第五章
package 與 import 機制

所謂的定律，也不過是目前尚未被人推翻的假設罷了。

前言

許多對 Java 程式設計有興趣的初學者，多半到書店買一本 Java 程式設計的入門書開始他的 Java 程式設計之旅。首先，他可能會到 http://www.javasoft.com 下載最新版本的 Java 2 SDK，將它安裝在自己的電腦上，然後依照書本的指示，用文字編輯器輸入一個名為 HelloWorld.java 的 Java 應用程式原始碼檔。熟悉 Java 的朋友當然可以順利通過編譯並成功地執行 Java 應用程式。但是對初學者來說，當他鍵入:

javac HelloWorld.java

之後可能會編譯成功，產生 HelloWorld.class，但是，接下來當他輸入:

java HelloWorld

時，卻可能發現 java.exe 跑出惱人的訊息:

Exception in thread "main" java.lang.NoClassDefFoundError:HelloWorld

如果是一位對命令列模式(console)沒有太多概念的初學者，可能連編譯這一步都無法跨過，編譯時螢幕上會出現一大堆讓人不知如何下手的的錯誤訊息。

因為這個進入 Java 程式設計領域的重大門檻，不知澆熄了多少 Java 初學者的熱情:連簡單一個 Hello World 的 Java 程式，想看到其執行結果都不是一件簡單的事情了，想當然書本裡頭五花八門的程式設計議題都不再有意義。這兩個門檻不知道阻礙了多少人學習 Java 程式設計。

本章上半部分將假設您是一位第一次使用 Java 的朋友，所以內容的編排都是為了要打破這進入門檻，讓大家可以快快樂樂地學習 Java 程式設計；而下半部分就稍微複雜了點，如果您希望深入了解運作細節，那麼您可以繼續看完後半部分，沒興趣的話略過也無妨。如果您曾試著使用 JBuilder 這種威力強大的開發工具，您一定也會發現如果對於 Java 的 package 與 import 機制沒有妥善的瞭解，那麼開發程式的時候一大堆的錯誤訊息一樣也會搞的您心煩意亂，希望看完本篇之後，可以讓您更輕鬆駕馭像 JBuilder 這種高階開發工具。

初探 package 與 import 機制

進入 Java 程式設計領域的第一個門檻，就是要了解 Java 的 package 與 import 機制，每一位學習 Java 的人都必須對這個機制有切確的了解，底下我們將假設您剛裝好 Java 2 SDK 於 d:\jdk1.3.0_01 目錄之中(**注意! package 與**

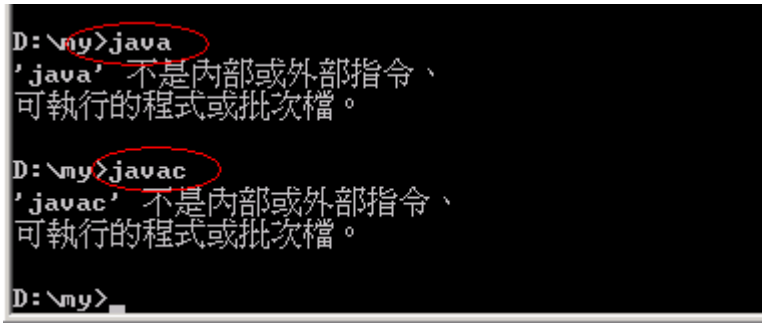
import 機制不會因為您用了新版的 JDK 而改變，如果您用的是 JDK 1.4.x，甚至是以後更新版本的 Java 2 SDK，都會產生與本章相同的結果)，並沒有修改任何的環境變數。接著我們在 D 磁碟機根目錄底下新增一個名為 my 的空目錄 (d:\my)，並以這個空目錄作為我們討論的起始點。

討論一:

首先，請您將目錄切換到 d:\my 底下，請先試著在命令列中輸入

java 或 **javac**

您的螢幕上可能會輸出錯誤訊息如下:



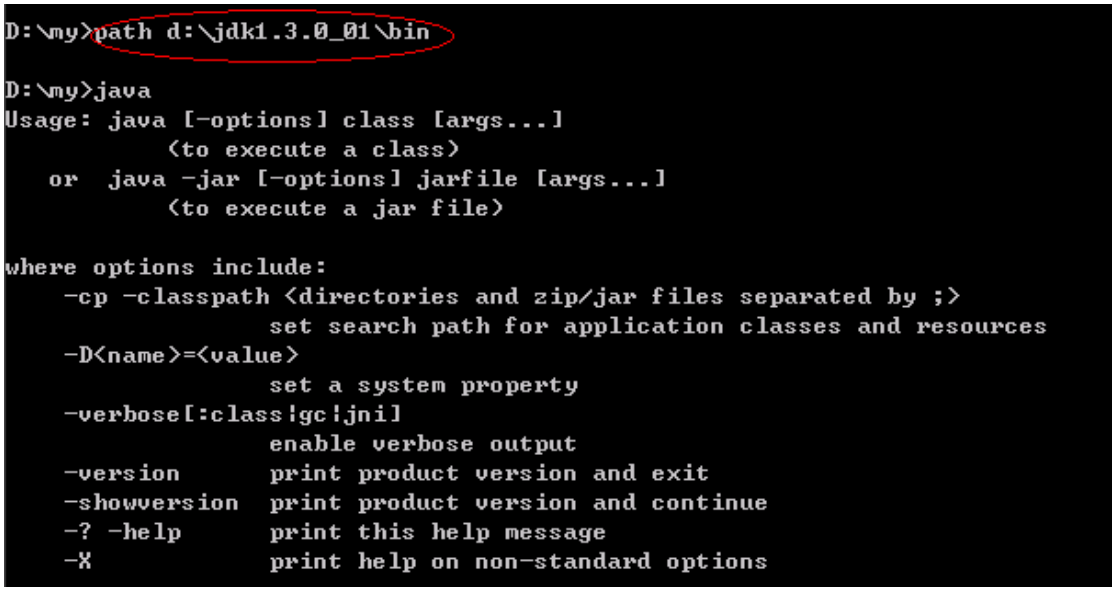
這是因為系統不知道到哪裡去找 java.exe 或 javac.exe 的緣故。所以您可以試著輸入指令:

path d:\jdk1.3.0_01\bin

然後您再重新輸入

java 或 **javac**

就會看到以下畫面:



```
D:\my>javac
Usage: javac <options> <source files>
where possible options include:
  -g                         Generate all debugging info
  -g:none                    Generate no debugging info
  -g:{lines,vars,source}     Generate only some debugging info
  -O                         Optimize; may hinder debugging or enlarge class file

  -nowarn                    Generate no warnings
  -verbose                   Output messages about what the compiler is doing
  -deprecation               Output source locations where deprecated APIs are us
ed
  -classpath <path>          Specify where to find user class files
  -sourcepath <path>         Specify where to find input source files
  -bootclasspath <path>      Override location of bootstrap class files
  -extdirs <dirs>            Override location of installed extensions
  -d <directory>             Specify where to place generated class files
  -encoding <encoding>       Specify character encoding used by source files
  -target <release>          Generate class files for specific VM version


D:\my>_
```

請注意，以上說明只是為了強調 path 環境變數的使用。事實上，幾乎大多數版本的 JDK 都會於安裝時主動在**<Windows 安裝目錄>\system32** 底下複製一份 java.exe，而**<Windows 安裝目錄>\system32** 通常又是 Windows 預設 path 環境變數中的其中一個路徑，所以一般的情況下，都會發生可以執行 java.exe，卻不能執行 javac.exe 的情形。這小小的差異會產生微妙的差別，如果您閱讀過本書第一章，他們之間的差異對您來說已經非常清楚。

接下來，請您在 d:\my 目錄下新增兩個檔案，分別是:

A.java

```
public class A
{
    public static void main(String[] args)
    {
        B b1=new B() ;
        b1.print() ;
    }
}
```

B.java

```
public class B
{
    public void print()
    {
        System.out.println("package test") ;
```

```
    }
}
```

接著請您在命令列輸入:

javac A.java

如果您的程式輸入無誤，那麼您就會在 d:\my 中看到產生了兩個類別檔，分別是 A.class 與 B.class。

請注意，javac.exe 是我們所謂的 Java 編譯器，它具有類似 make 的能力。舉例來說，我們的 A.java 中用到了 B 這個類別，所以編譯器會自動幫您編譯 B.java。反過來說，如果您輸入的指令是 **javac B.java**，由於 B.java 中並沒有用到類別 A，所以編譯器並不會自動幫您編譯 A.java，

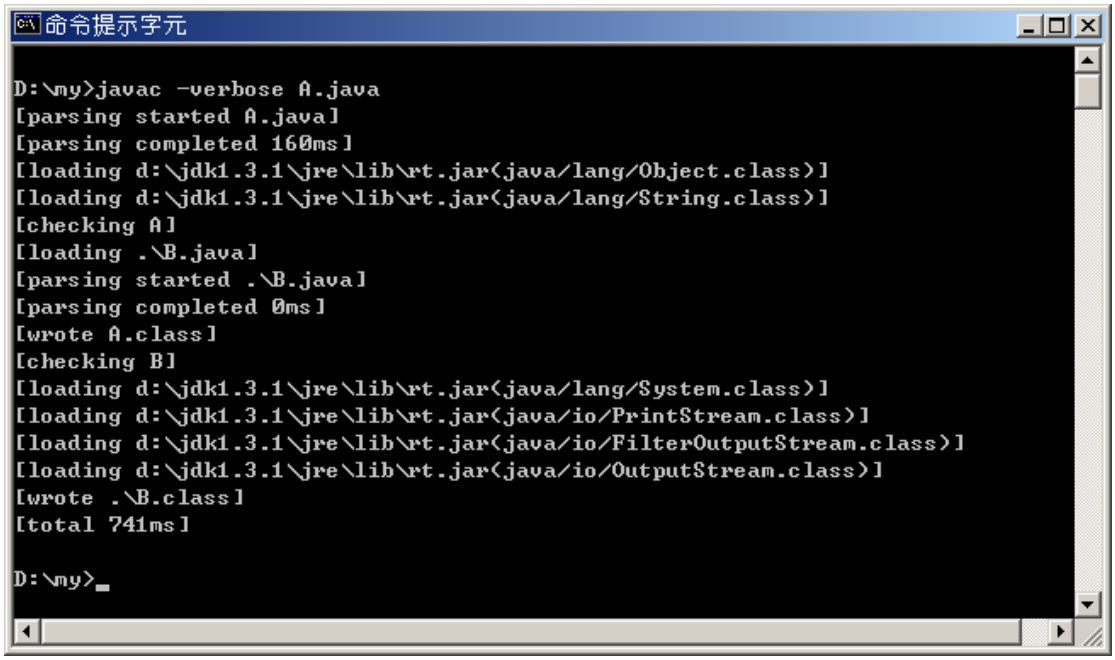
編譯成功之後，請輸入指令:

java A

您會看螢幕上成功地輸出

package test

在此推薦您一個非常好用的選項: **-verbose** 。您可以在 javac.exe 或 java.exe 中使用此選項。他可以讓您更了解編譯和執行過程中 JVM 所做的每件事情。如果您在編譯的時候使用 **–verbose** 選項，結果如下:

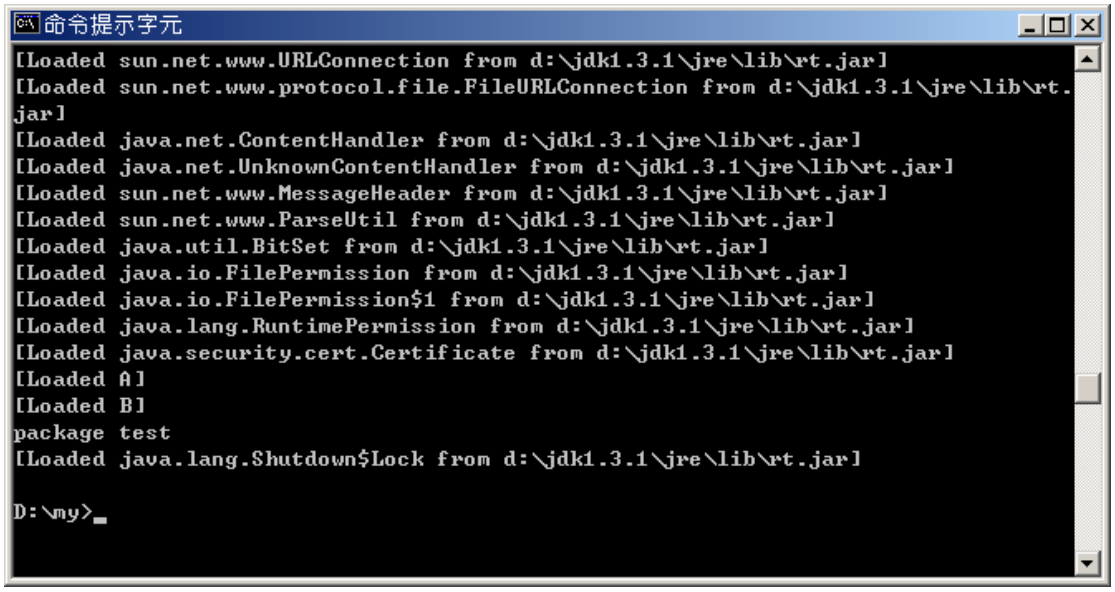


從這個選項，您可以發現，由於編譯器採用了 JDK 所在目錄底下那套 JRE 的緣故，所以編譯時使用<JRE 所在目錄>\lib\rt.jar 裡頭已經編譯好的核心類別函式

庫(第一章的時候我們提過 Java 編譯器也是用 Java 寫成的，您可以試著讓 javac.exe 使用其他 JRE 所在路徑之下的 rt.jar 來進行編譯嗎? 留給大家做個小測驗)。

當您執行 Java 程式時，使用 **–verbose** 選項的結果如下:

省略...



您可以發現，由於 java.exe 根據預設的邏輯而選擇使用了 JDK 所在目錄底下那套 JRE 來執行程式，所以執行時所載入的核心類別都是使用<JRE 所在目錄>\lib\rt.jar 裡頭那些。

您一定覺得上述討論一的內容很簡單，您也可以做到，所以接下來我們要更進一步，探究更複雜的機制。

討論二:

請先刪除討論一中所產生的類別檔，接著修改您的兩個原始碼，使其內容為:

A.java

```
import edu.nctu.* ;
public class A
{
    public static void main(String[] args)
    {
        B b1=new B() ;
        b1.print() ;
```

```
    }
}
```

B.java

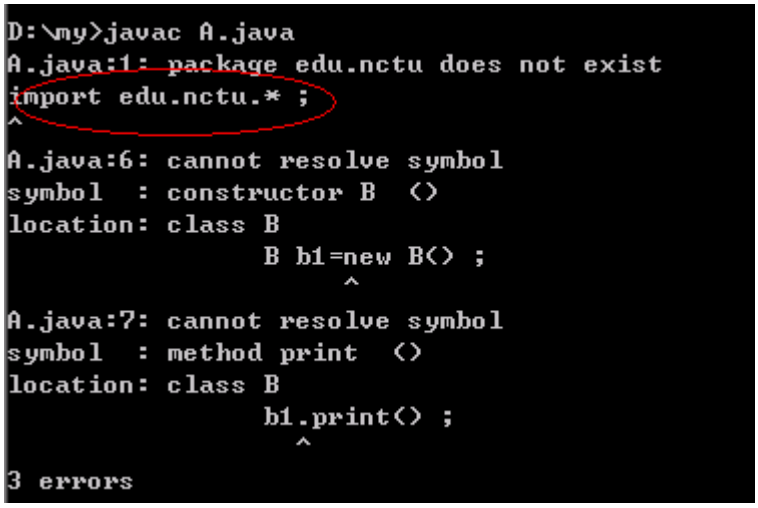
```
package edu.nctu ;
public class B
{
    public void print()
    {
        System.out.println("package test") ;
    }
}
```

接著請您在命令列輸入:

javac A.java

如果您的程式輸入無誤，那麼您會看到螢幕上出現了許多錯誤訊息，主要的錯誤在於

A.java:1: package edu.nctu does not exist



意思是說找不到 edu.nctu 這個 package。其他兩個錯誤都是因為第一個錯誤所衍生。

可是不對呀，按照 Java 的語法，既然 B 類別屬於 edu.nctu，所以我們在 B.java 之中一開始就使用

package edu.nctu ;

而 A 類別用到了 B 類別，所以我們也在 A.java 開頭加上了

import edu.nctu.* ;

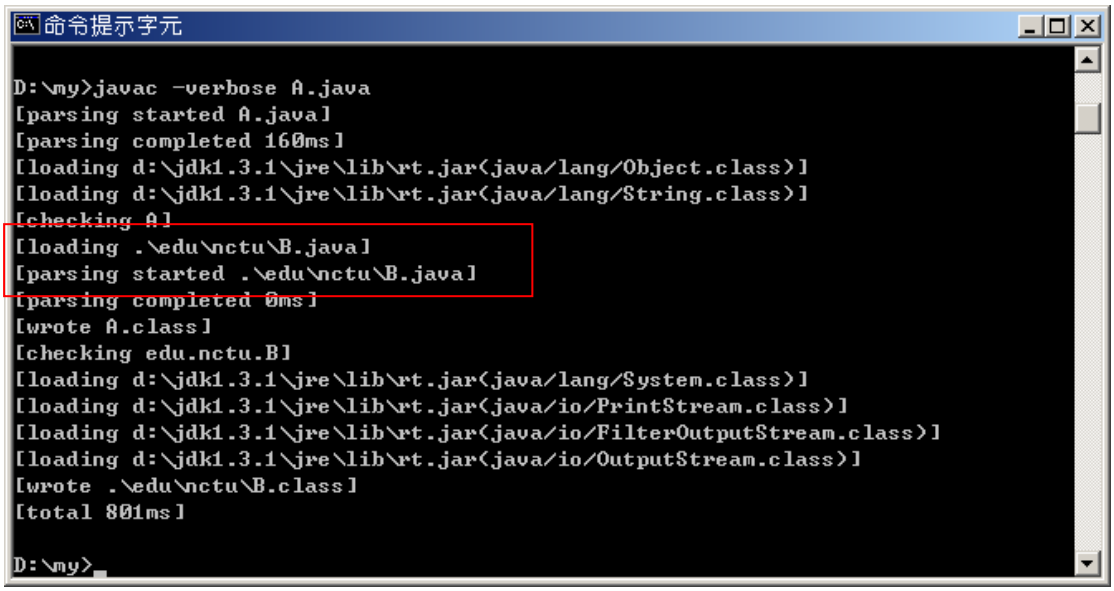
所以這段程式在理論上應該不會發生編譯錯誤才是。

為了解決這個問題，請您在 d:\my 目錄下建立一個名為 edu 的目錄，在 edu 目錄下再建立一個名為 nctu 的子目錄，然後將 B.java **移至**

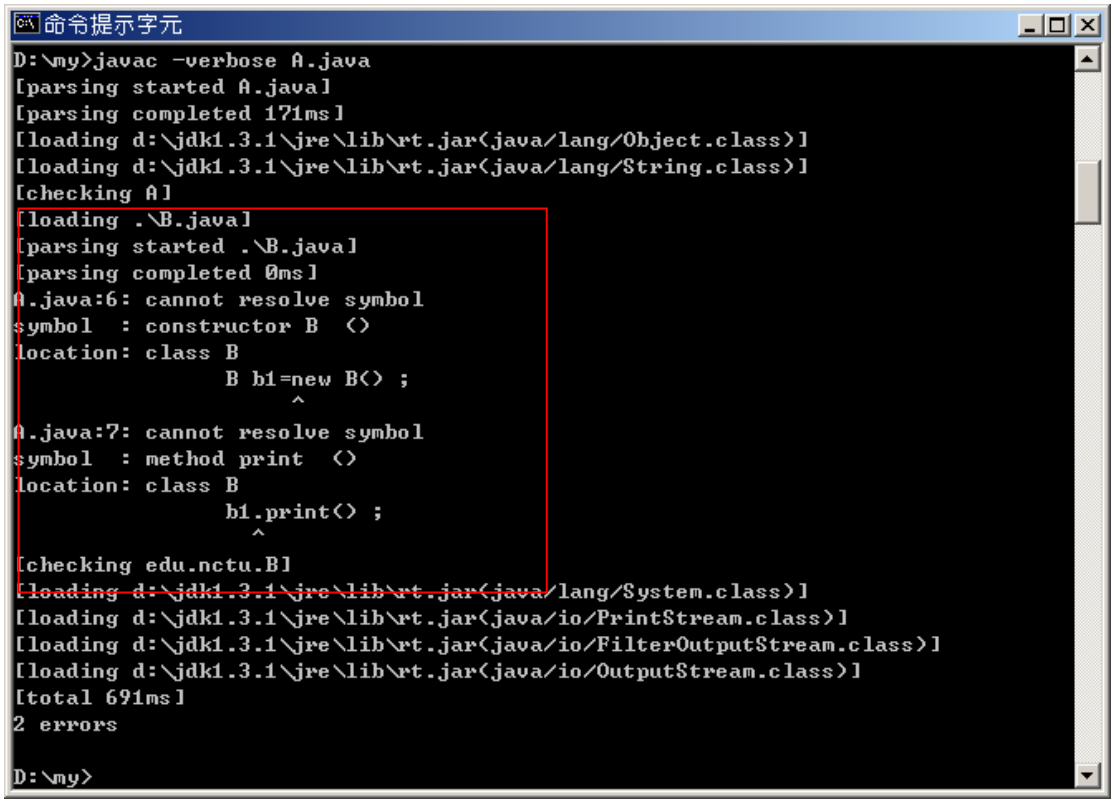
d:\my\edu\nctu 目錄下，請重新執行

javac –verbose A.java

如果操作無誤，那麼您的螢幕上會顯示底下訊息:



注意，如果您在 d:\my 底下仍保有原本的 B.java，則會產生底下錯誤訊息。



從這個輸出您可以發現，編譯器總是先到 A.java 本身所在的路徑中尋找 B.java，雖然編譯器找到了 B.java，可是比對過其 package 宣告之後，認為它應該位於 edu\nctu 目錄下，不該在此目錄下，因此產生錯誤訊息。這個動作在本章後面會有更詳細的解釋。

不管如何，這次終於編譯成功了!! 您會在 d:\my 目錄下找到 A.class，也會在 d:\my\edu\nctu 目錄下找到編譯過的 B.class。編譯成功之後，請輸入指令:

java A

您會看到螢幕上輸出

package test

程式順利地執行了。接這著咱們做個測試，請將 d:\my\edu\nctu 目錄下的 B.class 移除，重新執行

java A

螢幕上會輸出錯誤訊息:

```
D:\my>java A
Exception in thread "main" java.lang.NoClassDefFoundError: edu/nctu/B
        at A.main(A.java:6)
```

意思是說 java.exe 無法在 edu/nctu 這個目錄下找到 B.class。完成了這個實驗之後，請將剛剛移除的 B.class 還原，以便往後的討論。

到此處，我們得到了兩個重要的結論:

結論一:
如果您的類別屬於某個 package，那麼您就應該將它至於該 package 所對應的相對路徑之下。舉例來說，如果您有個類別叫做 C，屬於 xyz.pqr.abc 套件，那麼您就必須建立一個三層的目錄 xyz\pqr\abr，然後將 C.java 或是 C.class 放置到這個目錄下，才能讓 javac.exe 或是 java.exe 順利執行。
其實這裡少說了一個重點，就是這個新建的目錄應該從哪裡開始? 一定要從 d:\my 底下開始建立嗎? 請大家將這個問題保留在心裡，我們將在底下的討論之中為大家說明。

結論二:
當您使用 javac.exe 編譯的時候，類別原始碼的位置一定要根據結論一所說來放置，如果該原始碼出現在不該出現的地方(如上述測試中 B.java 同時存在 d:\my 與 d:\my\edu\nctu 之下)，除了很容易造成混淆不清，而且有時候抓不出編譯為何發生錯誤，因為 javac.exe 輸出的錯誤訊息根本無法改善問題。

在我們做進一步測試討論前，請再做一個測試，請將 d:\my\edu\nctu 目錄下的 B.class 複製到 d:\my 目錄中，執行指令:

java A

您應當還是會看到螢幕上輸出

package test

但是此時如果您重新使用編譯指令

javac A.java

重新編譯 A 類別，螢幕上會出現

bad class file: .\B.class

class file contains wrong class: edu.nctu.B

的錯誤訊息。

```
D:\my>javac A.java
A.java:6: cannot access B
bad class file: .\B.class
class file contains wrong class: edu.nctu.B
Please remove or make sure it appears in the correct subdirectory of the classpa
th.
                B b1=new B() ;
                ^
1 error
```

最後，請您刪掉 d:\my 底下剛剛複製過來的 B.class，也刪除 d:\my\edu\nctu 目錄中的 B.java，也就是讓整個系統中只剩下 d:\my\edu\nctu 目錄中擁有 B.class，然後再使用指令

javac A.java

您一定會發現，除了可以通過編譯之外，也可以順利地執行。

在測試中，我們又得到了兩個結論:

結論三:
編譯時，如果程式裡用到其他的類別，不需要該類別的原始碼也一樣能夠通過編譯。

結論四:
當您使用 javac.exe 編譯程式卻又沒有該類別的原始碼時，類別檔放置的位置應該根據結論一所說的方式放置。如果類別檔出現在不該出現的地方(如上述測試中 B.class 同時存在 d:\my 與 d:\my\edu\nctu 之下)，有很大的可能性會造成難以的編譯錯誤。雖然上述的測試中，使用 java.exe 執行 Java 程式時，類別檔亂放不會造成執行錯誤，但是還是建議您儘量不要這樣做，除了沒有意義之外，這種做法像是一顆不定時炸彈，隨時都有可能造成您的困擾。

討論三:

請先刪除討論二中所產生的所有類別檔，接著請將 d:\my 裡的 edu 目錄連同子目錄全部**移到**(不是複製喔!)d:\底下。然後執行

javac A.java

糟了，螢幕上又出現三個錯誤，意思是說找不到 edu.nctu 這個 package。其他兩個錯誤都是因為第一個錯誤所衍生。

```
D:\my>javac A.java
A.java:1: package edu.nctu does not exist
import edu.nctu.* ;
^
A.java:6: cannot resolve symbol
symbol  : constructor B  ()
location: class B
                B b1=new B() ;
                     ^
A.java:7: cannot resolve symbol
symbol  : method print  ()
location: class B
                b1.print() ;
                  ^
3 errors
```

接著請執行修改過的指令

javac –classpath d:\ A.java

就可以編譯成功，您可以在 d:\my 目錄下找到 A.class，也可以在 d:\edu\nctu 目錄下找到 B.class。

請執行指令

java A

又出現錯誤訊息了，意思是說 java.exe 無法在 edu\nctu 這個目錄下找到 B.class。

```
D:\my>java A
Exception in thread "main" java.lang.NoClassDefFoundError: edu/nctu/B
        at A.main(A.java:6)
```

聰明的你一定想到解決方法了，請執行修改過的指令

java –classpath d:\ A

可式螢幕上仍然出現錯誤訊息，但是這回 java.exe 告訴您他找不到 A.class，而不是找不到 B.class。

```
D:\my>java -classpath d:\ A
Exception in thread "main" java.lang.NoClassDefFoundError: A
```

好吧，那再將指令改成

java –classpath d:\;. A

哈哈!這回終於成功地執行了。

```
D:\my>java -classpath d:\;. A
package test
```

到此，我們歸納出一個新的結論，但是在此之前請回過頭看一次先前的結論一，再回來看新結論:

結論五:

結論一中我們提到，如果您有個類別叫做 C，屬於 xyz.pqr.abc 套件，那麼您就必須建立一個三層的目錄 xyz\pqr\abr，然後將 C.java 或是 C.class 放置到這個目錄下，才能讓 javac.exe 或是 java.exe 順利執行。但是這個新建的目錄應該從哪裡開始呢? 答案是:"可以建立在任何地方"。但是您必須告訴 java.exe 與 javac.exe 到哪裡去找才行，告訴的方式就是利用它們的**-classpath** 選項。在此測試中，我們把 edu\nctu 這個目錄移到 d:\之下，所以我們必須使用指令:

javac –classpath d:\ A.java

與

java –classpath d:\;. A

告訴 java.exe 與 javac.exe 說:"如果你要找尋類別檔，請到**-classpath** 選項後面指定的路徑去找，如果找不到就算了"。

不過這裡又衍生出兩個問題，首先第一個問題是，那麼為什麼之前的指令都不需要額外指定**-classpath** 選項呢?這是因為當您執行 java.exe 和 javac.exe 時，其實他已經自動幫您加了參數，所以不管是討論一或討論二裡所使用的指令，其實執行時的樣子如下:

javac –classpath . A.java

或

java –classpath . A

意思就是說，他們都把當時您所在的目錄當作是**-classpath** 選項的預設參數。那麼您可以發現，如果是 javac.exe 執行時，當時我們所在的路徑是 d:\my，所以很自然地，他會自動到 d:\my 底下的 edu\nctu 目錄尋找需要的檔案，java.exe 也是一樣的道理。

但是，一旦我們搬移了 edu\nctu 目錄之後，這個預設的參數使得 javac.exe 要尋找 d:\my\edu\nctu 底下的 **B.java 或 B.class 來進行編譯時，**發生連 d:\my\edu\nctu 目錄都找不到的情況(因為被我們移走了)，所以自然發生編譯錯誤。因此我們必須使用在指令中加上**–classpath d:**，讓 javac.exe 到 d:\下尋找相對路徑 d:\edu\nctu。執行 java.exe 時之所以也要在指令中加上**–classpath d:**，也是一樣的道理。

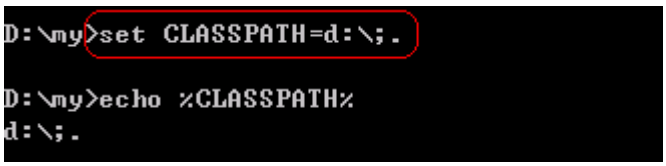
如果您仔細比較兩個修改過的指令，您還是會發現些許的差異，於是引發了第二個問題:為什麼 javac.exe 用的是**–classpath d:**，而 java.exe 如果不用**–classpath d:\;**.而只用**–classpath d:**卻無法執行呢? 我們在第二章曾經提過，-classpath 是用來指引 AppClassLoader 到何處載入我們所需要的類別。在執行時期，主程式 A.class 也是一個類別，所以如果不是預設的情況(預設指向目前所在目錄，即".")，我們一定要向 AppClassLoader 交代清楚所有相關類別檔的所在位置。但是對編譯器來說，A.java 就單純是個檔案，只要在目前目錄之下，編譯器就能找到，基本上和 AppClassLoader 毫無關聯，需要在 -classpath 指定路徑，只是為了**在編譯時期 AppClassLoader 可以幫我們載入**

B.class 或指引編譯器找到 **B.java**，所以必須這樣指定。因此，對 javac.exe 來說，-classpath 有上述兩種作用;然而對於 java.exe 來說，就只有傳遞給 AppClassLoader 一種作用而已。

那麼您一定會問，如果每次編譯會執行都要打那麼長的指令，那豈不是非常麻煩，雖然 java.exe 也提供一個簡化的選項**–cp**(功能同**–classpath**)，但是還是很麻煩。為了解決這個問題，所以設計了一個名為 CLASSPATH 的環境變數，這個環境變數可以省掉您不少麻煩。

您可以在命令列打上

set CLASSPATH=d:\;.



然後您就可以像之前一樣使用指令:

javac A.java

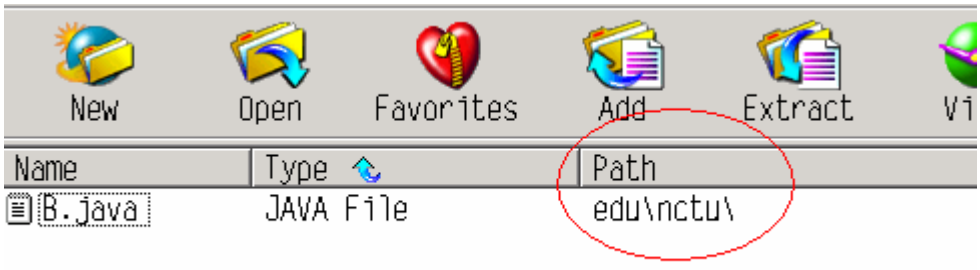
與

java A

完成編譯和執行的工作。但是請注意，如果您設定了環境變數 CLASSPATH，而在使用 java.exe 或 javac.exe 的時候也使用了**–classpath** 選項，那麼環境變數將視為無效，而改為以了**–classpath** 選項所指定的路徑為準。請注意，環境變數 CLASSPATH 與**–classpath** 選項分別所指定的路徑並不會有加成效果。

JAR 檔與 ZIP 檔的效用

如果上述討論都沒有問題了，接下來請利用 Winzip 之類的壓縮工具將 B.java 封裝到 edu.zip 之中，並將 edu.zip 放到 d:\之下。edu.zip 內容如下圖所示:



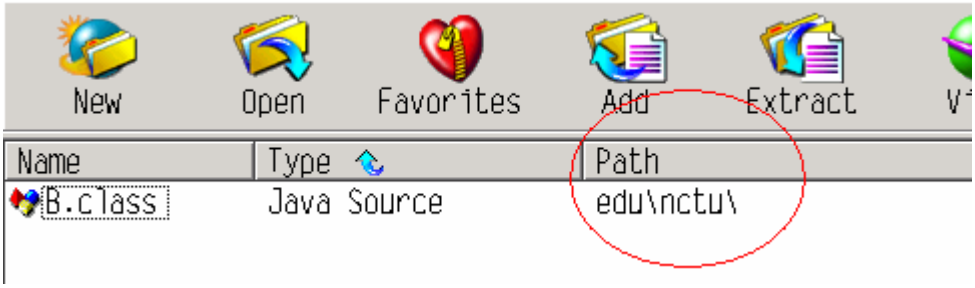
請執行

javac –classpath d:\edu.zip A.java

會出現錯誤訊息如下:

```
D:\my>javac -classpath d:\edu.zip A.java
d:\edu.zip(edu/nctu/B.java):2: class B is public, should be declared in a file n
amed B.java
(source unavailable)
1 error
```

從這裡您可以知道，原始碼檔放在壓縮檔之中是沒有任何效果的。前面既然說過編譯或執行都可以不需要原始碼，所以改將 B.class 封在 edu.zip 之中，edu.zip 的內容變成如下所示:



請重新執行

javac –classpath d:\edu.zip A.java

您會發現可以通過編譯。同時您也可以使用

java –classpath d:\edu.zip;. A

一樣可以成功地執行。

到這裡我們又歸納出一個新結論:

結論六:

使用 ZIP 檔的效果和單純的目錄相同，如果您在**–classpath** 選項指定了目錄，就是告訴 java.exe 和 javac.exe 到該目錄尋找類別檔；如果您在**–classpath** 選項指定了 ZIP 檔的檔名，那麼就是請 java.exe 和 javac.exe 到該壓縮檔中尋找類別檔。請注意，即使是在壓縮檔中，但是該有的相對路徑還是要有，否則 java.exe 和 javac.exe 仍然無法找到他們所需要的類別檔。

當然，在環境變數 CLASSPATH 中也可以指定 ZIP 檔作為其內容。

為何這此我們要提出 ZIP 這種格式的壓縮檔呢? 如果您常常使用別人所開發的套件，您一定會發現別人很喜歡使用 JAR 檔(.jar)這種檔案格式。其實 JAR 檔就是 ZIP 檔。為何大家喜歡使用 JAR 檔的封裝方式呢?這是因為一般別人所開發的套件總是會附帶許多類別檔和資源檔(例如圖形檔)，這麼多檔案全部放在目錄下很容易讓人感到雜亂不堪，如果將這些檔案全部封裝在一個壓所檔之中，不但可以減少檔案大小，也可以讓您的硬碟看起來更精簡。這也是為何每次我們下載了某人所開發的套件時，如果只有一個 JAR 檔，我們就必須在環境變數 CLASSPATH 或**–classpath** 選項中加上這個 JAR 檔的檔名，因為唯有如此，我們才能讓java.exe 和javac.exe 找到我們的程式中所使用到別人套件中類別的類別檔，並成功執行。(在第二章中，我們也曾經提到可以放到**<JRE 所在目錄\lib\ext** 底下，請回頭參閱第二章)

路徑的順序問題

在這個討論中最後要提的是，CLASSPATH 或**–classpath** 選項中所指定的路徑或 JAR 檔(或 ZIP 檔)的先後次序也是有影響的。

舉例來說，我們分別把 B.java 放在 d:\edu\nctu 與 d:\my\edu\nctu 目錄下，然後鍵入指令:

javac –classpath d:\ A.java

您將發現編譯器編譯的是 d:\edu\nctu 目錄下的 B.java。我們如果將指令修改為:

javac –classpath .;d:\ A.java

您將發現編譯器編譯的是 d:\my\edu\nctu 目錄下的 B.java。所以在指定環境變數 CLASSPATH 或**–classpath** 選項時，所給予的目錄名稱或 JAR 檔名稱之順序一定要謹慎。

之所以要特別提醒大家這個問題，是因為常常在寫程式的時候不小心在兩個地方都有相同的套件(這也是筆者本身的切身之痛)，一個比較舊，但指定 classpath 的時候該路徑較前，導致雖然另外一個路徑裡有新版的程式，但是執行或編譯時卻總是跑出舊版程式的執行結果，讓人丈二金剛摸不著腦袋。所以最後還是要提醒您特別注意正視這個問題。

討論四:

請刪除前面討論中製作的所有 JAVA 檔和類別檔，並重新在 d:\my 目錄下產生一個新檔案 A.java，內容如下:

A.java

```
package edu.nctu.mot ;
public class A
{
	public static void main(String[] args)
	{
		System.out.println("I'm A") ;
	}
}
```

在此我們把類別 A 歸屬到 edu.nctu.mot 套件之下。

接著請在命令列輸入指令:

javac A.java

咦? 您會發現竟然可以成功地編譯，這是怎麼一回事呢? 其實這裡的 A.java 和前面的B.java是有些許不同的。雖然A.java屬於edu.nctu.mot套件，而B.java屬於 edu.nctu 套件。但是之前我們編譯 B.java 時候並非直接編譯它，而是透過 javac.exe 類似 make 的機制間接編譯 B.java(因為類別 A 中用到了類別

B)。可是在這裡，我們直接編譯 A.java，沒有透過 make 機制，所以可以編譯成功(意思就是說前面討論中用到的 B.java，即使不放在該放置的目錄中一樣可以成功編譯，只要它自己以及所用到的類別可以在環境變數 CLASSPATH 或**–classpath** 選項指定的位置找到即可)。

好，編譯成功了，現在我們試著執行看看，請輸入:

java A

相信您的螢幕上一定出現了許多錯誤。

```
D:\my>java A
Exception in thread "main" java.lang.NoClassDefFoundError: A (wrong name: edu/nc
tu/mot/A)
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:486)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:11
1)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:56)
        at java.net.URLClassLoader$1.run(URLClassLoader.java:195)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:297)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:286)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:253)
        at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:313)
```

接下來我們應當根據結論一所說，建立一個名為\edu\nctu\mot 的目錄，並將 A.java 至於其中，並刪除 d:\my 裡面的 A.java 和 A.class。然後輸入指令

javac A.java

螢幕上出現錯誤訊息

```
D:\my>javac A.java
error: cannot read: A.java
1 error
```

奇怪了，前面不是說 javac.exe 會內定從目前所在目錄尋找類別的原始碼呢? 這裡怎麼無法運作呢? 其實問題和前面一樣，這是因為現在我們直接要求 javac.exe 編譯 A.java，而非透過 make 機制間接編譯，所以我們的指令無法讓 javac.exe 清楚地知道我們的意圖，所以我們必須更改指令內容，您會想到的指令可能有以下幾種:

1	**javac edu.nctu.mot.A.java**
	`D:\my>javac edu.nctu.mot.A.java`
	`error: cannot read: edu.nctu.mot.A.java`
	`1 error`

2	**javac -classpath .\my\edu\nctu\mot A.java**

	D:\my>javac -classpath .\edu\nctu\mot A.java error: cannot read: A.java 1 error
3	**javac -classpath d:\my\edu\nctu\mot A.java** D:\my>javac -classpath d:\my\edu\nctu\mot A.java error: cannot read: A.java 1 error
4	**javac d:\my\edu\nctu\mot\A.java**

其中，第一種指令是對 Java 比較熟悉的人可能會想到的方法，意義我們會在稍後介紹。第二和第三種指令其實意思是一樣的，只是一個用了相對路徑，一個用了絕對路徑。但是，真正能夠成功編譯 A.java 的卻只有第四種指令。所以從這裡可以得知，如果您的專案中充滿了許多類別，請儘量利用 javac.exe 的 make 機制來自動幫您編譯程式碼，否則可是很辛苦的。

現在我們試著執行，請輸入:

java A

java.exe 說他找不到類別 A。

```
D:\my>java A
Exception in thread "main" java.lang.NoClassDefFoundError: A
```

我們必須更改指令內容，您會想到的指令可能有以下幾種:

1	**java edu.nctu.mot.A** D:\my>java edu.nctu.mot.A I'm A
2	**java -classpath .\edu\nctu\mot A**
3	**java -classpath d:\my\edu\nctu\mot A**
4	**java d:\my\edu\nctu\mot\A**

第二和第三種指令其實意思是一樣的，只是一個用了相對路徑，一個用了絕對路徑，但是，真正能夠執行的卻只有第一種指令。這個指令的意思是告訴 java.exe 說:"請你根據環境變數 CLASSPATH 或**–classpath** 選項指定的位置執行相對路徑 \edu\nctu\mot 之下的 A.class"，前面我們提過，如果您沒有指定環境變數 CLASSPATH 或**–classpath** 選項，預設會用目前所在路徑，所以第一種指令真正執行時應該是:

java –classpath . edu.nctu.mot.A

因為當時我們所在的路徑為 d:\my，所以很自然地 java.exe 會尋找 d:\my\edu\nctu\mot 目錄中的 A.class 來執行。

根據結論五所說，如果你將指令改成

java –classpath d:\ edu.nctu.mot.A

那麼就無法執行了，除非您在 d:\edu\nctu\mot 裡也有 A.class。

以上做了那麼多測試，歸納了這幾點結論，最後請大家在利用 package 與 import 機制的時候務必做到下列幾項工作:

1. 將 JAVA 檔和類別檔確實安置在其所歸屬之 package 所對應的相對路徑下。
2. 不管使用 java.exe 或 javac.exe，最好明確指定**–classpath** 選項，如果怕麻煩，使用環境變數 CLASSPATH 可以省去您輸入額外指令的時間。請注意在他們所指定的路徑或 JAR 檔中是否存有 package 名稱和類別名稱相同的類別，因為名稱上的衝突很容易引起混淆。

只要確實做到上述事項，就可以減少螢幕上大量與實際問題不相干的錯誤訊息，並確保您在 Java 程式設計的旅途中順利航行。

在此要提醒大家，到此為止我們所舉的例子都是使用 java.exe 和 jvac.exe 的測試結果，但是在實際生活中，**只要和 Java 程式語言相關的工具，都會和 package 與 import 機制息息相關**。比方說如果您開發 MIDlet，必定會用到 preverify.exe；如果要使用 JNI，您就會用到 javah.exe，使用這些工具時如果您沒有遵循 package 與 import 機制的運作方式，可是無法得到您要的結果。

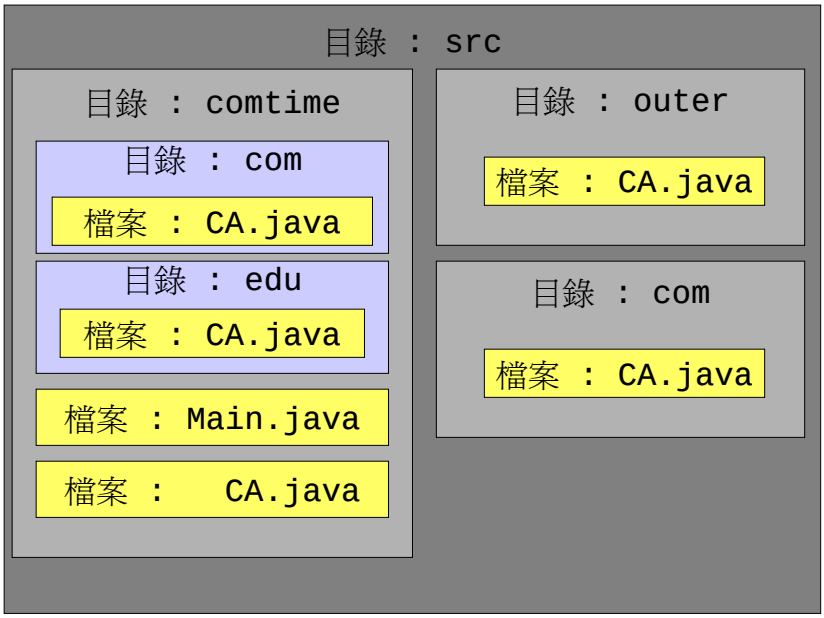
深入 package 與 import 機制

在本章前半個部分，筆者向大家說明了初學者進入 Java 程式設計領域常見的重大門檻 - package 與 import 機制。相信大家已經可以自行處理 Class Not Found 這一類 Exception 了。但是，前半個部分所使用的討論方式只是工程上的結果，那麼原理呢? 只要理解了這些錯誤之所以出現的理由，就可以知道編譯之所以成功或錯誤的原因。因此，本章下半部要用理論的方式為大家說明 package 與 import 機制運作的秘密。

接下來的討論都是假設你遵守「每一個類別都會根據其所屬的 package 放在正確的相對路徑上」。從前半個部分的討論我們可以知道，如果我們不把類別放在正確的相對位置，則會產生許多與問題本身不相干的錯誤訊息，這這部分涉及到 Java 編譯器自己本身對錯誤的處理方式，所以我們不加以討論。

編譯時期(Compile-time)的 Package 運作機制

在講解編譯時期的 Package 運作機制之前，首先我們要先製作一個完整的範例，我們測試所使用的目錄結構如下圖所示:



在我們的測試範例之中，我們總共有六個檔案，他們的內容分別如下:

src\comtime\Main.java

```
public class Main
{
    public static void main(String args[])
    {
        CA ca = new CA() ;
    }
}
```

src\comtime\CA.java

```
public class CA
{
}
```

src\comtime\edu\CA.java

```
package edu;
public class CA
{
}
```

src\comtime\com\CA.java

```
package com;
```

```
public class CA
{
}
```

src\outer\CA.java

```
package outer;
public class CA
{
}
```

src\com\CA.java

```
package com;
public class CA
{
}
```

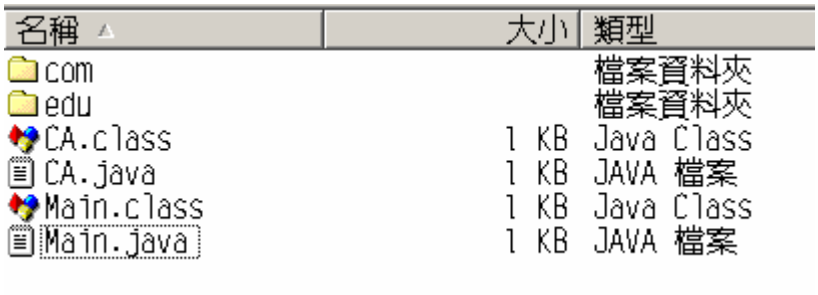
接下來，我們將使用 src\comtime 作為我們測試的根目錄(意即:直接放置在此目錄之下的類別可以不需屬於任何 package，而屬於任何 package 的類別將可以以此目錄做相對參考點，根據自己所屬的 package 產生相對應的目錄來放置自己)，因此請將提示符號切換到 src\comtime 目錄，並執行

javac Main.java ➔指令(1)

```
D:\src\comtime>javac Main.java

D:\src\comtime>
```

然後你將發現不屬於任何 package 的兩個類別分別被編譯成類別檔:



接下來，請將 Main.java 的內容修改成:

src\comtime\Main.java

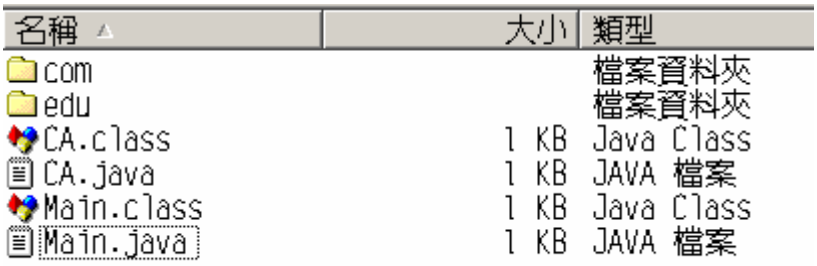
```
import com.*;
import edu.*;
public class Main
{
```

```
    public static void main(String args[])
    {
        CA ca = new CA() ;
    }
}
```

重新執行

javac Main.java ➔指令(2)

你會發現與使用指令(1)的測試結果相同，不屬於任何 package 的兩個類別被編譯成類別檔:



從這兩個簡單的測試大家可以發現，**不管你有沒有使用 import 指令，存在目前目錄下的類別都會被編譯器優先採用，只要它不屬於任何 package**。這是因為編譯器總是先假設您所輸入的類別名稱就是該類別的全名(不屬於任何 package)，然後從-classpath 所指定的路徑中搜尋屬於該類別的.java 檔或.class 檔。從這裡我們可以知道 default package 的角色非常特殊。

接下來，請將 src\comtime\CA.java 改名成 NU.java，代表它不再為我們所使用。

請重新執行

javac Main.java ➔指令(3)

你會發現底下錯誤訊息:

```
D:\src\comtime>javac Main.java
Main.java:7: reference to CA is ambiguous, both class edu.CA in edu and class co
m.CA in com match
                CA ca = new CA() ;
                ^
Main.java:7: reference to CA is ambiguous, both class edu.CA in edu and class co
m.CA in com match
                CA ca = new CA() ;
                        ^
2 errors
```

這個錯誤訊息說明了編譯器產生疑惑，因為 com 與 edu 這兩個 package 底下都有名為 CA 的類別，他不知道該用哪一個。解決的辦法就是明確地告訴編譯器我們要哪個 package 底下的 CA，解決方法有兩種:

1. 在 import 處明確宣告，也就是把 Main.java 改成

src\comtime\Main.java

```
import com.CA;
import edu.*;
public class Main
{
    public static void main(String args[])
    {
        CA ca = new CA() ;
    }
}
```

或者

2. 引用時詳細指定該類別的全名(即 套件名稱.類別名稱 的組合)

src\comtime\Main.java

```
import com.*;
import edu.*;
public class Main
{
    public static void main(String args[])
    {
        com.CA ca = new com.CA() ;
    }
}
```

假設我們要使用的是 com.CA 這個類別，只要利用這兩種方法的其中一種，編譯器就可以明確地知道我們所希望使用的類別，所以當你選用其中一種方法來修改 Main.java 之後，你一定可以發現 src\com\CA.java 被編譯成類別檔了。

好的，接下來我們再將 Main.java 修改成

src\comtime\Main.java

```
import com.*;
import edu.*;
import outer.*;
public class Main
{
    public static void main(String args[])
    {
        CA ca = new CA() ;
    }
}
```

執行

javac Main.java ➔指令(4)

之後，你會看到其中一項錯誤訊息:

```
D:\src\comtime>javac Main.java
Main.java:3: package outer does not exist
import outer.*;
^
Main.java:8: reference to CA is ambiguous, both class edu.CA in edu and class co
m.CA in com match
                CA ca = new CA() ;
                ^
Main.java:8: reference to CA is ambiguous, both class edu.CA in edu and class co
m.CA in com match
                CA ca = new CA() ;
                        ^
3 errors
```

框住的地方告訴我們編譯器找不到 outer 這個套件，為了解決這個問題，我們將指令改成

javac –classpath .. Main.java ➔指令(5)

結果螢幕上還是出現錯誤訊息:

```
D:\src\comtime>javac -classpath .. Main.java
Main.java:2: package edu does not exist
import edu.*;
^
Main.java:8: reference to CA is ambiguous, both class outer.CA in outer and clas
s com.CA in com match
                CA ca = new CA() ;
                ^
Main.java:8: reference to CA is ambiguous, both class outer.CA in outer and clas
s com.CA in com match
                CA ca = new CA() ;
                        ^
3 errors
```

這次編譯器竟然告訴我們找不到 edu 這個 package，我們只好再將指令改成

javac –classpath ..;. Main.java ➔指令(6)

這次不再出現找不到 package 的錯誤訊息了，而是編譯器再次告訴我們他不知道該採用哪一個 package 底下的 CA:

```
D:\src\comtime>javac -classpath ..;. Main.java
Main.java:8: reference to CA is ambiguous, both class outer.CA in outer and clas
s edu.CA in edu match
                CA ca = new CA() ;
                ^
Main.java:8: reference to CA is ambiguous, both class outer.CA in outer and clas
s edu.CA in edu match
                CA ca = new CA() ;
                        ^
2 errors
```

解決的方法我們在前面已經提過了。假設我們把 Main.java 改成

src\comtime\Main.java
import com.CA; import edu.*;

```
import outer.*;
public class Main
{
    public static void main(String args[])
    {
        CA ca = new CA() ;
    }
}
```

重新使用指令

javac –classpath ..;. Main.java ➔指令(7)

來編譯那麼你會發現什麼結果呢?你將發現 src\com 底下的 CA.java 被編譯了。

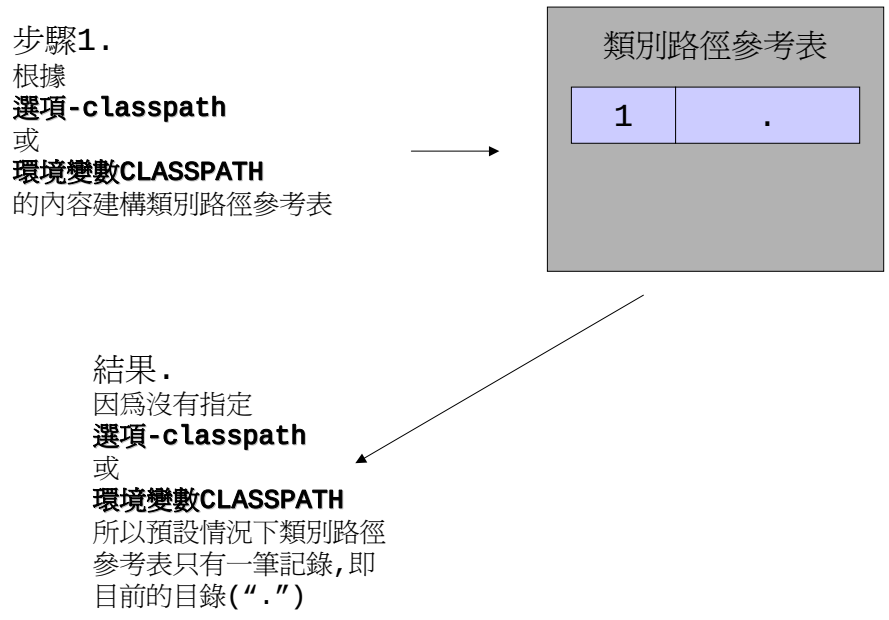
如果我們把指令改成:

javac –classpath .;.. Main.java ➔指令(8)

你將發現 src\comtime\com 底下的 CA.java 被編譯了。

從上面的測試結果，揭露了 java 編譯器的運作情形，當 java 編譯器開始編譯某個類別的原始碼時，首先他會作一件事情，就是建構「類別路徑參考表」，建構邏輯我們用下面幾張圖來表示:

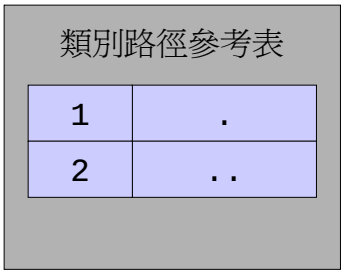
圖:使用指令(3)的時候，建構「類別路徑參考表」的程序



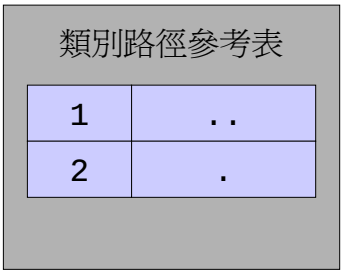
請注意，環境變數 CLASSPATH 的內容會被選項-classpath 所覆蓋，沒有加成效果。也就是說，只有其中一個會被拿來當作類別路徑參考表的內容。所以當我們使用指令(7)或指令(8)的時候，結果如下圖所示:

圖:使用指令(7)與使用指令(8)的時候，「類別路徑參考表」的內容

使用指令(7)的結果

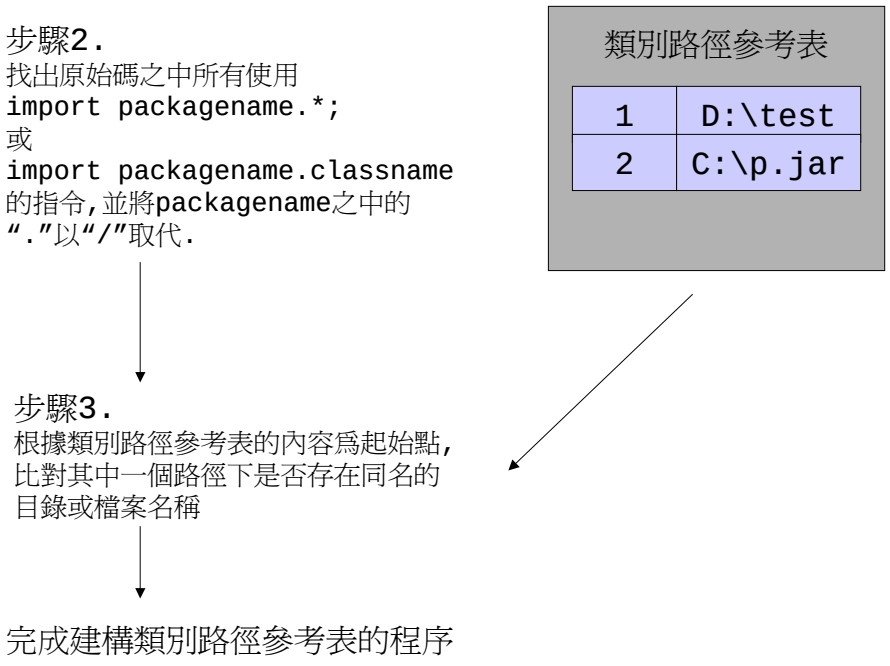


使用指令(8)的結果



當編譯器將類別路徑參考表建好之後，接著編譯器要確定它可以利用類別參考表裡的資料作為**相對起始路徑**，找到所有用到的 package，其流程如下圖所示:

圖:使用「類別路徑參考表」確認 package 之存在的流程。



以上面這張圖來說，這是假設我們用的指令是:

javac –classpath d:\test;c:\p.jar test.java

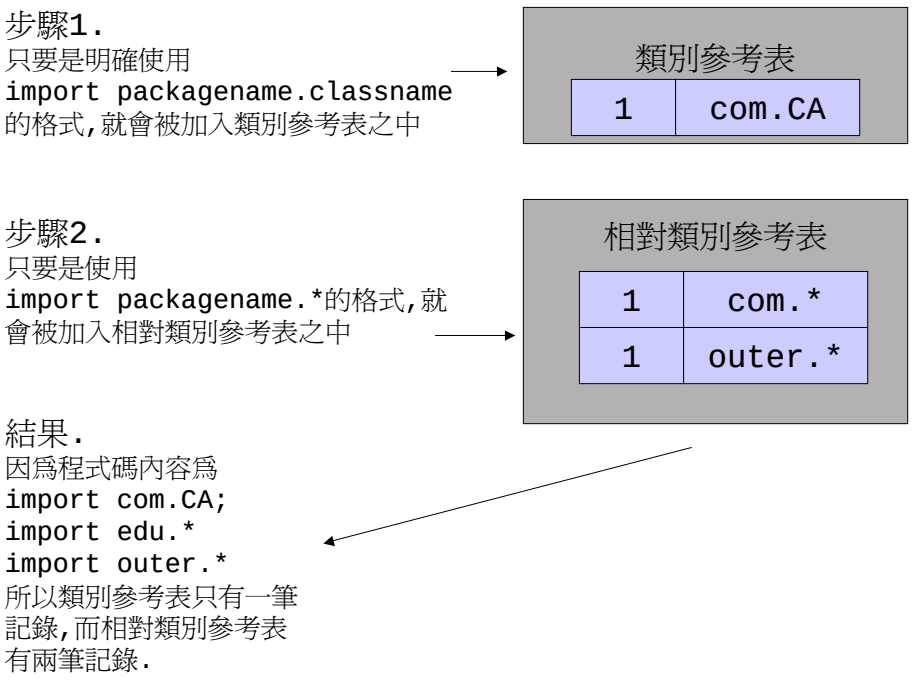
所以使得類別路徑參考表之中有兩筆資料，接著，假設我們的 test.java 之中有 import A.B.* ;

import C.D ;
那麼編譯器就會先看看 d:\test 目錄底下使否存在有 A\B 目錄，也就是說編譯器會檢查 d:\test\A\B 究竟是否存在，如果找不到，它會試著再找看看 c:\p.jar 之中是否存在 A\B 這個目錄，如果都沒找到，就會發出 package A.B does not exist 的訊息。同理，編譯器也會試著尋找是否存在 d:\test\C\D.class 或 d:\test\C\D.java(如果找不到就進入 c:\p.jar 裡頭找)，並確定他們所屬的 package 名稱無誤，也確保他們為 public(如此才能讓其他 package 的類別所存取，且檔名一定與類別名稱相同)，如果沒找到就會發出 cannot resolve symbol 的錯誤訊息。

從這裡我們回頭看指令(4)，為何它會說 package outer does not exist? 因為指令中根本沒有指定選項-classpath，我們也沒有設定環境變數 CLASSPATH，所以類別路徑參考表只有預設的"."，因此編譯器根本無法在 src\comtime 之下找到 outer 這個子目錄。同理，回頭看看指令(5)，因為設定了選項-classpath，所以所以類別路徑參考表只有".."，對 src\comtime 來說，其父目錄即 src，而編譯器在 src 目錄之下根本找不到 edu 這個子目錄，所以自然發出 package edu does not exist 的錯誤訊息。所以要讓編譯器**至少在建構類別路徑參考表的過程沒有問題**，我們就必須使用指令(6)才行。

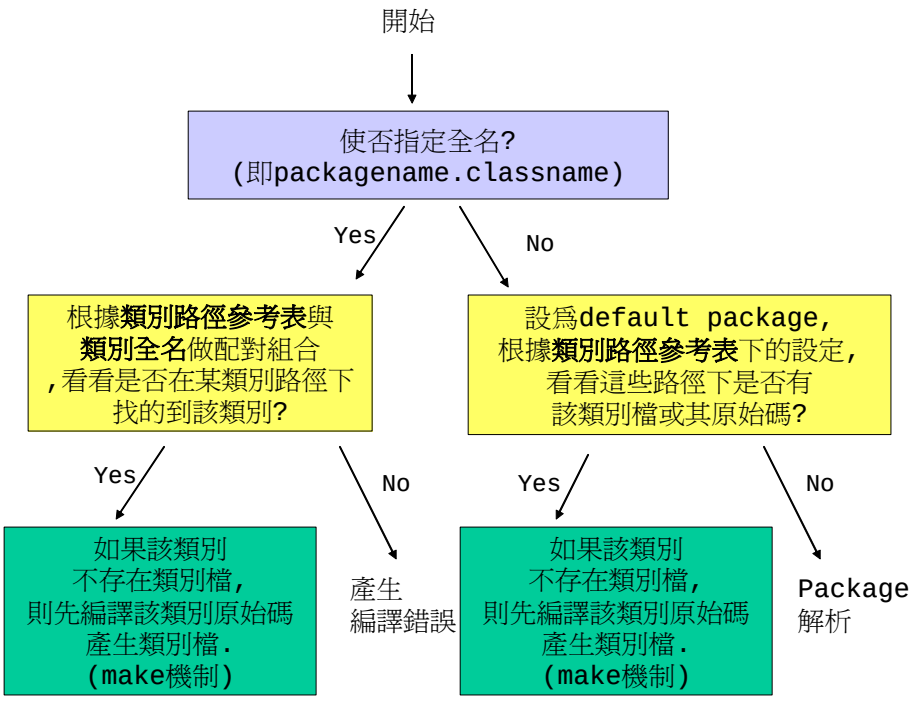
根據上述討論，編譯器會完成一張名為「類別參考表」與「相對類別參考表」的資料結構，如下圖所示

圖:使用指令(7)的時候，建構「類別參考表」的程序



最後，編譯器必須開始解析程式碼裡頭所有的類別名稱，其邏輯如下:

圖:類別名稱解析的程序

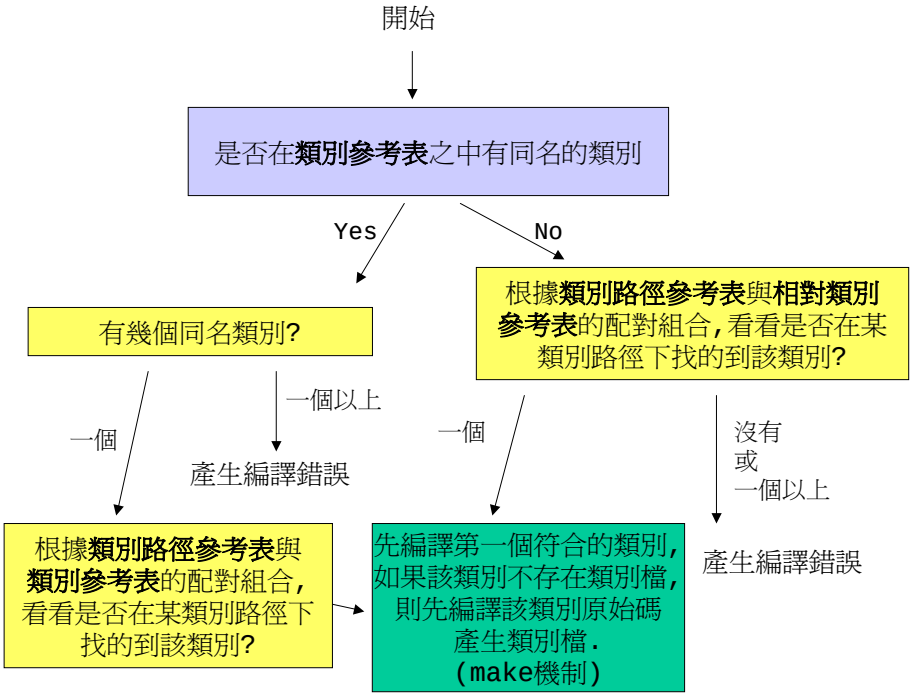


由上面這張圖，我們可以解釋指令(1)(2)的結果，同時也可以解釋當我們使用指令(3)之後，為了解決了類別名稱曖昧不明的情形，而我們使用程式碼

```
com.CA ca = new com.CA() ;
```

之後的編譯結果。但是要解釋指令(7)(8)的結果，就必須用下圖來說明:

圖:package 解析的程序(接上圖)



從上圖我們可以解釋，可以解釋當我們使用指令(3)之後，為了解決了類別

名稱曖昧不明的情形，而我們使用程式碼

import com.CA ;

之後的編譯結果。

當我們使用指令(7)的時候，由於類別參考表之中只有 com.CA 一個名為 CA 的類別，所以接下來編譯器根據相對類別參考表的內容，先合成..\com\CA.class 與..\com\CA.java，所以我們才會看到 src\com 底下的 CA 被編譯。同理，如果改用指令(8)，由於會合成.\com\CA.class 與.\com\CA.java，所一才導致 src\comtime\com 底下的 CA.java 被編譯。

上面這張圖還可以解釋指令(6)之所以發出錯誤訊息的原因，也同樣地可以解釋如果我們的原始碼為:

src\comtime\Main.java

```
import com.CA;
import edu.CA;
import outer.*;
public class Main
{
    public static void main()
    {
        CA ca = new CA() ;
    }
}
```

編譯器一樣發出錯誤訊息

```
D:\src\comtime>javac -classpath ..;. Main.java
Main.java:8: reference to CA is ambiguous, both class edu.CA in edu and class co
m.CA in com match
                CA ca = new CA() ;
                ^
Main.java:8: reference to CA is ambiguous, both class edu.CA in edu and class co
m.CA in com match
                CA ca = new CA() ;
                        ^
2 errors
```

的原因。

從上面所討論的 Java 編譯器運作邏輯，亦對兩個一般人常常有的誤解做了解釋:

1. import 和 C/C++裡的 include 同意義。
2. 在 import 裡使用萬用字元，如 import x.y.* ，會讓編譯器效率減低。

其中，第一個誤解也常讓很多程式設計師覺得不解，為什麼在 Java 之中無法使用

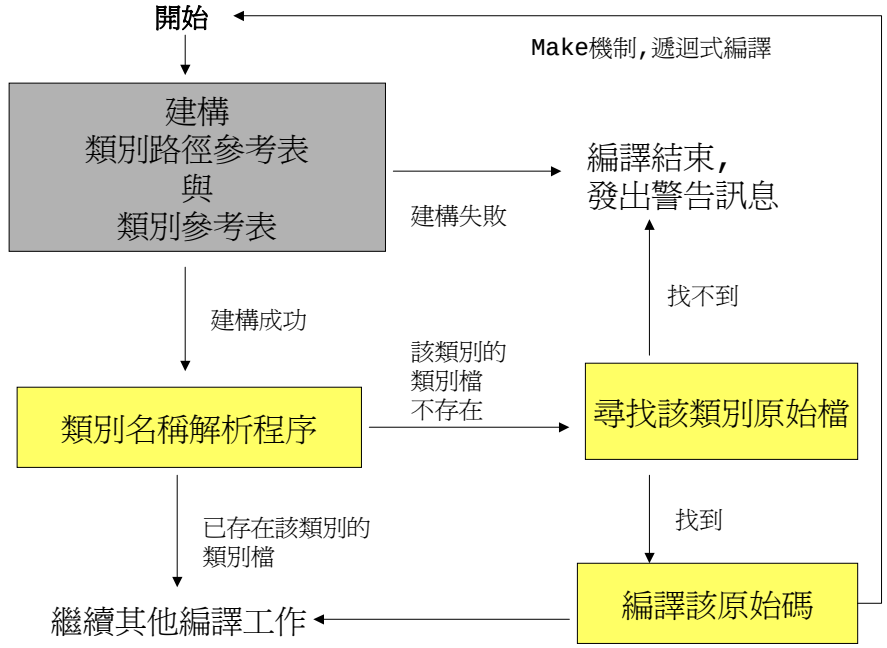
import java.*.*

這種有一個以上萬用字元的語法?

由上面的討論你可以知道，在 Java 裡頭，import 應該和 C/C++裡的 namespace 等價才對，而非一般所說的 include。
假設你有兩個類別，分別是 X.A 與 X.B.C，你必須分別
import X.* ;
import X.B.* ;
才行，因為這兩個指令會讓編譯器分別在類別路徑參考表之中建立兩筆資料，如此一來當編譯器將類別名稱與類別路徑參考表裡的資料作合成的時候，才能讓你的程式:
C c1 = new C();
通過編譯，如果你只有
import X.* ;
則編譯器回發出錯誤訊息，說它無法解析類別 C。也就是說，類別路徑參考表裡的資料，只是單純拿來合成，讓編譯器可以很快地定位到類別檔或原始碼的位置，並不會遞迴式地幫我們自動搜尋該路徑底下其他的類別，所以不管在 import 處使用完整的類別名稱或萬用字元，即使原始碼會使得類別路徑參考表和相對類別參考表很大很大，但是 javac.exe 內部在處理這些資料表格時也是採用 Hash Table 的設計，所以對編譯速度的影響幾乎微乎其微。

好的，到這裡為止，我們已經將整個編譯機制前半部關於 package 與 import 機制的部分向大家解釋清楚了，最後我們把整個編譯程序之中與 package 與 import 機制相關的部分用一張完整的圖做表示:

圖:整個編譯程序與 package 相關的步驟解析



其實類別名稱解析程序之中，還有一個重要的步驟，在圖中並沒有說明，就是**當編譯器找到該類別的類別檔或原始碼之後，都會加以驗證他們使否真的屬於這個 package，如果有問題，編譯器一樣會發出錯誤訊息;但是如果並非透過 make(自動解析名稱，自動編譯所需類別)機制來編譯程式，是不會做這項驗證工作的**。這也是為什麼筆者再三強調每一個類別都要根據其所屬的 package 放在正確的相對路徑之下，否則會導致編譯器產生許多難以理解的編譯錯誤。

從這裡我們也可以發現，如果我們不夠透過 make 機制，而直接切到 src\com 底下直接打指令

javac CA.java

也可以編譯成功，而且就算 CA.java 的內容為

src\com\CA.java

```
package test;
public class CA
{
}
```

也就是說所屬 package 和所在路徑不一致的情況下，一樣可以編譯成功，仔細回頭看看之前所有的示意圖，我們將可以發現這段程式碼並不會引發任何 package 與 import 機制的相對應動作，所以可以編譯成功。也因為如此，請大家儘量讓整個應用程式所用到的類別都可以透過 make 機制來編譯，否則一個一個手動編譯不但辛苦，也會有潛在的錯誤發生。

Java 的動態連結本質

用過 Java 的朋友都知道，Java 會將每個 class 宣告編譯成一個附檔名為.class 的檔案。不管你在同一個原始碼(.java)之使用了幾個類別宣告，他們都會一一編譯成.class 檔，即使是內部內別、匿名類別都一樣，舉例來說:

MultiClass.java

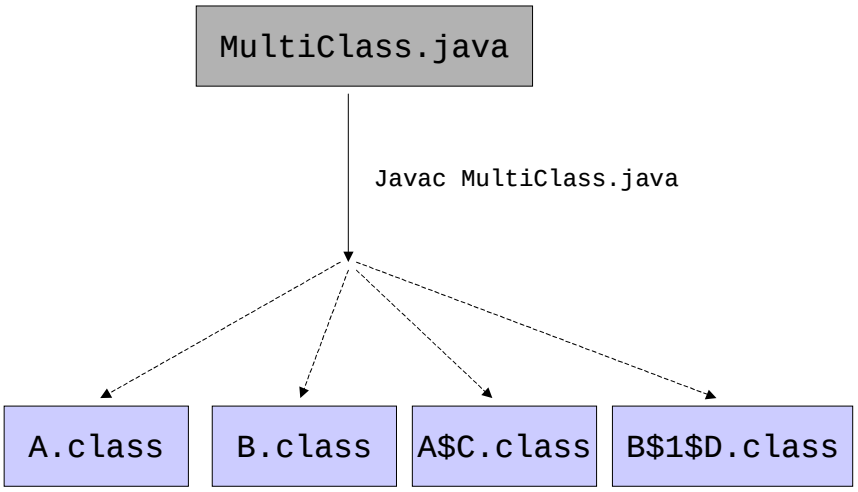
```
class A
{
    class C
    {
    }
}

class B
{
    public void test()
    {
```

```
        class D
        {
        }
    }
}
```

當你編譯 MultiClass.java 之後，你會發現目錄下出現了底下幾個類別檔，他們分別是:

A.class
A$C.class
B.class
B1D.class



這種方式與傳統的程式語言有很大的不同，一般的程式語言在最後都是將所有的程式碼 static link 至同一個執行檔(.exe)之中，如果要做到動態抽換功能模組的的話，就必須借助動態聯結函式庫(Windows 上為 DLL，UNIX 上為 share object)。但是，在 Java 之中，每一個類別所構成的類別檔我們都能將它視為動態聯結函式庫。

好，假設程式碼如下所示，

src\comtime\Main.java

```
import com.CA;
import edu.*;
import outer.*;
public class Main
{
```

```
    public static void main(String args[])
    {
        CA ca = new CA() ;
    }
}
```

src\comtime\com\CA.java

```
package com;
public class CA
{
}
```

而各位執行了上述的指令(8)，各位會看到編譯之後產生了 src\ccomtime\Main.class 與 src\comtime\com\CA.class。接著我們執行指令:

java Main ➔指令(9)

程式將成功地執行完畢，螢幕上沒有任何錯誤訊息，但也沒有其他訊息，

於是我們將 CA.java 改成:

src\comtime\com\CA.java

```
package com;
public class CA
{
    public CA()
    {
        System.out.println("New CA") ;
    }
}
```

並手動切換到 src\comtime\com\CA.java 之下編譯 CA.java，再重新執行指令(9)，結果如下圖所示:

```
D:\src\comtime>cd com

D:\src\comtime\com>javac CA.java

D:\src\comtime\com>cd ..

D:\src\comtime>java Main
New CA

D:\src\comtime>
```

根據之前我們討論的，我們可以保證 Main.java 並沒有被重新編譯，只有 CA.java 被我們重新編譯而已，而如各位所見，有了新的輸出結果，足以證明 Java 動態連結的本質。如果各位需要更細部的資訊，請嘗試著使用:

java –verbose Main ➔指令(10)

這個部分輸出非常多訊息，我們只取其中一個部分讓大家看:

```
[Loaded java.util.jar.Manifest$FastInputStream from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded java.util.jar.Attributes$Name from d:\jdk1.3.1\jre\lib\rt.jar]
[Loaded com.CA]
New CA
[Loaded java.lang.Shutdown$Lock from d:\jdk1.3.1\jre\lib\rt.jar]

D:\src\comtime>
```

螢幕上會告訴我們系統究竟到哪裡載入了什麼類別，聰明的各位，也看出了這些輸出所代表的意函嗎?

執行時期(Run-time)的 Package 運作機制

根據上述所描述的 Java 動態連結本質，那麼，大家一定很疑惑，究竟整個系統是根據什麼樣的原則來載入適當的類別檔呢?

首先我們先來看看前面兩個類別編譯之後產生的類別檔之內容:

src\comtime\Main.class 的內容:

```
CA FE BA BE 00 03 00 2D 00 12 0A 00 05 00 0E 07 ; ■  ?..-........
00 0F 0A 00 02 00 0E 07 00 10 07 00 11 01 00 06 ; ................
3C 69 6E 69 74 3E 01 00 03 28 29 56 01 00 04 43 ; <init>...()V...C
6F 64 65 01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 ; ode...LineNumber
54 61 62 6C 65 01 00 04 6D 61 69 6E 01 00 16 28 ; Table...main...(
5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 ; [Ljava/lang/Stri
6E 67 3B 29 56 01 00 0A 53 6F 75 72 63 65 46 69 ; ng;)V...SourceFi
6C 65 01 00 09 4D 61 69 6E 2E 6A 61 76 61 0C 00 ; le...Main.java..
06 00 07 01 00 06 63 6F 6D 2F 43 41 01 00 04 4D ; ......com/CA...M
61 69 6E 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F ; ain...java/lang/
4F 62 6A 65 63 74 00 21 00 04 00 05 00 00 00 00 ; Object.!........
00 02 00 01 00 06 00 07 00 01 00 08 00 00 00 1D ; ................
00 01 00 01 00 00 00 05 2A B7 00 01 B1 00 00 00 ; ........*?.?..
01 00 09 00 00 00 06 00 01 00 00 00 04 00 09 00 ; ................
0A 00 0B 00 01 00 08 00 00 00 25 00 02 00 02 00 ; ..........%.....
00 00 09 BB 00 02 59 B7 00 03 4C B1 00 00 00 01 ; ...?.Y?.L?...
00 09 00 00 00 0A 00 02 00 00 00 08 00 08 00 09 ; ................
00 01 00 0C 00 00 00 02 00 0D                   ; ..........
```

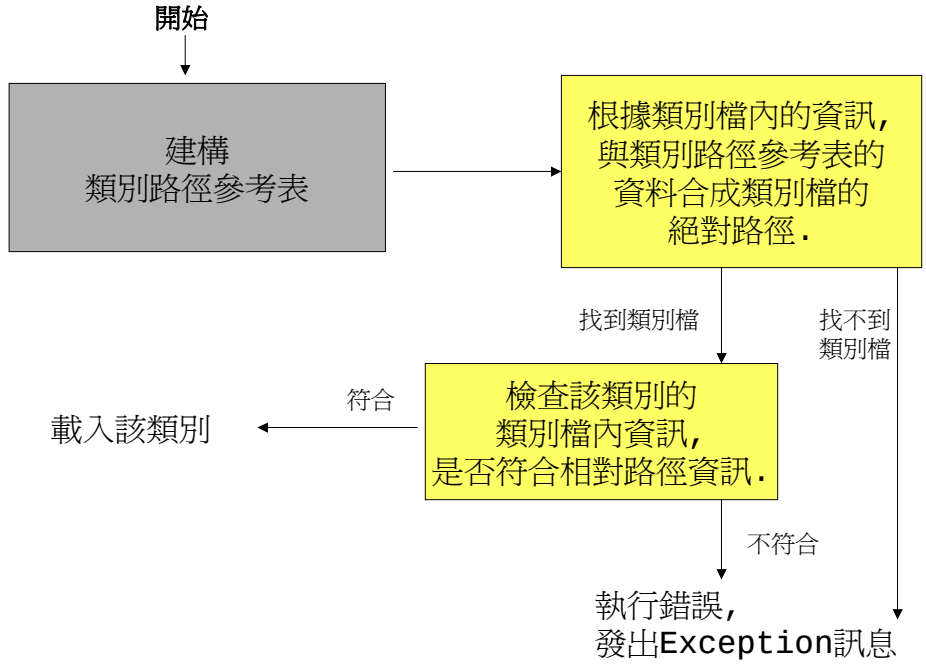
src\comtime\com\CA.class 的內容

```
CA FE BA BE 00 03 00 2D 00 1B 0A 00 06 00 0D 09 ; ■  ?..-........
00 0E 00 0F 08 00 10 0A 00 11 00 12 07 00 13 07 ; ................
00 14 01 00 06 3C 69 6E 69 74 3E 01 00 03 28 29 ; .....<init>...()
56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65 4E ; V...Code...LineN
75 6D 62 65 72 54 61 62 6C 65 01 00 0A 53 6F 75 ; umberTable...Sou
72 63 65 46 69 6C 65 01 00 07 43 41 2E 6A 61 76 ; rceFile...CA.jav
61 0C 00 07 00 08 07 00 15 0C 00 16 00 17 01 00 ; a.............
06 4E 65 77 20 43 41 07 00 18 0C 00 19 00 1A 01 ; .New CA .......
00 06 63 6F 6D 2F 43 41 01 00 10 6A 61 76 61 2F ; ..com/CA ..java/
6C 61 6E 67 2F 4F 62 6A 65 63 74 01 00 10 6A 61 ; lang/Object...ja
76 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 01 00 ; va/lang/System..
03 6F 75 74 01 00 15 4C 6A 61 76 61 2F 69 6F 2F ; .out...Ljava/io/
50 72 69 6E 74 53 74 72 65 61 6D 3B 01 00 13 6A ; PrintStream;...j
61 76 61 2F 69 6F 2F 50 72 69 6E 74 53 74 72 65 ; ava/io/PrintStre
61 6D 01 00 07 70 72 69 6E 74 6C 6E 01 00 15 28 ; am...println...(
4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E ; Ljava/lang/Strin
67 3B 29 56 00 21 00 05 00 06 00 00 00 00 00 01 ; g;)V.!........
00 01 00 07 00 08 00 01 00 09 00 00 00 2D 00 02 ; ............-..
00 01 00 00 00 0D 2A B7 00 01 B2 00 02 12 03 B6 ; ......*?.?...
00 04 B1 00 00 00 01 00 0A 00 00 00 0E 00 03 00 ; ..?...........
00 00 05 00 04 00 06 00 0C 00 07 00 01 00 0B 00 ; ................
00 00 02 00 0C                                  ; .....
```

從這兩個檔案的內容我們可以知道，在類別檔之中，所有對於特定類別的所有動作都被轉換成該類別的全名，我們在 Main.java 之中所寫的 import 資訊全部都不見了(也就是說，import 除了用來指引編譯器解析出正確的類別名稱之外，沒有其他功能了)，Main.class 之中原來我們只有寫 CA 的部分，編譯之後變成 com/CA。同樣的情況也出現在 CA.class 之中，我們可以發現編譯器自動把 package 名稱和類別名稱做了結合，也組成了 com\CA 於類別檔之中。

在執行時期時，仍然用到一個與編譯器相同的程序，就是類別路徑參考表的建構，而利用動態連結載入類別檔的機制如下圖所示

圖:執行時期的動態連結相關步驟解析



我們以指令(9)的例子來說明，當系統執行 Main.class 的時候，首先先建立

類別路徑參考表，因為我們沒有指令選項-classpath 或環境變數 CLASSPATH，所以類別路徑參考表之中只有一筆資料，即預設的”.”(目前的目錄)，首先，它會先尋找 Main.class，因為 Main.class 就在目前的目錄之下，所以系統合成”.\Main.class”，自然就找到該類別檔了。接著系統要檢查 Main.class 之內的資訊，由於我們是以 src\comtime 為起始點，所以 Main.class 可以不需要屬於任何類別，所以符合相對路徑資訊，因此開始備載入執行。

接著，Main.class 之中用到 com/CA，所以它會合成”.\com\CA.class”，因此系統也可以找到 CA.class，在 CA.class 之中有個資訊為 com/CA，所以符合 src\comtime\com 這個相對路徑，因此也正常地載入執行。

所以，我們可以很大膽地預測，如果您用的指令是

java –classpath .. Main

其執行結果:

```
D:\src\comtime>java -classpath .. Main
Exception in thread "main" java.lang.NoClassDefFoundError: Main

D:\src\comtime>
```

系統告訴我們它找不到 Main 這個類別，因為從類別路徑參考表的資料來看，根本沒有足夠的路徑供我們找到 Main.class。這樣各位明白了。

檢視

利用相同的邏輯，我們回頭來解釋一下，在本章前半部之中談到 package 與 import 機制那個段落的最後一個範例，我們所使用的檔案是:

A.java

```
package edu.nctu.mot ;
public class A
{
    public static void main(String[] args)
    {
        System.out.println("I'm A") ;
    }
}
```

編譯用的指令有以下幾種，我們分別解釋當時其成功與失敗之原因:

1	**javac edu.nctu.mot.A.java**
	`D:\my>javac edu.nctu.mot.A.java`
	`error: cannot read: edu.nctu.mot.A.java`
	`1 error`
	類別路徑參考表中只有一個”.”，以此為起始點，找不到名為 edu.nctu.mot.A.java 的檔案。也就是說，**在編譯時期，不會自動把命**

	令列之中所指定的原始檔之中的"."轉換成"/"。我們指定什麼檔案，編譯器一律認為是單純的檔名。

2	**javac -classpath .\my\edu\nctu\mot A.java**
	`D:\my>javac -classpath .\edu\nctu\mot A.java`
	`error: cannot read: A.java`
	`1 error`
	類別路徑參考表中只有一個".\edu\nctu\mot"，以此為起始點，其實可以找到名為 A.java 的檔案，可是問題是，**A.java 並非透過 make 機制來編譯，而是我們手動打指令編譯，所以根本不會用到類別路徑參考表**。因此編譯器根本找不到 A.java。

3	**javac -classpath d:\my\edu\nctu\mot A.java**
	`D:\my>javac -classpath d:\my\edu\nctu\mot A.java`
	`error: cannot read: A.java`
	`1 error`
	類別路徑參考表中只有一個"d:\my\edu\nctu\mot"，以此為起始點，其實可以找到名為 A.java 的檔案，可是問題是，**A.java 並非透過 make 機制來編譯，而是我們手動打指令編譯，所以根本不會用到類別路徑參考表**。因此編譯器根本找不到 A.java。

4	**javac d:\my\edu\nctu\mot\A.java** 在指定的路徑之下可以找到該檔案，可以編譯成功。

執行用的指令有以下幾種，我們分別解釋當時其成功與失敗之原因:

1	**java edu.nctu.mot.A**
	`D:\my>java edu.nctu.mot.A`
	`I'm A`
	類別路徑參考表中只有一個"."，以此為起始點，**系統會自動把"."轉換成"/"**，因此會找到".\edu\nctu\mot"底下的 A.class。

2	**java -classpath .\edu\nctu\mot A** 類別路徑參考表中只有一個".\edu\nctu\mot"，以此為起始點，系統會 A.class，可是在檢查類別檔內部資訊的時候，發現 A.class 屬於 edu.nctu.mot 這個 package，所以應該放置在".\edu\nctu\mot\edu\nctu\mot"這個目錄之下，因此出現錯誤訊息。
3	**java -classpath d:\my\edu\nctu\mot A** 類別路徑參考表中只有一個"d:\my\edu\nctu\mot"，以此為起始點，系統會 A.class，可是在檢查類別檔內部資訊的時候，發現 A.class 屬於 edu.nctu.mot 這個 package，所以應該放置在".\edu\nctu\mot\edu\nctu\mot"這個目錄之下，因此出現錯誤

	訊息。
4	**java d:\my\edu\nctu\mot\A** 類別路徑參考表中只有一個”.”。但是系統根本無法接受這樣子的命令。

結論

在本章上半部分，筆者先採用程式黑手的方式討論整個 package 與 import 機制，說明一些初學 Java 的人常見的問題。不過畢竟是程式黑手的方式，有太多的使用案例會造成許多的錯誤訊息，我們並無法完全討論到，因而讓人有騷不到癢處的感覺。因此在本章下半部分，筆者採用由理論驗證實際的方式，為大家說明整個 package 與 import 機制的來龍去脈，只要理解了原理，就能以不變應萬變。希望本章的內容可以讓您豁然開朗。