

2

Java 程式設計簡介

規則是用來打破的

- ▼ 前言
- ▼ Java 的 package 機制
- ▼ 編譯時期（Compile-time）的 Package 運作機制
- ▼ Java 的動態連結本質
- ▼ 執行時期（Run-time）的 Package 運作機制
- ▼ 檢視
- ▼ 實際上的使用情形
- ▼ 總結

前言 ▼

在進入行動裝置程式設計的世界之前，我們必須要先加強自己對 Java 2 SDK 的掌握程度，才能達到事半功倍的效果。不過，市面上已經有不少關於 Java 程式設計的好書，況且只用一章小小的篇幅也無法完整地介紹 Java 程式設計。

所以本章打算介紹一個幾乎所有書籍都忽略掉的主題-package 機制。由於 Java 在本質上為動態連結，因此 package 機制會涉及許多在 Java 程式設計時期或是執行時期的行為。如果您已經是 Java 程式設計的老手，請藉由這個章節重新複習 package 機制；如果您是剛進入 Java 程式設計領域的新手，希望本章的說明可以讓您更了解這個在 Java 之中扮演著重要角色的機制。

Java 的 package 機制 ▼

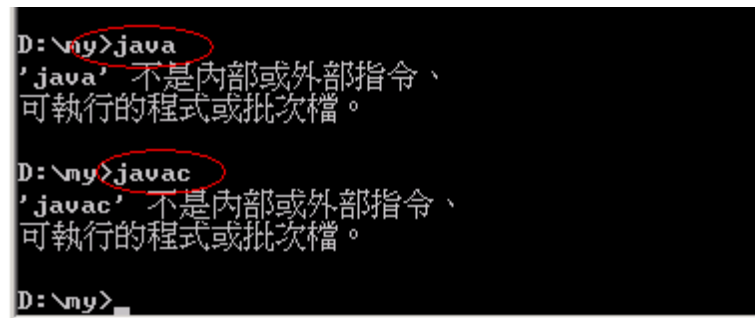
Java 的 package 機制，是每一位學習 Java 的人都必須突破的門檻，所以接下來我們將就這個議題做深入討論，底下假設您剛裝好 Java 2 SDK 於 d:\jdk1.3.0_01 目錄之中，而且沒有修改任何的環境變數（請使用指令：echo%CLASSPATH%，看看 CLASSPATH 這個環境變數是否已經設定，如果該環境變數已經被設定了，那麼您的測試結果將會和底下的測試結果有所不同）。接著我們在 D 磁碟機根目錄底下新增一個名為 my 的空目錄（d:\my），並以這個空目錄作為我們討論的起始點。



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：



```
D:\my>java
'java' 不是內部或外部指令、
可執行的程式或批次檔。

D:\my>javac
'javac' 不是內部或外部指令、
可執行的程式或批次檔。

D:\my>
```

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>path d:\jdk1.3.0_01\bin
D:\my>java
Usage: java [-options] class [args...]
        (to execute a class)
    or  java -jar [-options] jarfile [args...]
        (to execute a jar file)

where options include:
    -cp <classpath>
        set search path for application classes and resources
    -D<name>=<value>
        set a system property
    -verbose[:class!gc!jnil]
        enable verbose output
    -version
        print product version and exit
    -showversion
        print product version and continue
    -? -help
        print this help message
    -X
        print help on non-standard options

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 used
    -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>
```

接著，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。

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



```
D:\my>java A
package test

D:\my>
```

您一定覺得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，變成如下所示：

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") ;
    }
}
```

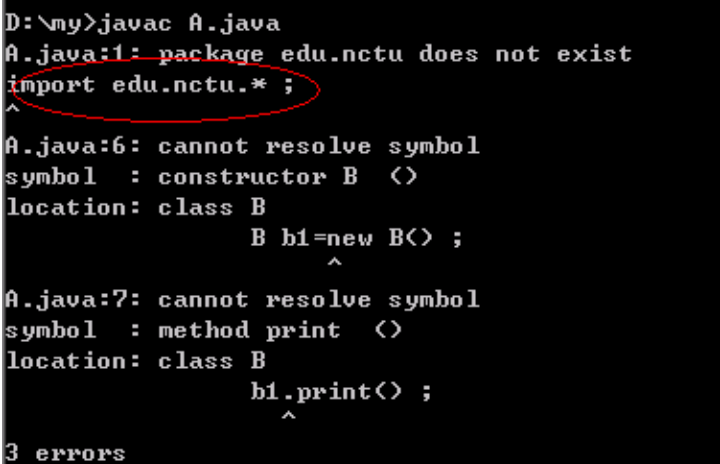
接著請您在命令列輸入：



```
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
```



```
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
```

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.* ;
```



所以程式理論上應該不會發生編譯錯誤才是。

好的，接下來請在 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 複製到 d:\my\edu\nctu 目錄下，請重新執行

javac A.java

如果操作無誤，那麼您的螢幕上會因出兩個錯誤訊息：

```
D:\my>javac A.java
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(<) ;
            ^
2 errors
```

請再將 d:\my 底下的 B.java 刪去，重新執行

javac A.java

呼~~ 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

嗯，程式順利地執行了。

接下來，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 1

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分曉。



⊕ 結論 2

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 classpath.
      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。

從這麼冗長的測試中，我們又得到了兩個結論：

⊕ 結論 3

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 4

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莫名其妙的編譯錯誤。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結論一再回來看新結論：



⊕ 結論 5

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 與 d:\my\edu\nctu 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 之後，這個預設的參數可以讓 javac.exe 找到 A.java，但是當它要尋找 d:\my\edu\nctu 的時候卻找不到，所以自然發生編譯錯誤了。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？其實這不是什麼大問題，如果做更多的測試，您將可以發現對 javac.exe 來說，不管 -classpath 選項如何指定，“目前所在的目錄（即”.”）”一定會被包含在其中，但是對 java.exe 來說就不是如此了，java.exe 會忠實地使用您所指定的路徑，不會擅加路徑到 -classpath 選項之中。

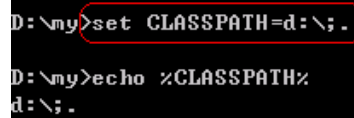
那麼您一定會問，如果每次編譯會執行都要打那麼長的指令，那豈不是非常麻煩，雖然 java.exe 也有提供一個簡化的選項 -cp



(功能同-classpath)，但是還是很麻煩。為了解決這個問題，所以提供了一個環境變數，名為 CLASSPATH，有了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:\;.



```
D:\my>set CLASSPATH=d:\;.  
D:\my>echo %CLASSPATH%  
d:\;.
```

然後您就可以像之前一樣使用指令：

javac A.java

與

java A

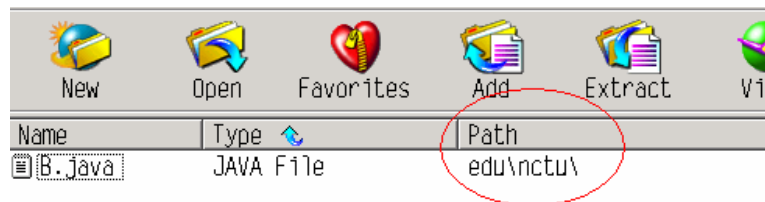
完成編譯和執行的工作。但是請注意，如果您設定了環境變數 CLASSPATH，而在使用 java.exe 或 javac.exe 的時候也使用了 -classpath 選項，那麼環境變數將視為無效，而改為以 -classpath 選項所指定的路徑為準。

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。



好的，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：



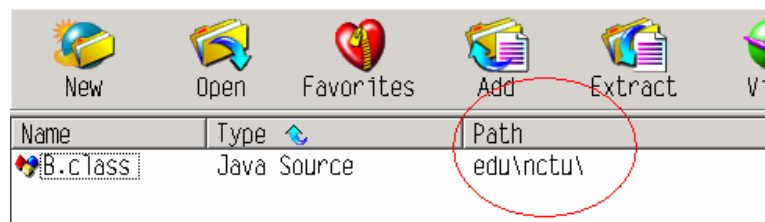
請執行

```
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
```

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 6

ZIP 檔的效力和單純的目錄是一樣的，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` 到該壓所檔中尋找類別檔。

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，如果將這些檔案全部封裝在一個壓縮檔之中，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。

在此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。

注意

前面我們曾說過，對 `javac.exe` 來說，不管 `-classpath` 選項如何指定，“目前所在的目錄（即 `.`）”一定會被包含在其中。從上述測試可以得知，這個預設的路徑一定是附加在我們額外指定的路



徑或 JAR 檔之後，即：

```
javac -classpath d:\ A.java
```

等同於

```
javac -classpath d\;. A.java
```

為什麼要特別提醒大家這個問題呢？這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（這也是我本身的切身之痛），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改版的程式，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许多錯誤。

```
D:\my>java A
Exception in thread "main" java.lang.NoClassDefFoundError: A (wrong name: edu/nctu/mot/A)
    at java.lang.ClassLoader.defineClass0(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:486)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:111)
    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	<pre>javac edu.nctu.mot.A.java D:\my>javac edu.nctu.mot.A.java error: cannot read: edu.nctu.mot.A.java 1 error</pre>
2	<pre>javac -classpath .\my\edu\nctu\mot A.java D:\my>javac -classpath .\edu\nctu\mot A.java error: cannot read: A.java 1 error</pre>
3	<pre>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</pre>
4	<pre>javac d:\my\edu\nctu\mot\A.java</pre>



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。

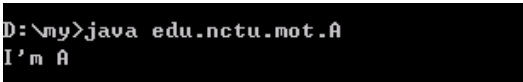
現在我們試著執行，請輸入：

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 
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`。

做了那麼多測試，歸納了這幾點結論，最後請大家在利用 `package` 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。

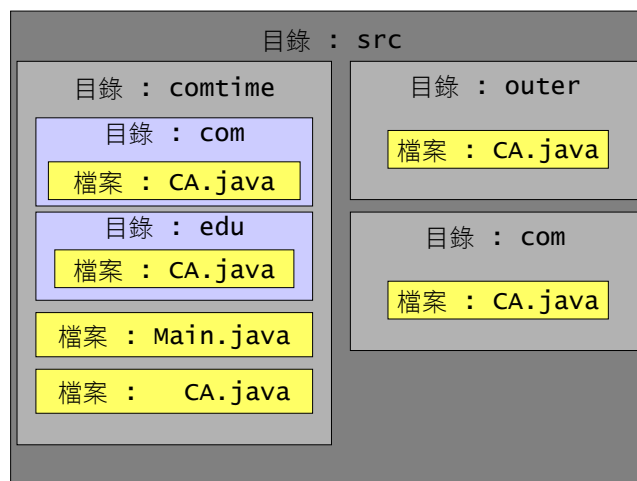


上述所得的結論只是工程上的測試結果，那麼原理呢？只要理解了這些錯誤之所以出現的理由，就可以知道成功與錯誤的原因。因此，接下來我們將向大家說明的就是 package/make 機制運作的秘密。

以下討論都假設各位遵守「每一個類別都會根據其所屬的 package 放在正確的相對路徑上」。因為從上述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，所以我們不再討論。

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：

名稱 ▲	大小	類型
com		檔案資料夾
edu		檔案資料夾
CA.class	1 KB	Java Class
CA.java	1 KB	JAVA 檔案
Main.class	1 KB	Java Class
Main.java	1 KB	JAVA 檔案

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：

名稱 ▲	大小	類型
com		檔案資料夾
edu		檔案資料夾
CA.class	1 KB	Java Class
CA.java	1 KB	JAVA 檔案
Main.class	1 KB	Java Class
Main.java	1 KB	JAVA 檔案

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所採用，只要它不屬於任何 package。

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
n.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
n.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

```

紅色框框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。

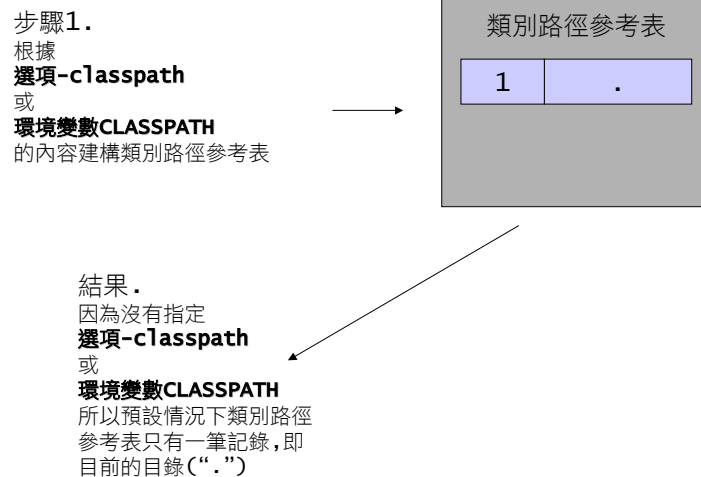
如果我們把指令改成：

javac -classpath ../.. Main.java ==>指令(8)

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



OK，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：



使用指令(3)的時候，建構「類別路徑參考表」的程序

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：



使用指令(7)的結果

類別路徑參考表	
1	..
2	.

使用指令(8)的結果

類別路徑參考表	
1	.
2	..

使用指令(7)與使用指令(8)的時候，「類別路徑參考表」的內容

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：

步驟2.

找出原始碼之中所有使用
`import packagename.*;`
 或
`import packagename.classname`
 的指令，並將 `packagename` 之中的
 “.” 以 “/” 取代。

類別路徑參考表	
1	D:\test
2	C:\p.jar

步驟3.

根據類別路徑參考表的內容為起始點，
 比對其中一個路徑下是否存在同名的
 目錄或檔案名稱

完成建構類別路徑參考表的程序

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 ;
```

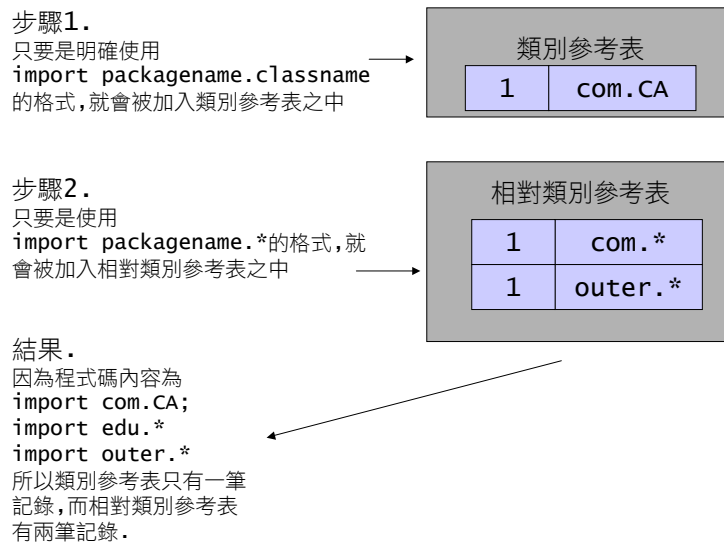
那麼編譯器就會先看看

d:\test 目錄底下是否存在有 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。

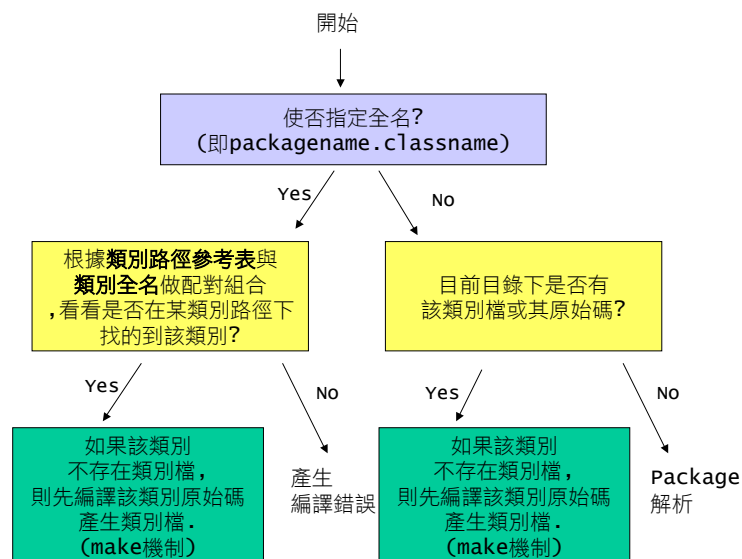
接下來，編譯器會完成一張名為「類別參考表」與「相對類別參考表」的資料結構，如下圖所示



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：



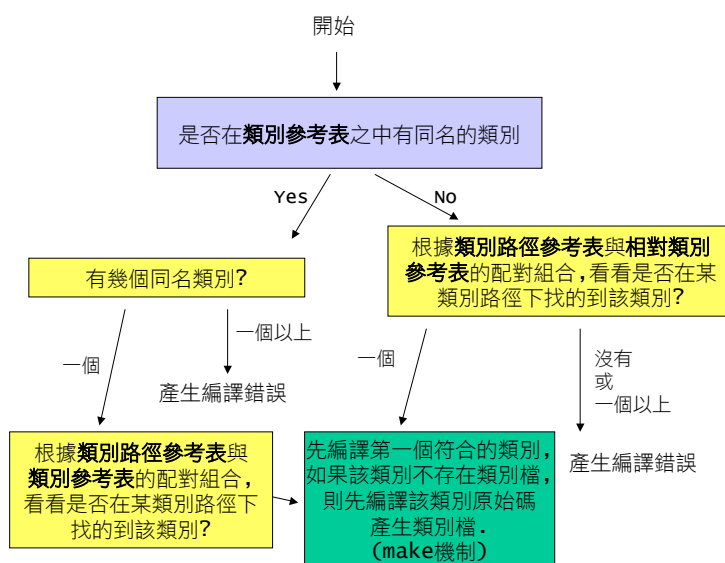


類別名稱解析的程序

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：



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();
```

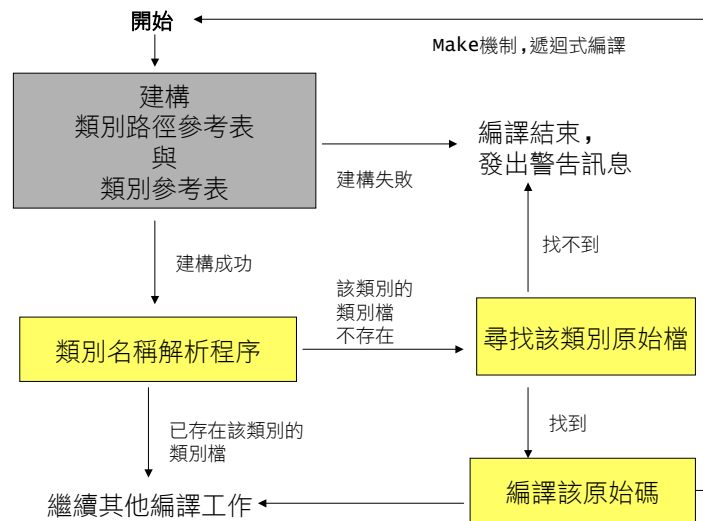
通過編譯，如果你只有

```
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，除非類別路徑參考表和相對類別參考表很大很大，否則對編譯速度來說並沒有很大的影響。



好的，到這裡為止，我們已經將整個編譯機制前半部關於 package/make 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/make 機制相關的部分用一張完整的圖做表示：



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，都會加以驗證他們是否真的屬於這個 package，如果不屬於，編譯器一樣會發出錯誤訊息；但是如果並非透過 make 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，否則會導致許多莫名其妙的問題的原因。



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/make 機制的相對應動作，所以可以編譯成功。也因為如此，請大家盡量讓整個應用程式所用到的類別都可以透過 package/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，即使是內部類別、匿名類別都一樣，舉例來說：

```
MultiClass.java

class A
{
    class C
    {
    }
}

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

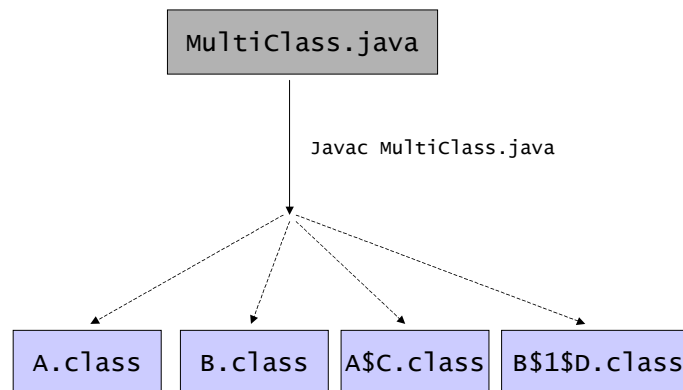
當你編譯 MultiClass.java 之後，你會發現目錄下出現了底下幾個類別檔，他們分別是：

A.class

A\$C.class

B.class

B\$1\$D.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\comtime\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，

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

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結果，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涵嗎？

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

```

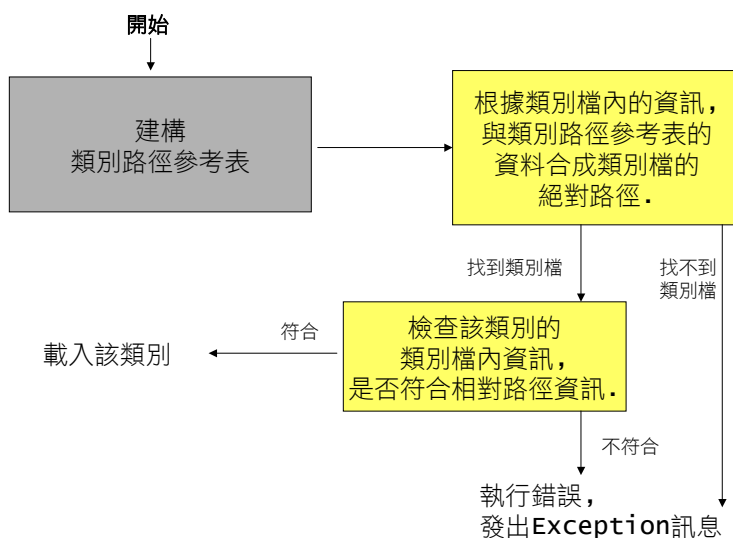
CA FE BA BE 00 03 00 2D 00 1B 0A 00 06 00 0D 09 ; █ ?...-.....
00 0E 00 0F 08 00 10 0A 00 11 00 12 07 00 13 07 ; .....<init>...()
00 14 01 00 06 3C 69 6E 69 74 3E 01 00 03 28 29 ; .....V...Code...LineN
56 01 00 04 43 6F 64 65 01 00 0F 4C 69 6E 65 4E ; umberTable...Sou
75 6D 62 65 72 54 61 62 6C 65 01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





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

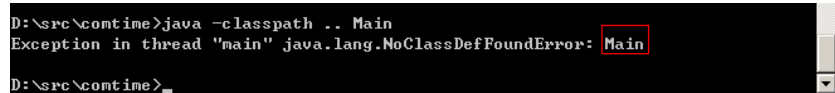


相對路徑，因此也正常地載入執行。

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：

A screenshot of a Windows command prompt window. The prompt is 'D:\src\comtime>'. The user has entered 'java -classpath .. Main'. The output is 'Exception in thread "main" java.lang.NoClassDefFoundError: Main', where 'Main' is highlighted with a red box. The prompt is now 'D:\src\comtime>'.

```
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。這樣各位明白了嗎。

檢視 ▼

利用相同的邏輯，我們回頭來解釋一下，第一個段落中最後談到的範例，我們所使用的檔案是：

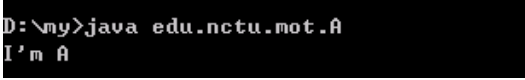
```
A.java
package edu.nctu.mot ;
public class A
{
    public static void main(String[] args)
    {
        System.out.println("I'm A") ;
    }
}
```

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	<pre>javac edu.nctu.mot.A.java</pre> <pre>D:\my>javac edu.nctu.mot.A.java error: cannot read: edu.nctu.mot.A.java 1 error</pre> 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	<pre>javac -classpath .\my\edu\nctu\mot A.java</pre> <pre>D:\my>javac -classpath .\edu\nctu\mot A.java error: cannot read: A.java 1 error</pre> 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，也就是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	<pre>javac -classpath d:\my\edu\nctu\mot A.java</pre> <pre>D:\my>javac -classpath d:\my\edu\nctu\mot A.java error: cannot read: A.java 1 error</pre> 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，也就是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	<pre>javac d:\my\edu\nctu\mot\A.java</pre> 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	<pre>java edu.nctu.mot.A</pre>  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	<pre>java -classpath .\edu\nctu\mot A</pre> 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	<pre>java -classpath d:\my\edu\nctu\mot A</pre> 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	<pre>java d:\my\edu\nctu\mot\A</pre> 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。

注意

接下來的部分，最好在您讀完第三章之後再掉頭回來看會比較適當，所以您可以暫時先略過不看。如果您已經熟悉 MIDP 程式設計，那麼請繼續看下去。



實際上的使用情形 ▼

到此為止我們所舉的例子都是使用 `java.exe` 和 `javac.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 機制息息相關。比方說如果您開發手機上的 Java 程式，必定會用到 `preverify.exe`、您要在模擬程式在手機上的執行結果時，您會用到 `midp.exe`，使用這些工具時如果您沒有遵循 package 機制的運作方式，您將無法得到您預期的結果。因此底下我們將就開發手機應用程式時會用到的工具 - `Preverify.exe` 與 `midp.exe` 做討論。

假設我們撰寫的 MIDlet 用到了額外的輔助類別，我的程式碼將看起來如下所示：

HelloMIDlet.java

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import edu.nctu.* ;
public class HelloMIDlet extends MIDlet implements
CommandListener
{
    private Command exitCommand;
    private Display display;
    public HelloMIDlet()
    {
        HelpClass hc = new HelpClass() ;
        display = Display.getDisplay(this);
        exitCommand = new Command("Exit", Command.SCREEN, 2);
    }
    public void startApp()
    {
        TextBox t = new TextBox("Hello MIDlet", "Test string",
256, 0);
        t.addCommand(exitCommand);
    }
}
```



```

        t.setCommandListener(this);
        display.setCurrent(t);
    }
    public void pauseApp()
    {
    }
    public void destroyApp(boolean unconditional)
    {
    }
    public void commandAction(Command c, Displayable s)
    {
        if (c == exitCommand)
        {
            destroyApp(false);
            notifyDestroyed();
        }
    }
}

HelpClass.java
package edu.nctu ;
public class HelpClass
{
}

```

如果您將兩個檔案放在同一個目錄下，輸入：

```

javac -g:none -classpath d:\midp-fcs\classes;. -
bootclasspath d:\midp-fcs\classes HelloMIDlet.java

```

螢幕上將出現編譯錯誤的訊息：

```

D:\jdk1.3.0_01\my>midjavac HelloMIDlet.java

D:\jdk1.3.0_01\my>javac -g:none -classpath d:\midp-fcs\classes;. -bootclasspath
d:\midp-fcs\classes HelloMIDlet.java
HelloMIDlet.java:3: package edu.nctu does not exist
import edu.nctu.* ;
^
HelloMIDlet.java:13: cannot resolve symbol
symbol  : constructor HelpClass <>
location: class HelpClass
    HelpClass hc = new HelpClass<> ;
                        ^
2 errors

```



根據我們前面的討論，於是我們將 `HelpClass.java` 放到相對目錄 `edu\nctu` 之下，然後重新執行

```
javac -g:none -classpath d:\midp-fcs\classes;. -  
bootclasspath d:\midp-fcs\classes HelloMIDlet.java
```

就可以編譯成功，您會在 `edu\nctu` 中發現 `HelpClass.class` 也一併幫您編譯了。

接下來，我們要進行預先審核，我們使用指令

```
d:\midp-fcs\bin\preverify -classpath d:\midp-fcs\classes;. -  
-d HelloMIDlet HelloMIDlet
```

預先審核的過程並不會發生任何錯誤。

於是我們試著執行指令：

```
d:\midp-fcs\bin\midp -classpath d:\midp-fcs\classes;.\ -  
HelloMIDlet -descriptor HelloMIDlet
```

執行此應用程式時您會發現螢幕上出現了錯誤訊息：

```
D:\jdk1.3.0_01\my>d:\midp-fcs\bin\preverify -classpath d:\midp-fcs\classes;. -d  
HelloMIDlet HelpClass  
Error loading class HelpClass
```

仔細到 `HelloMIDlet` 目錄中一看，目錄下只有 `HelloMIDlet.class`，`preverify.exe` 並沒有像 `javac.exe` 一樣自動幫您產生預先審核過的 `HelpClass.class`。

於是您會試著輸入指令：



```
d:\midp-fcs\bin\preverify -classpath d:\midp-fcs\classes;.
-d HelloMIDlet HelpClass
```

螢幕上會出現：

```
D:\jdk1.3.0_01\my>d:\midp-fcs\bin\preverify -classpath d:\midp-fcs\classes;. -d
HelloMIDlet HelpClass
Error loading class HelpClass
```

根據前面的討論，我們必須使用指令：

```
d:\midp-fcs\bin\preverify -classpath d:\midp-fcs\classes;.
-d HelloMIDlet edu.nctu.HelpClass
```

您會在 HelloMIDlet\edu\nctu 目錄下找到預先審核過的 HelpClass.class。

如此一來，再次使用指令：

```
d:\midp-fcs\bin\midp -classpath d:\midp-fcs\classes;.
\HelloMIDlet HelloMIDlet
```

就可以成功執行此程式。

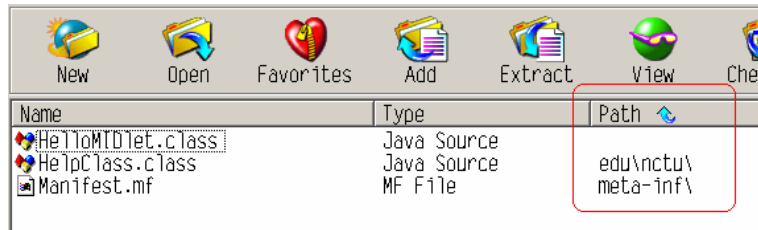
測試成功之後，我們要將它包裝成 JAR 檔，所以我們執行指令：

```
jar -cfm HelloMIDlet.jar HelloMIDlet.jad -C.\HelloMIDlet.
```

就可以產生 HelloMIDlet.jar。

HelloMIDlet.jar 的內容如下：





最後，使用指令：

```
d:\midp-fcs\bin\midp -classpath d:\midp-fcs\classes;.\
HelloMID let.jar -descriptor HelloMIDlet.jad
```

一樣可以喚起模擬器來執行我們的應用程式。

如果我們將 HelloMIDlet. java 改成下面的樣子：

```

HelloMIDlet.java

package edu ;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import edu.nctu.* ;
public class HelloMIDlet extends MIDlet implements
CommandListener
{
    .. . . . 略 .. . . .
}

```

也就是將 HelloMIDlet 類別歸屬到 edu 這個 package 之下，那麼您應該使用指令：

```
javac -g:none -classpath d:\midp-fcs\classes;. -
bootclasspathd:\midp-fcs\classes .\edu\HelloMIDlet.java
```

才能編譯成功。



接著您應該分別執行：

```
d:\midp-fcs\bin\preverify -classpath d:\midp-fcs\classes;.
-d HelloMIDlet edu.HelloMIDlet
```

與

```
d:\midp-fcs\bin\preverify -classpath d:\midp-fcs\classes;.
-d HelloMIDlet edu.nctu.HelpClass
```

然後使用指令：

```
d:\midp-fcs\bin\midp -classpath d:\midp-
fcs\classes;.\HelloMIDlet edu.HelloMIDlet
```

就可以直接測試我們的程式。

測試成功之後，我們要將它包裝成 JAR 檔，所以我們執行指令：

```
jar -cfm HelloMIDlet.jar HelloMIDlet.jad -
c .\HelloMIDlet .
```

就可以產生 HelloMIDlet.jar。

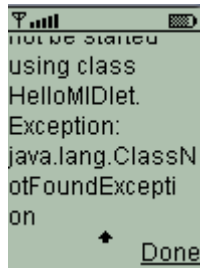
HelloMIDlet.jar 的內容如下：

 New  Open  Favorites  Add  Extract  View  CheckOut		
Name	Type	Path
 HelloMIDlet.class	Java Source	edu\
 HelpClass.class	Java Source	edu\nctu\
 Manifest.mf	MF File	meta-inf\

最後，使用指令：

```
d:\midp-fcs\bin\midp -classpath d:\midp-
fcs\classes;.\HelloMIDlet.jar -descriptor HelloMIDlet.jad
```

一樣可以喚起模擬器來執行我們的應用程式，可是這次執行時您會發現模擬器出現錯誤訊息：



命令列上也會出現錯誤訊息：

```
D:\jdk1.3.0_01\my>d:\midp-fcs\bin\midp -classpath d:\midp-fcs\classes;.\HelloMID
let.jar -descriptor HelloMIDlet.jad
java.lang.NullPointerException
Unable to create MIDlet HelloMIDlet
```

要解決這個問題，請修改 HelloMIDlet.jad，讓內容如下：

HelloMIDlet.jad

```
MIDlet-Name: HelloWorld
MIDlet-Version: 1.0.0
MIDlet-Vendor: Sun Microsystems, Inc.
MIDlet-Description: Sample Hello world MIDlet
MIDlet-Info-URL: http://java.sun.com/j2me/
MIDlet-Jar-URL: http://127.0.0.1/HelloMIDlet.jar
MIDlet-Jar-Size: 1020
MicroEdition-Profile: MIDP-1.0
MicroEdition-Configuration: CLDC-1.0
MIDlet-1: HelloWorld,,edu.HelloMIDlet
```

我們的應用程式就可以順利地執行了。



總 結 ▼

在本章中，我們為大家說明了進入 Java 程式設計領域必然碰到的進入門檻，希望透過本章，可以提供更多初學者一個沒有障礙的空間，也讓 Java 的老手能夠重新審視 Java 的 package 觀念，唯有先練好紮實的內功，才能在學習神功的路上事半功倍。

如果您對 Java 程式語言不熟悉，那麼只看完本章是不夠的，在此推薦大家兩本原文書，這兩本一直是筆者的最愛：

Core Java 2, volume 1: Fundamentals, 5/e



http://www.phptr.com/ptrbooks/ptr_0130894680.html



Core Java 2 , Volume 2: Advanced Features, 4/e



http://www.phptr.com/ptrbooks/ptr_0130819344.html

