能使用其他的資訊來繪製元件。



## 元件特有屬性

本節以 `JComboBox` 為例來討論元件屬性。`JComboBox` 繼承 `JComponent` 內所有的屬性，並定義更多它專有的屬性。表 15-12 說明部份 `JComboBox` 專有的屬性。

表 15-12 元件特有屬性 `Layout Managers`

| 屬性                       | 方法                                                 |
|--------------------------|----------------------------------------------------|
| 最大列數 (Maximum row count) | <code>void setMaximumRowCount(int count)</code>    |
| 模式 (Model)               | <code>void setModal(ComboBoxModel cbm)</code>      |
| 已選取索引 (Selected index)   | <code>int getSelectedIndex()</code>                |
| 已選取項目 (Selected Item)    | <code>Object getSelectedItem()</code>              |
| 項目總數 (Item count)        | <code>int getItemCount()</code>                    |
| 描繪程式 (Renderer)          | <code>void setRenderer(ListCellRenderer ar)</code> |
| 可編輯 (Editable)           | <code>void setEditable(boolean flag)</code>        |

.

## 版面配置管理程式

版面配置管理程式測定元件在程式容器內的大小及位置。相對於使用版面配置管理程式，另一種方法則是按照像素座標定位於絕對位置 (**absolute positioning**) 上。將程式容器的版面配置屬性設定為 **null**，即可使用絕對位置。但絕對位置未必可跨平臺使用。基於座標的版面配置，在原本平臺雖是正確的，卻可能因某些問題如：字型大小、螢幕解析度等，而無法在其他作業平臺使用。

不同於絕對位置，版面配置管理程式具有的機制，能妥善處理以下情況：

- 使用者更改 **GUI** 的大小
- 不同作業系統或使用者自訂的不同字型與字型大小
- 文字版面配置需要不同的跨國語系排列方式（從左至右、從右至左、從上至下）

欲善加處理這些情況，版面配置管理程式依循預先決定好的原則 (**policy**)，來展示元件。例如，**GridLayout** 原則將各個子元件放置在相同大小的單格 (**cells**) 中，從左上角開始，由左至右，由上至下，直到整個網格 (**grid**) 放滿為止。

接下去的章節說明一些可供你使用的版面配置管理程式。每一節都會強調目前討論的版面配置管理程式所使用的原則。

## BorderLayout 版面配置管理程式

**BorderLayout** 將元件佈置在五個不同的部位。**CENTER**、**NORTH**、**SOUTH**、**EAST**、以及 **WEST**。邊框版面配置限制各部位只能加入一個元件。

元件的位置必須明確指定。若未指定位置，按照預設方式，元件會被加在 **CENTER**。所有剩餘的額外空間，都會被分給在 **CENTER** 的元件。

**BorderLayout** 是 **JFrame**、**JDialog**、及 **JApplet** 預設的版面配置。15-31 頁圖 15-6 展示使用邊框版面配置的螢幕畫面。此例展示五個加入 **JFrame** 的 **JButton**。

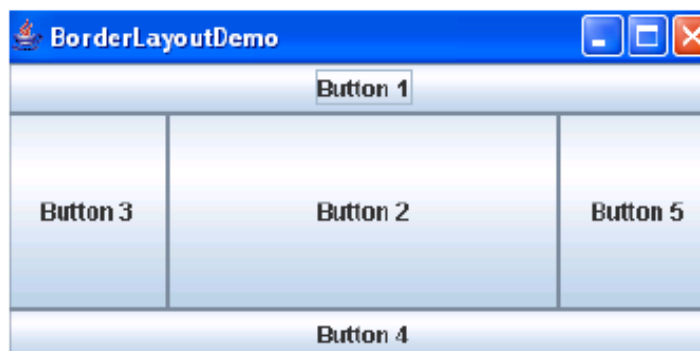


圖 15-6 BorderLayout 範例

## FlowLayout 版面配置管理程式

FlowLayout 將元件排成一列。按照預設方式，它會把元件排成 LEFT\_TO\_RIGHT。此方向可以用 `ComponentOrientation` 屬性改變為 RIGHT\_TO\_LEFT。可以指定元件間垂直和水平的間距。若未指定，則依預設，將垂直和水平的間距設為五個單位。圖 15-7 展示連貫版面配置的範例。類似邊框版面配置的範例，此例亦展示五個加入 `JFrame` 的 `JButton`。



圖 15-7 FlowLayout 範例

以下的程式碼可以顯示一個 FlowLayout 範例。它將五個按鈕加入 `JFrame`。

```

1 import javax.swing.*;
2 import java.awt.*;
3
4 public class LayoutExample {
5 private JFrame f;
6 private JButton b1;
7 private JButton b2;
8 private JButton b3;
9 private JButton b4;
10 private JButton b5;
11
12 public LayoutExample() {
13 f = new JFrame("GUI example");

```

```

14 b1 = new JButton("Button 1");
15 b2 = new JButton("Button 2");
16 b3 = new JButton("Button 3");
17 b4 = new JButton("Button 4");
18 b5 = new JButton("Button 5");
19 }
20
21 public void launchFrame() {
22 f.setLayout(new FlowLayout());
23 f.add(b1);
24 f.add(b2);
25 f.add(b3);
26 f.add(b4);
27 f.add(b5);
28 f.pack();
29 f.setVisible(true);
30 }
31
32 public static void main(String args[]) {
33 LayoutExample guiWindow = new LayoutExample();
34 guiWindow.launchFrame();
35 }
36
37 } // end of LayoutExample class

```

## BoxLayout 版面配置管理程式

**BoxLayout** 建構子接收一個 **axis** 參數，指定元件對齊的方向。此參數可以接受以下任意一個值：

- **X\_AXIS** —— 元件水平的從左至右排列
- **Y\_AXIS** —— 元件垂直的從上至下排列
- **LINE\_AXIS** —— 元件按照一行內文字順序的方向排列
- **PAGE\_AXIS** —— 元件按照一頁內每行順序的方向排列
-

圖 15-8 展示方塊式版面配置的範例。類似邊框版面配置的範例，此例亦展示五個加入 `JFrame` 的 `JButton`。在這個範例中，用來對齊元件的參數設定為 `Y_AXIS`。

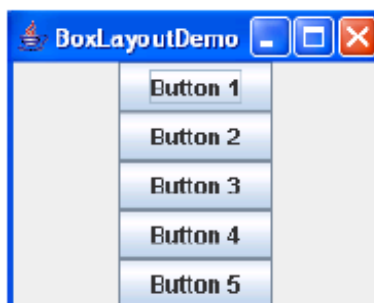


圖 15-8 BoxLayout 範例

## CardLayout 版面配置管理程式

`CardLayout` 將元件當作一疊卡片來放置。每張卡片都可以顯示一個元件，而同一時間內只能顯現一張卡片。但如果你所放置的這個元件為程式容器的話，就能在一張卡片上顯示許多個元件了。卡片式版面配置的使用法如圖 15-9 所示。

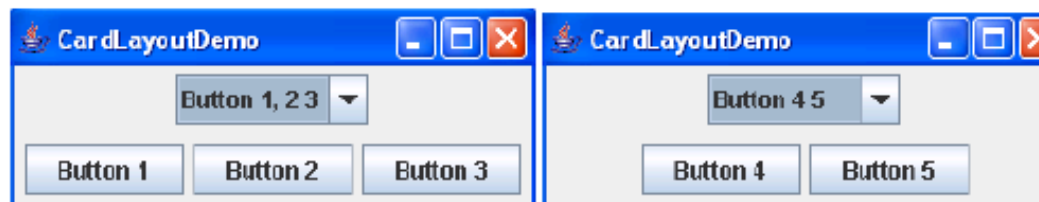


圖 15-9 CardLayout 範例

圖 15-9 所示的範例中 `Button1`、`Button2`、與 `Button3` 放在 `Card1` 上，而 `Button4` 與 `Button5` 放在 `Card2` 上。使用複合框來選擇要顯示的卡片。

## GridLayout 版面配置管理程式

**GridLayout** 將元件按照行與列來放置。各元件在程式容器裡佔用相同大小的空間。建立網格式版面配置時，須指定行數與列數。若未指定，則依照預設，版面配置管理程式會建立單行單列。可以指定元件間的垂直間距和水平間距。圖 15-10 展示網格式版面配置的用法。類似 15-31 頁圖 15-6 邊框版面配置的範例，此例亦在 **JFrame** 中加入五個 **JButton**。

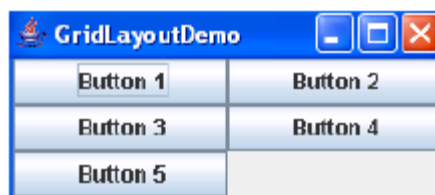


圖 15-10 GridLayout 範例

## GridBagLayout 版面配置管理程式

**GridBagLayout** 將元件按照行與列來放置，如同網格式版面配置，但提供各式各樣的彈性選項，來變動元件的大小及位置。此種版面配置是用於設計複雜的 GUI。元件的限制條件用 **GridBagConstraints** 類別來指明。該類別內含的限制條件有：**gridwidth**、**gridheight**、**gridx**、**gridy**、**weightx**、與 **weighty**...等等。圖 15-11 展示 **GridBagLayout** 的用法。**JFrame** 中加入了五個 **JButton**。注意：元件有不同的大小，並且各放置於特定的位置上。

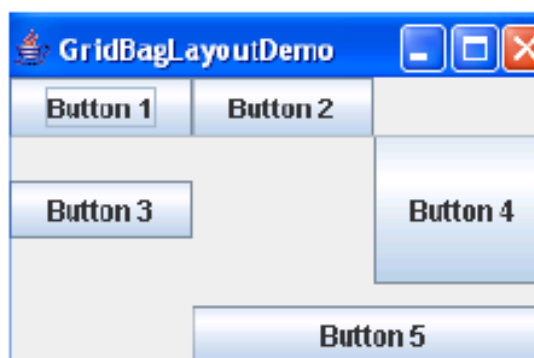


圖 15-11 GridBagLayout 範例

## GroupLayout 版面配置管理程式

除了上述的那些版面配置管理程式外，**Java SE 第 6 版** 還加入了 **GroupLayout**。此版面配置管理程式，是為了讓 **GUI 工具開發人員** 使用而加入的。它是 **NetBeans IDE GUI 生成器** 工具所採用的版面配置管理程式

## 建構 GUI 應用程式

Java GUI 應用程式可用以下兩種方法建立：

- 使用程式建構  
使用程式碼建立 GUI。此法對學習 GUI 的建構十分有幫助。然而，在實際製作產品時，這樣做會非常吃力。
- 運用 GUI 產生器工具  
使用 GUI 產生器工具來建立 GUI。GUI 開發者透過視覺化的方法，拖放程式容器與元件至工作區域。此工具可藉由指示裝備（如電腦滑鼠）來變動程式容器與元件的位置與大小。在每一個步驟，此工具均自動產生出製作 GUI 所必需的 Java 技術類別。

### 使用程式建構

本節說明如何建立一個簡單的 GUI，顯示「Hello World」。程式 15-1 中的程式碼會建立一個標題為「HelloWorldSwing」的 JFrame 程式容器。接著它加入 JLabel，並將它的「存取名稱 (Accessible Name)」屬性設為「Hello World」。

程式 15-1 HelloWorldSwing 應用程式

```

1 import javax.swing.*;
2 public class HelloWorldSwing {
3 private static void createAndShowGUI() {
4 JFrame frame = new JFrame("HelloWorldSwing");
5 //Set up the window.
6 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7 JLabel label = new JLabel("Hello World");
8 // Add Label
9 frame.add(label);
10 frame.pack();
11 // Display Window
12 frame.setVisible(true);}
13
14 public static void main(String[] args) {
15 javax.swing.SwingUtilities.invokeLater(new Runnable() {
16 //Schedule for the event-dispatching thread:
17 //creating, showing this app's GUI.
18 public void run() {createAndShowGUI();}
19 });
20 }
21 }
```



圖 15-12 展示這段程式產生的 GUI 介面。JFrame 預設的版面配置是 BorderLayout。因此，依照預設方式，JLabel 元件被加入在程式容器的中間位置。請注意該標籤會佔滿整個框架，因為在 BorderLayout 中，中間的元件會佔據程式容器剩餘的所有空間。

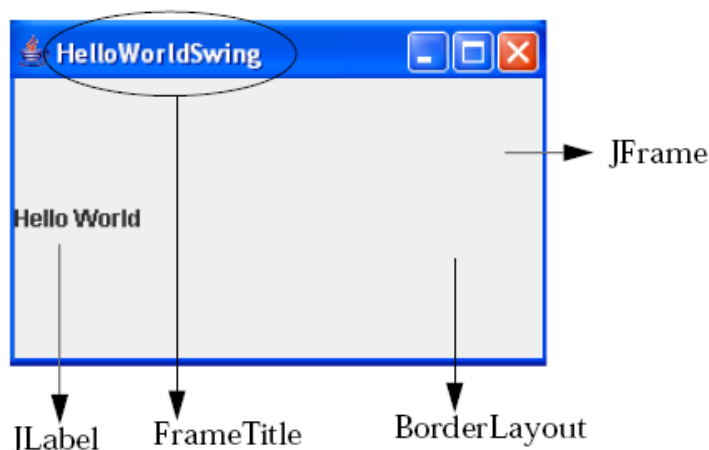


圖 15-12 HelloWorldString 輸出畫面

## 關鍵方法

本節說明程式 15-1 中使用到的一些關鍵方法。這些方法可以分成兩大類。

1. 用來建立框架及加上標籤的方法。
  - a. **setDefaultLookAndFeelDecorated(true)**: 此方法提供指示，說明建立的 JFrame 是否要有視窗裝飾物，例如 —— 邊框、工具集、最小化、關閉、以及最大化視窗的選項。
  - b. **setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE)**: 此方法定義當關閉視窗動作進行時，JFrame 該有的反應。處理的方法有四種 ——
    1. **DO\_NOTHING\_ON\_CLOSE**: 當關閉視窗動作進行時，無須作事。此常數已定義在 **WindowsConstants** 內。
    2. **HIDE\_ON\_CLOSE**: 叫用所有 **WindowsListener** 物件，並將框架隱藏起來。此常數已定義在 **WindowsConstants** 內。
    3. **DISPOSE\_ON\_CLOSE**: 叫用所有 **WindowsListener** 物件，並將框架隱藏起來後釋放其佔用的資源。此常數已定義在 **WindowsConstants** 內。
    4. **EXIT\_ON\_CLOSE**: 結束應用程式。此常數已定義在 **JFrame** 類別內。

- c. **pack()**: **JFrame** 從 **java.awt.Window** 繼承了本方法。本方法協助改變視窗大小，讓所有元件的版面配置及它們偏好的大小都被保存起來。
  - d. **setVisible(true)**: 當 **JFrame** 首次被建立時，它會建立一個不可見的框架。要讓它成為可見的話，**setVisible** 的參數須設為 **true**。**JFrame** 從 **java.awt.Component** 繼承了本方法。
  - e. **add(Component c)**: 本方法將元件加入程式容器中。**JFrame** 從 **java.awt.Component** 類別繼承了本方法。在 **java.awt.Component** 內共定義了五種不同的多載 (overloaded) 方法。
2. 用來維護 GUI 執行緒安全 (thread-safe) 及使 GUI 更有效率的方法。許多工作都與有效率的顯示 GUI 有關。這些工作可以大致定義為：
- a. 執行應用程式的程式碼。  
此工作牽涉到啟動 GUI 應用程式，與執行繪製 GUI 的程式碼。
  - b. 處理 GUI 發生的事件：  
**GUI** 內的元件可能發生許多事件。例如，當按鈕被按下時就產生了一個事件。處理這個事件須要定義事件監聽程式 (listener)。本項工作將事件派送至恰當的事件監聽程式去處理。

c. 處理某些耗時的程式：

某些動作可能相當耗時，可以讓它們在背景執行，使 GUI 能更快速有效的回應。本項工作處理這類活動。

想要有效率的處理這些工作，**Swing** 框架使用執行緒這類輕型 (**light-weight**) 的程式。所以上述工作，都能分開來且同時用這些執行緒來處理。開發人員應該善用這些執行緒。**Swing** 框架在 **SwingUtilities** 類別中供應了一整組的工具方法。

1 **SwingUtilities.invokeLater(new Runnable()):**

在 **Java** 程式語言中，執行緒是使用 **Runnable** 介面來建立。該介面定義了 **run** 方法，所有使用此介面的類別均須實作它。**invokeLater** 方法會規劃事件處理的執行緒，在所有未處理的事件處理完後，才非同步的 (**asynchronously**) 開始執行 **run** 方法，進行 GUI 建立工作。



# 處理 GUI 產生的事件 (event)

---

## 目標

完成本單元後，您將學會：

- 事件及處理事件 (event handling)
- 檢視 Java SE 事件模型
- 描述 GUI 的行為
- 確認會引發事件的使用者動作
- 開發 listener
- 能夠理解以 Swing 為基礎的 GUI 其並行性 (concurrency)，以及描述 SwingWorker 類別的特色

## 其他資源



其他資源 — 對於本模組中敘述之主題，提供以下額外資訊做為參考：

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O' Reilly Media. 2005.

## 何謂事件？

當使用者在使用者介面上執行動作時（點擊滑鼠或按下按鈕），就會引發出事件。事件是描述發生何種狀況的物件。有許多不同類型的事件類別，可以用來描述不同種類的使用者動作。

圖 16-1 展示委派 (delegation) 事件模型的抽象化觀點。當使用者點擊 GUI 按鈕，JVM 就會建立事件物件，並且該按鈕會呼叫 `actionPerformed` 方法，將此事件送交給負責該按鈕的事件處理程式。

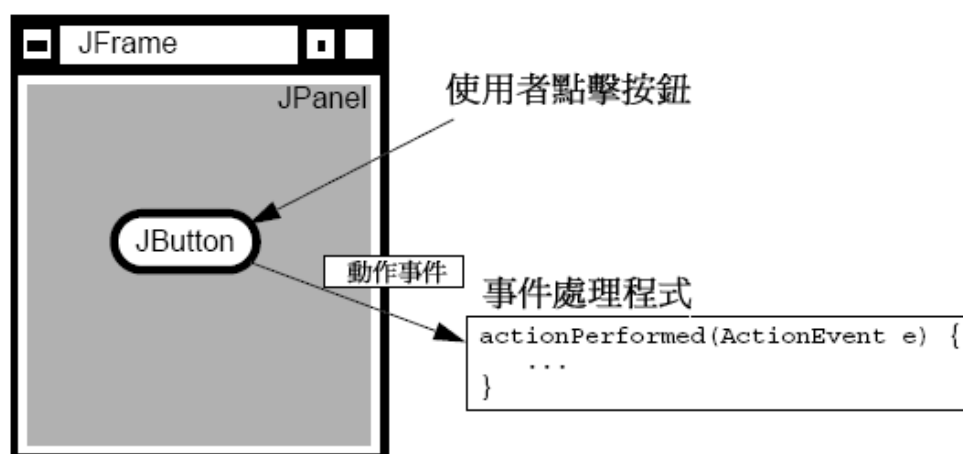


圖 16-1 使用者動作觸發了一個事件

### 事件來源 (Event Source)

事件來源意指事件的產生者。例如，當滑鼠點擊在 JButton 元件上，就會產生一個來源為該按鈕的 `ActionEvent` 實例 (instance)。ActionEvent 實例是一個物件，它含有關於剛才發生事件的資訊。ActionEvent 內含：

- `getActionCommand`，傳回與該動作相關的命令名額
- `getModifiers`，傳回在該動作期間是否按下了任何輔助按鍵
- `getWhen`，傳回事件發生時的時間戳記
- `paramString`，傳回識別該動作及相關命令的字串

### 事件處理器 (event handlers)

事件處理器指的是一個方法，它接收事件物件，解釋它，並且處理該與使用者進行怎樣的互動。

## Java SE 事件模型

本節解釋委派事件模型。

### 委派模型

委派事件模型從 **JDK version 1.1** 起就已存在。在此模型下，事件從其發生處被送交給元件，再由各元件決定將事件傳遞給（一個以上）稱為監聽程式 (**Listener**) 的已註冊類別。監聽程式內含事件處理器，以接收和處理事件。依此方式，事件處理器可以存在於跟元件完全隔離的另一個物件內。監聽程式是實作 **EventListener** 介面的類別。圖 16-2 展示事件委派模型。

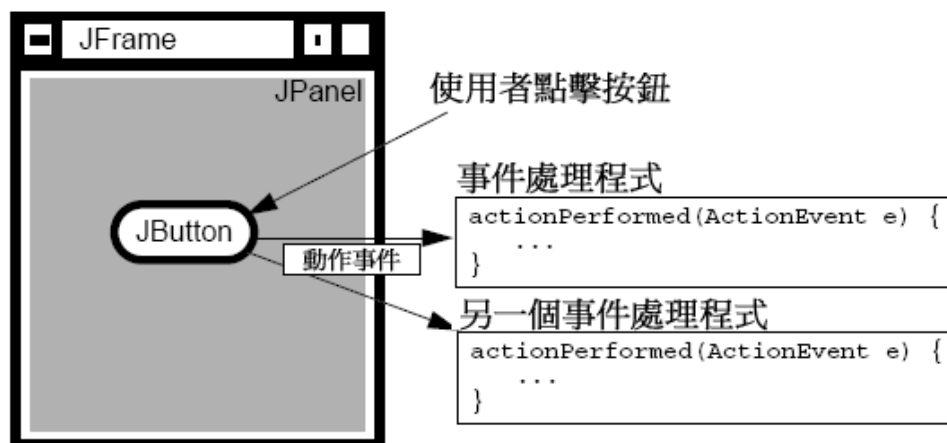


圖 16-2 具有多個監聽程式的委派事件模型

事件是只回報給已註冊監聽程式的物件。每個事件都有相關的監聽程式介面，指示在適合接收該類型事件的類別裡，必須定義哪些方法。實作該介面的類別定義了這些方法後，才可以註冊為監聽程式。

從元件來的事件，若沒有註冊任何監聽程式，就不會被傳送出去。



## 監聽程式範例

舉個例子，程式 16-1 的程式碼是有一個 **JButton** 在上面的簡易 **JFrame**。

程式 16-1 TestButton 範例

```

1 import javax.swing.*;
2 import java.awt.*;
3
4 public class TestButton {
5 private JFrame f;
6 private JButton b;
7
8 public TestButton() {
9 f = new JFrame("Test");
10 b = new JButton("Press Me!");
11 b.setActionCommand("ButtonPressed");
12 }
13
14 public void launchFrame() {
15 b.addActionListener(new ButtonHandler());
16 f.add(b, BorderLayout.CENTER);
17 f.pack();
18 f.setVisible(true);
19 }
20
21 public static void main(String args[]) {
22 TestButton guiApp = new TestButton();
23 guiApp.launchFrame();
24 }
25 }

```

**ButtonHandler** 類別，如程式 16-2 所示，是事件所委任的處理程式類別。

程式 16-2 ButtonHandler 範例

```

1 import java.awt.event.*;
2
3 public class ButtonHandler implements ActionListener {
4 public void actionPerformed(ActionEvent e) {
5 System.out.println("Action occurred");
6 System.out.println("Button's command is: "
7 + e.getActionCommand());
8 }
9 }

```

16-5 頁的程式 16-2 具有下列特性：

處理 GUI 產生的事件 (event)

Copyright 2007 Sun Microsystems, Inc. All Rights Reserved. Sun Services, Revision G

Unauthorized reproduction or distribution prohibited. Copyright © 2010, Oracle and/or its affiliates.

- `JButton` 類別從它的父類別 `javax.swing.AbstractButton` 繼承了 `addActionListener(ActionListener)` 方法。
- `ActionListener` 介面定義了一個 `actionPerformed` 方法，接收 `ActionEvent`。
- 在 `JButton` 物件建立後，它可以透過 `addActionListener()` 方法，將一個物件註冊為 `ActionEvent` 的監聽程式。註冊之監聽程式，須是由實作 `ActionListener` 介面的類別所建立的實例。
- 當點擊 `JButton` 物件時，便送出 `ActionEvent`。該按鈕用它的 `addActionListener()` 方法註冊 `ActionListener`，而 `ActionEvent` 便透過所有已註冊 `ActionListener` 的 `actionPerformed()` 方法，來被接收。
- `ActionEvent` 類別的 `getActionCommand()` 方法，會傳回與此動作相關的命令名稱。在第 11 行中，此按鈕的命令名稱被設定為「`ButtonPressed`」。

---

註釋 — 尚有其他方法可以確認為何會收到某個事件。其中 `java.util.EventObject` 基礎類別內的 `getSource()` 方法通常特別有用，因為它能讓你獲得送出該事件的物件參照 (reference)。

---

事件並不會被意料之外的處理器所處理。規劃用來監聽某一個 GUI 元件上某一些事件的物件，會事先將它們本身向該物件註冊。

- 當事件發生時，只有註冊過的物件會收到該事件已發生的訊息。
- 委派模型適合將工作分配在數個物件之間。

事件不一定要與 `Swing` 元件有關，例如事件模型也提供 `JavaBeans` 架構所需的支援。

# GUI 的行為

本節說明事件的各類。

## 事件分類

用來從元件接收事件的概況，前面已在只有一種事件類型的背景下討論過了。許多事件類別都存在於 `java.awt.event` 套件裡，但尚有其他事件存在於 API 內的別處。圖 16-3 展示 GUI 事件類別的 UML 類別階層架構。

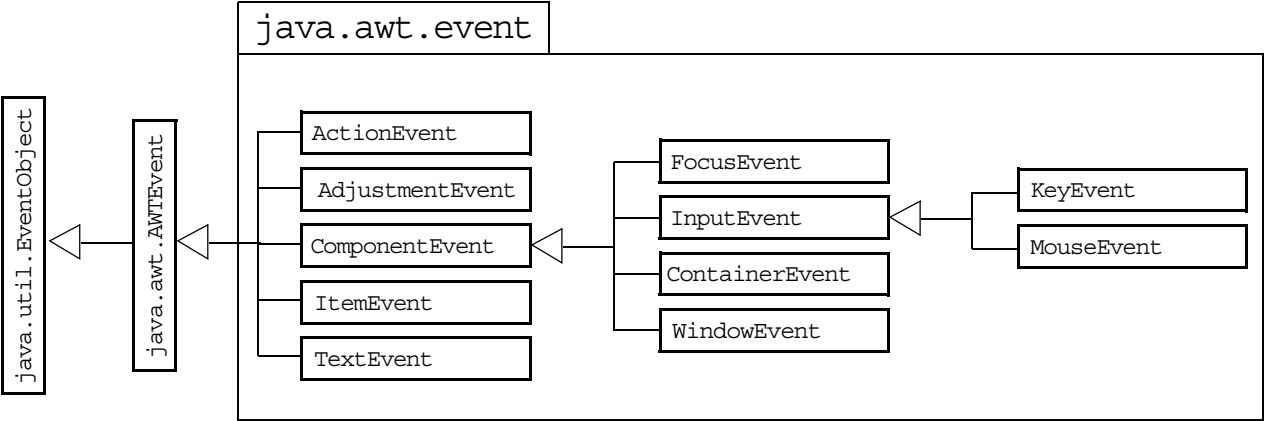


圖 16-3 GUI 事件的類別階層架構

各事件分類都具有有一個介面，由欲接收該類事件的物件類別所實作。該介面會要求定義一個以上的方法。當特定的事件發生時，就呼叫這些方法。表 16-1 列舉這些分類及介面。

表 16-1 方法、分類、及介面

| 分類          | 介面名稱           | 方法                                                                                                                                       |
|-------------|----------------|------------------------------------------------------------------------------------------------------------------------------------------|
| 動作 (Action) | ActionListener | actionPerformed(ActionEvent)                                                                                                             |
| 項目 (Item)   | ItemListener   | itemStateChanged(ItemEvent)                                                                                                              |
| 滑鼠 (Mouse)  | MouseListener  | mousePressed(MouseEvent)<br>mouseReleased(MouseEvent)<br>mouseEntered(MouseEvent)<br>mouseExited(MouseEvent)<br>mouseClicked(MouseEvent) |

表 16-1 方法、分類、及介面

| 分類                      | 介面名稱                | 方法                                                                                                                                                                                                                       |
|-------------------------|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 滑鼠動作<br>(Mouse motion)  | MouseMotionListener | mouseDragged(MouseEvent)<br>mouseMoved(MouseEvent)                                                                                                                                                                       |
| 按鍵 (Key)                | KeyListener         | keyPressed(KeyEvent)<br>keyReleased(KeyEvent)<br>keyTyped(KeyEvent)                                                                                                                                                      |
| 焦點<br>(Focus)           | FocusListener       | focusGained(FocusEvent)<br>focusLost(FocusEvent)                                                                                                                                                                         |
| 調整<br>(Adjustment)      | AdjustmentListener  | adjustmentValueChanged<br>(AdjustmentEvent)                                                                                                                                                                              |
| 元件<br>(Component)       | ComponentListener   | componentMoved(ComponentEvent)<br>componentHidden(ComponentEvent)<br>componentResized(ComponentEvent)<br>componentShown(ComponentEvent)                                                                                  |
| 視窗<br>(Window)          | WindowListener      | windowClosing(WindowEvent)<br>windowOpened(WindowEvent)<br>windowIconified(WindowEvent)<br>windowDeiconified(WindowEvent)<br>windowClosed(WindowEvent)<br>windowActivated(WindowEvent)<br>windowDeactivated(WindowEvent) |
| 程式容器<br>(Container)     | ContainerListener   | componentAdded(ContainerEvent)<br>componentRemoved(ContainerEvent)                                                                                                                                                       |
| 視窗狀態<br>(Window state)  | WindowStateListener | windowStateChanged(WindowEvent e)                                                                                                                                                                                        |
| 視窗焦點<br>(Window focus)  | WindowFocusListener | windowGainedFocus(WindowEvent e)<br>windowLostFocus(WindowEvent e)                                                                                                                                                       |
| 滑鼠滾輪<br>(Mouse wheel)   | MouseWheelListener  | mouseWheelMoved(MouseWheelEvent e)                                                                                                                                                                                       |
| 輸入方法<br>(Input methods) | InputMethodListener | caretPositionChanged (InputMethodEvent e)<br>inputMethodTextChanged (InputMethodEvent e)                                                                                                                                 |

表 16-1 方法、分類、及介面

| 分類                        | 介面名稱                    | 方法                                                                   |
|---------------------------|-------------------------|----------------------------------------------------------------------|
| 階層架構 (Hierarchy)          | HierarchyListener       | hierarchyChanged(HierarchyEvent e)                                   |
| 階層架構界限 (Hierarchy bounds) | HierarchyBoundsListener | ancestorMoved(HierarchyEvent e)<br>ancestorResized(HierarchyEvent e) |
| AWT                       | AWTEventListener        | eventDispatched(AWTEvent e)                                          |
| 文字 (Text)                 | TextListener            | textValueChanged(TextEvent)                                          |

## 複雜的範例

本節檢視一個更複雜的 **Java** 軟體程式範例。當按下滑鼠按鍵時，它會追蹤滑鼠的動作（滑鼠拖曳）。即使未按下滑鼠按鍵，它也會偵測滑鼠的動作（滑鼠移動）。

無論有無按下滑鼠按鍵，移動滑鼠所產生的事件，可以被實作 **MouseMotionListener** 介面的類別物件接取。此介面要求兩個方法：**mouseDragged()** 與 **mouseMoved()**。即使你只對拖曳滑鼠有興趣，仍須同時提供兩個方法。不過，你可以空著 **mouseMoved()** 方法的程式主體。

每當滑鼠或鍵盤事件發生時，有關滑鼠位置或按下的按鍵資訊，都會出現在它所產生的事件中。在 16-5 頁程式 16-2 的事件處理中，有一個獨立的類別 **ButtonHandler** 會處理事件。程式 16-3 裡，事件是在一個稱為 **TwoListener** 的類別內處理。

### 程式 16-3 TwoListener 範例

```

1 import javax.swing.*;
2 import java.awt.event.*;
3 import java.awt.*;
4
5 public class TwoListener
6 implements MouseMotionListener, MouseListener {
7 private JFrame f;
8 private JTextField tf;
9
10 public TwoListener() {
11 f = new JFrame("Two listeners example");
12 tf = new JTextField(30);

```

```
13 }
14
15 public void launchFrame() {
16 JLabel label = new JLabel("Click and drag the mouse");
17 // Add components to the frame
18 f.add(label, BorderLayout.NORTH);
19 f.add(tf, BorderLayout.SOUTH);
20 // Add this object as a listener
21 f.addMouseMotionListener(this);
22 f.addMouseListener(this);
23 // Size the frame and make it visible
24 f.setSize(300, 200);
25 f.setVisible(true);
26 }
27
28 // These are MouseMotionListener events
29 public void mouseDragged(MouseEvent e) {
30 String s = "Mouse dragging: X = " + e.getX()
31 + " Y = " + e.getY();
32 tf.setText(s);
33 }
34
35 public void mouseEntered(MouseEvent e) {
36 String s = "The mouse entered";
37 tf.setText(s);
38 }
39
40 public void mouseExited(MouseEvent e) {
41 String s = "The mouse has left the building";
42 tf.setText(s);
43 }
44
45 // Unused MouseMotionListener method.
46 // All methods of a listener must be present in the
47 // class even if they are not used.
48 public void mouseMoved(MouseEvent e) { }
49
50 // Unused MouseListener methods.
51 public void mousePressed(MouseEvent e) { }
52 public void mouseClicked(MouseEvent e) { }
53 public void mouseReleased(MouseEvent e) { }
54
55 public static void main(String args[]) {
56 TwoListener two = new TwoListener();
57 two.launchFrame();
58 }
```

59 }

以下章節說明 16-9 頁程式 16-3 的幾個重點。

## 實作多個介面

第 5 跟第 6 行宣告類別時，使用了以下指令：

```
implements MouseMotionListener, MouseListener
```

你可以用逗號隔開，宣告多個介面。

## 監聽多個來源

若你在第 20 及第 21 行呼叫以下方法

```
f.addMouseListener(this);
f.addMouseMotionListener(this);
```

則兩種類型的事件都會導致 **TwoListener** 類別內的方法被呼叫。一個物件可以依其需要，監聽多個事件來源。此物件之類別須實作所需的介面。

## 取得關於事件的詳細內容

事件管理程式的方法（如 `mouseDragger()`）被呼叫時使用的事件參數，含有關於原始事件可能很重要的資訊。對於各個事件分類，想要詳細確認有哪些資訊可以使用，請參考 `java.awt.event` 套件裡對應的類別文件。

## 多重監聽程式

AWT 事件監聽架構允許在同一個元件上附加多個監聽程式。一般來說，若想撰寫一個基於單一事件而執行多個動作的程式，那就將此行為的程式碼放進你的管理程式方法 (**handler method**) 內。然而程式的設計，有時候需要多個在程式中互不相關的部份去回應同一個事件，例如：在現有程式中加入背景感應的協助工具。

監聽程式的機制允許你依需求而多次呼叫 `addXxxListener()` 方法，並且你也能照設計需求而指定多種不同的監聽程式。當事件發生時，所有已註冊監聽程式的管理程式方法都會被呼叫到。



---

註釋 -- 管理程式方法被呼叫的次序並不是確定的。一般來說，若呼叫的順序會有所影響，則管理程式通常互不相關。在此情況下，只須註冊第一個監聽程式，並讓此監聽程式直接呼叫其他的監聽程式。

---



## 開發事件監聽程式

在本節中，你將學習有關實作事件監聽程式時，設計與作法上的一些選擇。

### 事件配接器 (adapters)

對於各個監聽程式介面，需要極大的工夫才能實作所有方法，尤其是 `MouseListener` 介面與 `WindowListener` 介面。

舉例來說，`MouseListener` 介面宣告了以下方法：

```
public void mouseClicked(MouseEvent event)
public void mouseEntered(MouseEvent event)
public void mouseExited(MouseEvent event)
public void mousePressed(MouseEvent event)
public void mouseReleased(MouseEvent event)
```

為了方便起見，Java 程式語言提供配接程式類別，實作每個含有超過一個方法的介面。在配接程式類別中，這些方法都是空的。

你可以延伸一個配接器類別，並且只覆寫 (override) 你需要的那些方法，如程式 16-4 所示。

程式 16-4 `MouseClickedHandler` 範例

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4
5 public class MouseClickHandler extends MouseAdapter {
6
7 // We just need the mouseClicked handler, so we use
8 // an adapter to avoid having to write all the
9 // event handler methods
10
11 public void mouseClicked(MouseEvent e) {
12 // Do stuff with the mouse click...
13 }
14 }
```

## 使用內部類別 (inner classes) 進行事件處理

程式 16-5 的第 26 行與 12-18 行，展示如何將事件處理程式建立為內部類別。使用內部類別做為事件處理程式，讓你能夠存取外部類別 (outer class) 的私有 (private) 資料 (第 16 行)。

程式 16-5 TestInner 範例

```

1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 public class TestInner {
5 private JFrame f;
6 private JTextField tf;
7
8 public TestInner() {
9 f = new JFrame("Inner classes example");
10 tf = new JTextField(30);
11 }
12
13 class MyMouseMotionListener extends MouseMotionAdapter {
14 public void mouseDragged(MouseEvent e) {
15 String s = "Mouse dragging: X = " + e.getX()
16 + " Y = " + e.getY();
17 tf.setText(s);
18 }
19 }
20
21 public void launchFrame() {
22 JLabel label = new JLabel("Click and drag the mouse");
23 // Add components to the frame
24 f.add(label, BorderLayout.NORTH);
25 f.add(tf, BorderLayout.SOUTH);
26 // Add a listener that uses an Inner class
27 f.addMouseMotionListener(new MyMouseMotionListener());
28 f.addMouseListener(new MouseClickHandler());
29 // Size the frame and make it visible
30 f.setSize(300, 200);
31 f.setVisible(true);
32 }
33
34 public static void main(String args[]) {
35 TestInner obj = new TestInner();
36 obj.launchFrame();
37 }
38 }

```

## 使用匿名類別進行事件處理

你可以將整個類別的定義含括在一個表示式的範圍中。此法定義所謂的匿名內部類別，並同時建立它的實例。匿名內部類別常用於事件處理，程式 16-6 是一個範例。

程式 16-6 TestAnonymous 範例

```

1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4
5 public class TestAnonymous {
6 private JFrame f;
7 private JTextField tf;
8
9 public TestAnonymous() {
10 f = new JFrame("Anonymous classes example");
11 tf = new JTextField(30);
12 }
13
14 public void launchFrame() {
15 JLabel label = new JLabel("Click and drag the mouse");
16 // Add components to the frame
17 f.add(label, BorderLayout.NORTH);
18 f.add(tf, BorderLayout.SOUTH);
19 // Add a listener that uses an anonymous class
20 f.addMouseListener(new MouseMotionAdapter() {
21 public void mouseDragged(MouseEvent e) {
22 String s = "Mouse dragging: X = " + e.getX()
23 + " Y = " + e.getY();
24 tf.setText(s);
25 }
26 }); // <- note the closing parenthesis
27 f.addMouseListener(new MouseClickHandler()); // Not shown
28 // Size the frame and make it visible
29 f.setSize(300, 200);
30 f.setVisible(true);
31 }
32
33 public static void main(String args[]) {
34 TestAnonymous obj = new TestAnonymous();
35 obj.launchFrame();
36 }
37 }

```



---

註釋 -- 編譯匿名類別時會產生一個檔案，如 `TestAnonymous$1.class`。

---

## Swing 的並行性 (concurrency)

含有 GUI 的應用程式，需要數個執行緒以有效率的處理 GUI。

- 負責執行應用程式的執行緒，稱為現行執行緒 (**current threads**)。
- 負責處理各種元件所產生之事件之執行緒，稱為事件派送執行緒 (**event dispatch threads**)。
- 負責執行耗時工作的執行緒，稱為工作執行緒 (**worker threads**)，這些工作例如：等待某些共用的資源、等待使用者輸入、網路或磁碟 I/O 擁塞、執行特別耗用 CPU 或記憶體之運算等，可以在背景執行，而不影響 GUI 的效能。

你可以使用 **SwingWorker** 類別的實例去代理這些工作執行緒。**SwingWorker** 類別延伸自 **Object** 類別，並實作了 **RunnableFuture** 介面。

**SwingWorker** 類別提供以下的工具方法：

- 為了溝通和協調工作執行緒的工作與其他執行緒的工作，**SwingWorker** 類別提供若干屬性，如 **progress** 及 **state**，以支援執行緒間相互交流。
- 欲執行簡單的背景工作，可使用 **doInBackground** 方法來在背景執行工作。
- 欲執行中間會產生結果的工作，可用 **publish** 與 **process** 方法將結果顯示在 GUI。
- 欲取消背景執行緒，可用 **cancel** 方法將之取消。



註釋 -- 有關 如何使用 **SwingWorker** 類別的資訊，不在本單元的說明範圍內。欲獲得更多協助，請參考以下 URL：

<http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html>

