

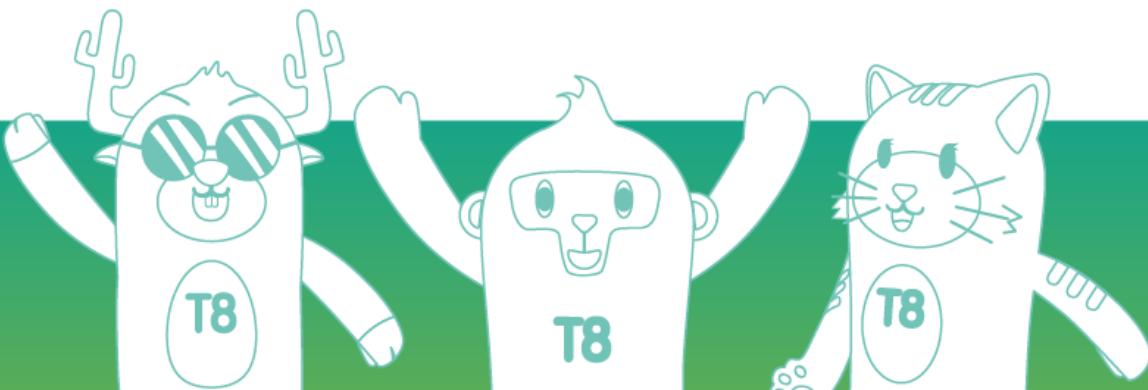
Java物件導向程式設計(基礎)

授課講師

吳冠宏

教材編寫

吳冠宏



緯
育 *TibaMe*

即學 · 即戰 · 即就業

<https://www.tibame.com/>

課程大綱

- ◆ 模組1：Java技術簡介
- ◆ 模組2：Java環境介紹
- ◆ 模組3：Java開發環境建立
- ◆ 模組4：Java基本資料型別
- ◆ 模組5：變數與常數
- ◆ 模組6：運算子功能
- ◆ 模組7：流程控制 – 選擇結構
- ◆ 模組8：流程控制 – 重複結構

課程大綱

- ◆ 模組9：方法設計與應用
- ◆ 模組10：認識物件
- ◆ 模組11：物件導向概論
- ◆ 模組12：陣列 (1)
- ◆ 模組13：陣列 (2)
- ◆ 模組14：Java字串
- ◆ 模組15：varargs機制
- ◆ 模組16：使用封裝 (1)

課程大綱

- ◆ 模組17：使用封裝 (2)
- ◆ 模組18：建構子
- ◆ 模組19：static修飾字
- ◆ 模組20：繼承
- ◆ 模組21：使用繼承
- ◆ 模組22：多型
- ◆ 模組23：多型操作
- ◆ 模組24：抽象機制與目的

課程大綱

- ◆ 模組25：介面
- ◆ 模組26：介面程式設計與多型
- ◆ 模組27：package套件介紹
- ◆ 模組28：import與類別路徑
- ◆ 模組29：Object類別介紹
- ◆ 模組30：包裝類別介紹
- ◆ 附件：Eclipse操作說明



授課講師介紹

授課講師

吳冠宏

簡歷

Oracle Certified Java Trainer

OCPJP 8

OCPWCD 6

聯絡方式

vladylo98@gmail.com

專長

Java SE,

Java EE,

Android APP

Design Pattern

學習本課程須知

先備知識

- A. 擁有基本電腦操作能力
- B. 認識英文26個字母

學習目標

- A. 清楚瞭解Java語言的功能特性、環境建立與設定
- B. 熟悉與掌握程式語言共同特性，如資料類型，運算與流程控制等
- C. 對類別設計與物件運用能有所理解與操作

學習本課程須知

學習方式

- A. 上課前預習
- B. 下課後複習
- C. 課堂練習
- D. 閱讀與理解範例程式碼
- E. 課後作業練習

須完成哪些作業或考試

- A. Java小考兩次
- B. 課後相關模組對應作業題目繳交

模組1

Java技術簡介

- 1-1 :
Java技術源起
- 1-2 :
Java語言關鍵概念
- 1-3 :
Java技術產品種類

Java發展背景

- Java Programming Language起源於1991年，由Sun Microsystems (昇陽)公司所開發
 - Java一開始被稱為Oak(橡樹)，而當時開發用途是想創造一種「能在不同CPU的消費性電子產品(如電視、電話)中，能共同使用的程式語言」
 - 專案名稱：Green Project
 - 專案主持人：James Gosling (Java之父)
- 計劃失敗後，Sun公司看見Oak在網際網路(WWW)上應用的前景，於是改造了Oak，於1995年5月以Java的名稱正式釋出。Java伴隨著網際網路的迅速發展而發展，逐漸成為重要的網路程式語言。
- 1995年5月23日，Oak正式更名為Java，同時JDK (Java Development Kits) 1.0a2版正式對外發表



Java白皮書與其專業術語

- Java白皮書 (Java “White Paper”)
 - Java發明者寫了一個有影響力的白皮書，說明他們設計目標與成就
 - May 1996, by James Gosling and Henry McGilton
 - 網址：<http://www.oracle.com/technetwork/java/langenv-140151.html>
- James Gosling用以下11個專業術語，對Java語言做摘要描述

Simple	Portable
Object-Oriented	Interpreted
Network-Savvy	High Performance
Robust	Multithreading
Secure	Dynamic
Architecture Neutral	

Java語言關鍵概念 - Simple (簡單)

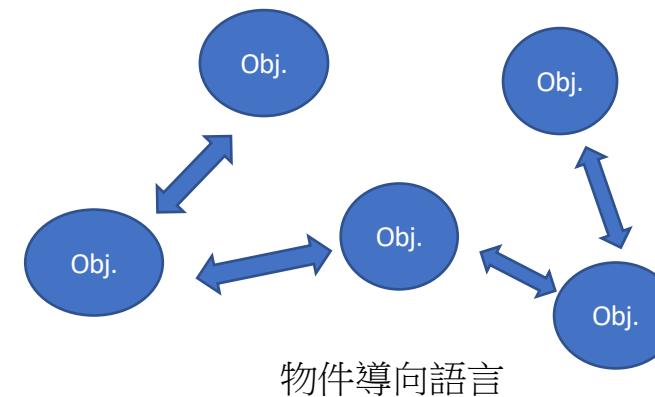
- Java語言之所以簡單，是因為發明者移除了一些程式語言較為複雜的地方與含糊不清的程式結構
- Java不允許程式設計者直接使用指標(pointer)去操控記憶體位址，這不但複雜而且容易發生錯誤，這是C與C++使用的常見問題
- Java只允許程式設計者使用物件參考(Object reference)來操控物件，和資源回收機制(Garbage collection)來自動處理已不被參考的物件
- 另一個簡單的原因，是因為Java的布林(boolean)資料型態可以有true或false的值，不像其它語言只有1與0的值

Java語言關鍵概念 - Object-Oriented (物件導向)

- 程序性程式語言 (Procedural Programming)
 - 所著重的是利用撰寫程式的先後次序來解決問題
- 物件導向程式語言 (Object-Oriented Programming, OOP)
 - 著重於物件之間可以相互作用的關係來解決問題



程序式語言



物件導向語言

Java語言關鍵概念 - Distributed (分散式運算)

- Java是一種分散式程式語言，支援許多分散式網路技術
 - RMI (Remote Method Invocation)
 - CORBA (Common Object Request Broker Architecture)
 - URL (Universal Resource Locator)
- Java的動態類別載入功能允許部份的程式碼可由網路下載，並在個人電腦執行，如Applet



Java語言關鍵概念 - Platform-independent (跨平台)

- Java程式設計者先使用Java編譯器 (Java Compiler)，將Java程式碼譯成與平台無關的位元碼 (byte code)
- 位元碼透過各系統專有的Java虛擬機器 (JVM, Java Virtual Machine)上的硬體去執行
- JVM (Java Virtual Machine)目前已有多種平台版本，如Linux, Mac, Windows, Solaris...等，除了這些大型作業系統外，還有針對各種小型系統設計的JVM，如PDA, 手機等...
- Java可以在不同的CPU或不同作業系統環境上執行，**JVM**即是Java能跨平台的主要原因

Java技術產品區分

- Java Standard Edition (Java SE)
 - 標準版
 - 適用於開發用戶端程式
- Java Enterprise Edition (Java EE)
 - 企業版
 - 適用於開發伺服器端程式
- Java Micro Edition (Java ME)
 - 手持裝置版
 - 適用於開發手機、無線設備等程式
- 自Java 6以後，已取消了J2SE, J2EE等用法

Java版本演進

- Java版本更新周期於2017年時通過從原本兩年一版變更為每六個月釋出一版更新
- J2SE 5.0可被視為是一次Java的重大版本更新，許多常用的語言機制與Libraries於此時加入
- Java SE 8 (LTS意思為長期支援)，是目前企業使用率最高，也是Java認證考試的主要版本

版本	發布日期	最終免費公開更新時間 ^{[3][4]}	最後延伸支援日期
JDK Beta	1995	?	?
JDK 1.0	1996 年 1 月	?	?
JDK 1.1	1997 年 2 月	?	?
J2SE 1.2	1998 年 12 月	?	?
J2SE 1.3	2000 年 5 月	?	?
J2SE 1.4	2002 年 2 月	2008 年 10 月	2013 年 2 月
J2SE 5.0	2004 年 9 月	2009 年 11 月	2015 年 4 月
Java SE 6	2006 年 12 月	2013 年 4 月	2018 年 12 月
Java SE 7	2011 年 7 月	2015 年 4 月	2022 年 7 月
Java SE 8 (LTS)	2014 年 3 月	Oracle 於 2019 年 1 月停止更新（商用） Oracle 於 2020 年 12 月停止更新（非商用） AdoptOpenJDK 於 2026 年 5 月或之前停止更新 Amazon Corretto 於 2023 年 6 月或之前停止更新	2030 年 12 月
Java SE 9	2017 年 9 月	OpenJDK 於 2018 年 3 月停止更新	不適用
Java SE 10	2018 年 3 月	OpenJDK 於 2018 年 9 月停止更新	不適用
Java SE 11 (LTS)	2018 年 9 月	Amazon Corretto 於 2024 年 8 月或之前停止更新 AdoptOpenJDK 於 2022 年 9 月停止更新	2026 年 9 月
Java SE 12	2019 年 3 月	OpenJDK 於 2019 年 9 月停止更新	不適用
Java SE 13	2019 年 9 月	OpenJDK 於 2020 年 3 月停止更新	不適用
Java SE 14	2020 年 3 月	OpenJDK 於 2020 年 9 月停止更新	不適用
Java SE 15	2020 年 9 月	OpenJDK 於 2021 年 3 月停止更新	不適用
Java SE 16	2021 年 3 月	OpenJDK 於 2021 年 9 月停止更新	不適用
Java SE 17 (LTS)	2021 年 9 月	待定	待定

格式： ■ 舊版本 ■ 舊版本, 仍被支援 ■ 目前版本 ■ 未來版本

資料來源：維基百科

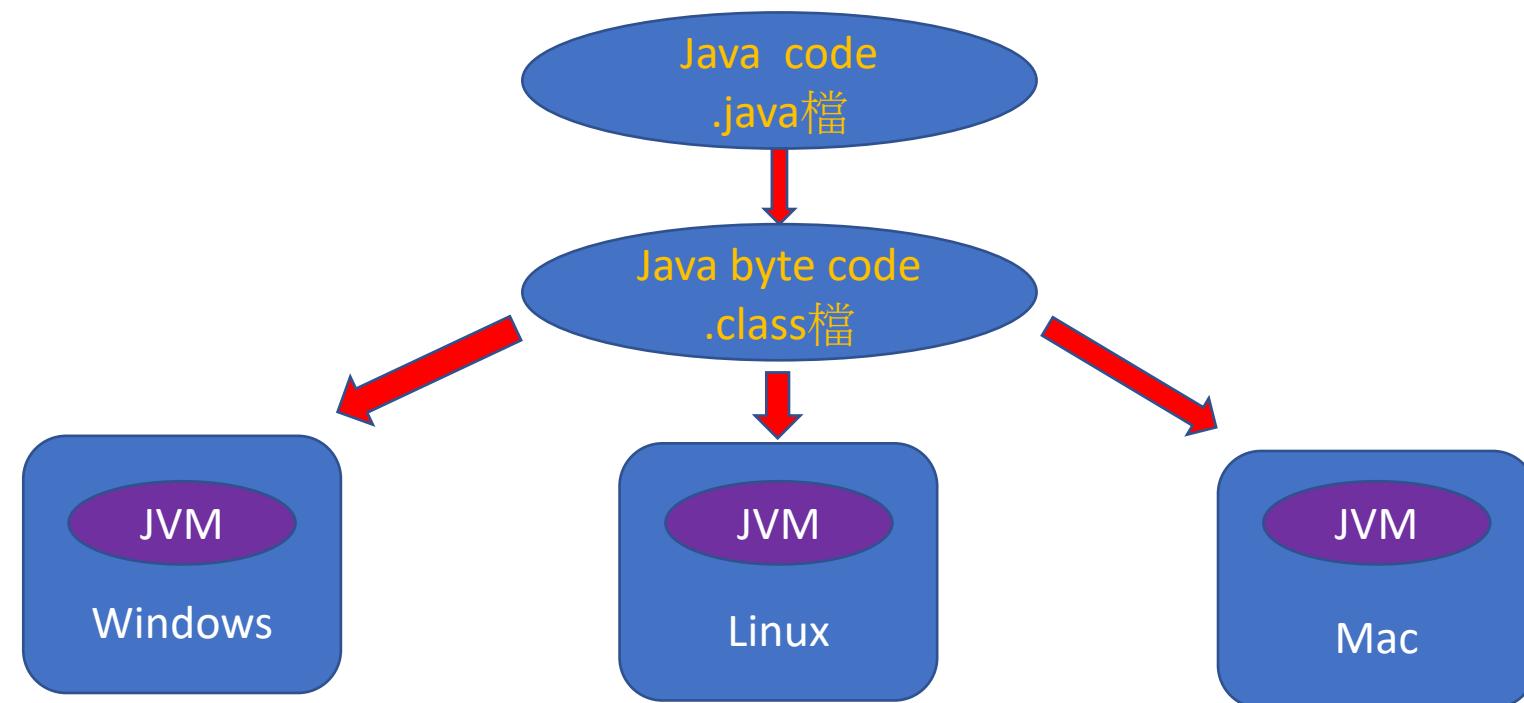
模組2 Java環境介紹

- 2-1：
java檔與class檔機制
- 2-2：
Java執行環境(JRE)
- 2-3：
Java開發環境(JDK)

java檔與class檔關係

- java檔是什麼：
 - 程式設計師藉由文字編輯器將程式碼撰寫完成後所儲存的檔案格式
 - 此為Java程式語言的標準規範
 - java檔裡面儲存的內容為原始碼，又稱為Source code
- class檔是什麼：
 - 藉由編譯器(Java Compiler)將指定的java檔編譯成功後的檔案格式
 - 此為Java程式語言的標準規範
 - class檔裡面儲存的內容為位元碼，又稱為Java byte code
 - 與任何作業系統平台無關

java檔與class檔關係 (圖解)



Java Runtime Environment (JRE)

- Java 程式的執行環境
 - 一個Java程式只需要一個Java虛擬機器(JVM)去執行
 - 一個Java程式也需要一套針對此平台所設計的標準**Java類別函式庫**(Java class libraries)
- JVM與Java類別函式庫的組合被稱為：
 - Java 執行環境 (Java Runtime Environment)

Write once, run anywhere

- Java虛擬機器(Java Virtual Machine)與類別函式庫為Java平台的技術規範
 - 如Oracle的HotSpot、IBM的J9等
- Sun Microsystems (昇陽)在開發Java同時，對許多常見的系統平台都提供了Java執行環境，使程式設計師在撰寫Java程式時，無需考慮到系統與硬體平台的問題，專心致力於程式功能與邏輯實現
- Write once, run anywhere已成為Java程式設計師們的精神指標！

Java Development Kit

- 從一般使用者晉升成為開發者的Java程式設計師，光是JRE是無法滿足開發時的相關需求，例如像是原始碼藉由Java編譯器編譯成為Java byte code，**JRE本身是不包含開發相關工具的**
- **Java Development Kits**，簡稱**JDK**，為當時昇陽公司針對Java程式開發人員所發布的免費軟體開發套件，並在2006年基於GPL授權而開源，最後也促成OpenJDK的發布
- 註：一般的軟體開發套件都通稱為**SDK (Software Development Kit)**，如Android SDK，Facebook SDK, Google SDK等...但為了特別突顯Java語言，所以用了J取代了**Software**的稱呼

Java Development Kit

- 一般使用者其實不需要建立開發環境，只要可以執行應用程式即可，也就是安裝JRE執行環境就好，但**JDK裡包含了Java程式開發所需的編譯、除錯等功能**；甚至為了確認程式設計上是否無誤，所以**JDK其實也內含了JRE執行環境**
- 包含在JDK裡的主要開發相關工具(可在%JAVA_HOME%\bin路徑下找到)：
 - javac
 - java
 - jar
 - javadoc
 - javap
 - jdb

模組3

Java開發環境

建立

- 3-1 :
JDK安裝與環境變數
- 3-2 :
第一隻Java程式
- 3-3 :
main方法與程式架構

JDK安裝與環境變數設定 (課堂實作)

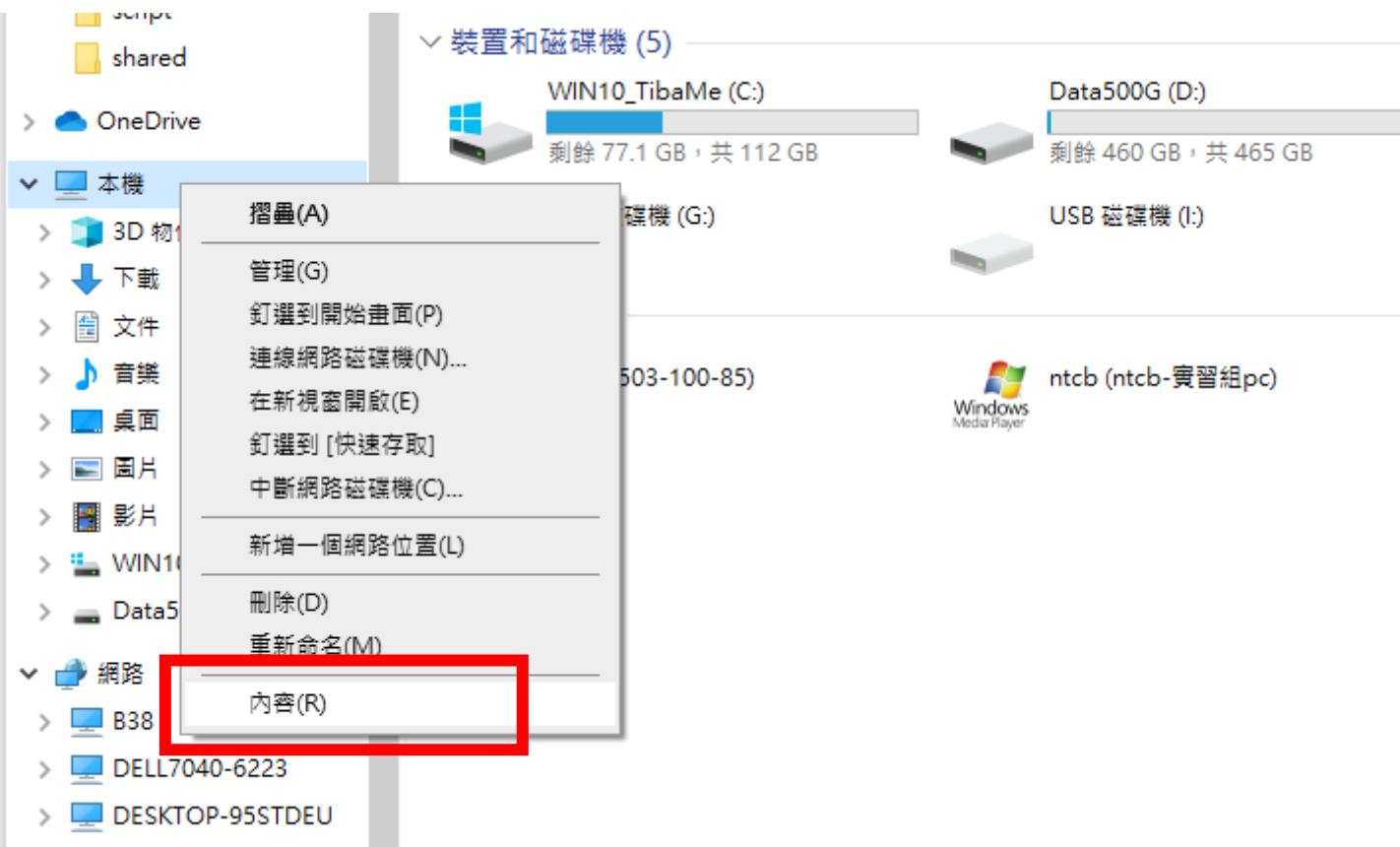
- 下載JDK
 - JDK可自行到相關網站下載，如Oracle JDK或是AdoptOpenJDK
 - 選擇對應的系統平台下載對應的JDK
- 安裝JDK
 - 請預先在C:/ (以WINDOWS系統)下建立資料夾，如jdk-17.0.1
(XX為版本號碼，注意資料夾名稱勿有空白)
 - 執行JDK安裝程式，並將檔案安裝在剛建立的資料夾裡
- 環境變數設定 (我的電腦→右鍵→內容→進階→環境變數)，並於下方的系統變數→新增或編輯以下三行：

- JAVA_HOME	如： C:\jdk-17.0.1
- Path	如： %JAVA_HOME%\bin
- classpath	如： .;

原因課程中說明

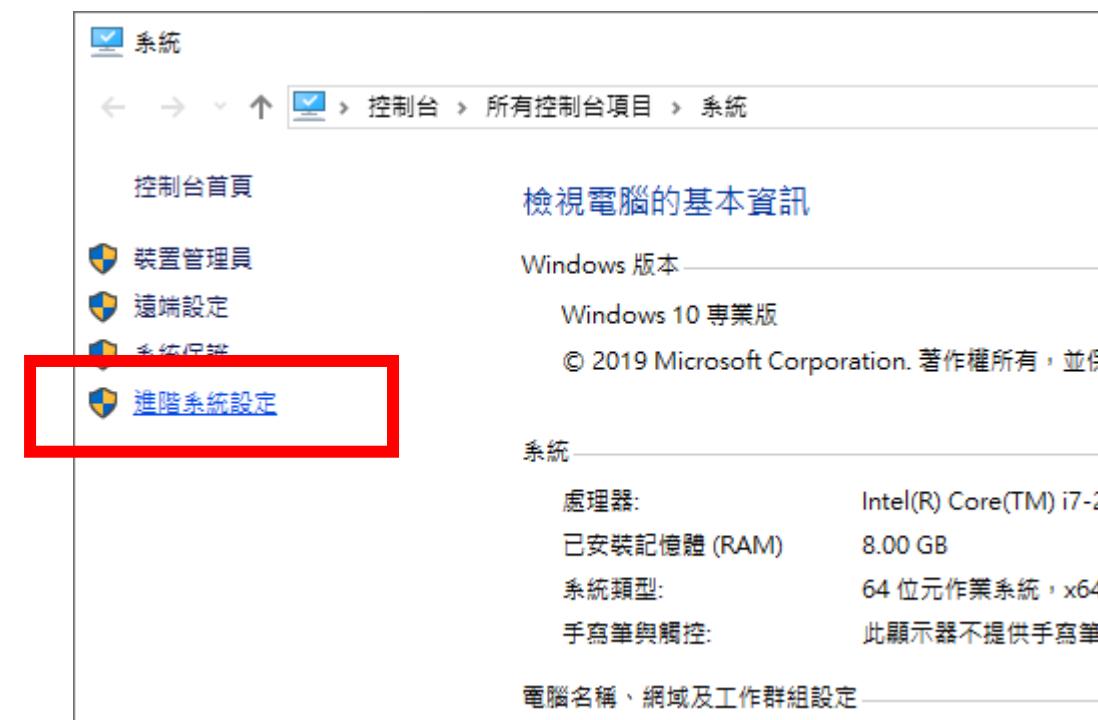
JDK安裝與環境變數設定 (圖解)

- 對著本機右鍵點選”內容”



JDK安裝與環境變數設定 (圖解)

- 點選”進階系統設定”



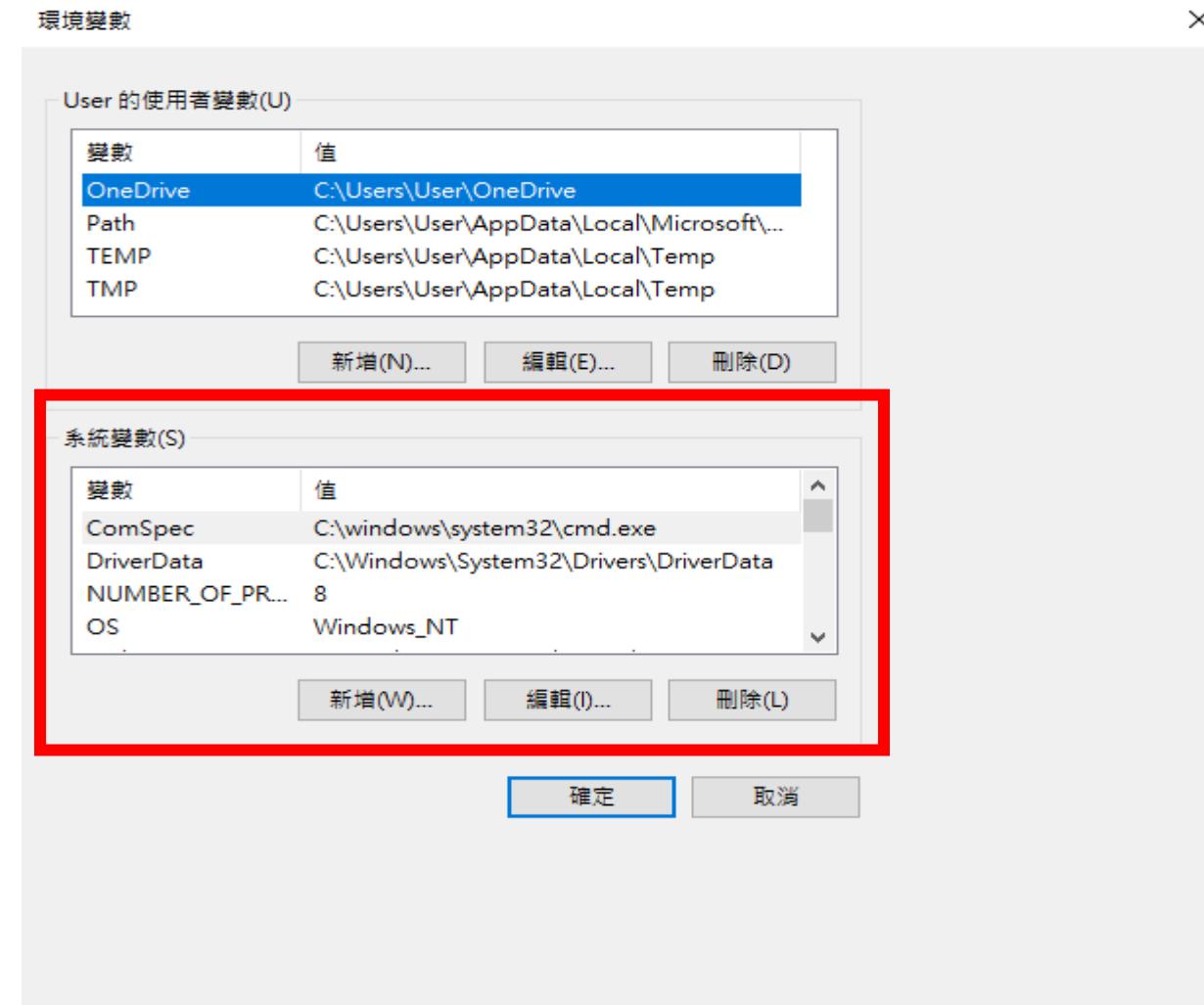
JDK安裝與環境變數設定 (圖解)

- 點選”環境變數”



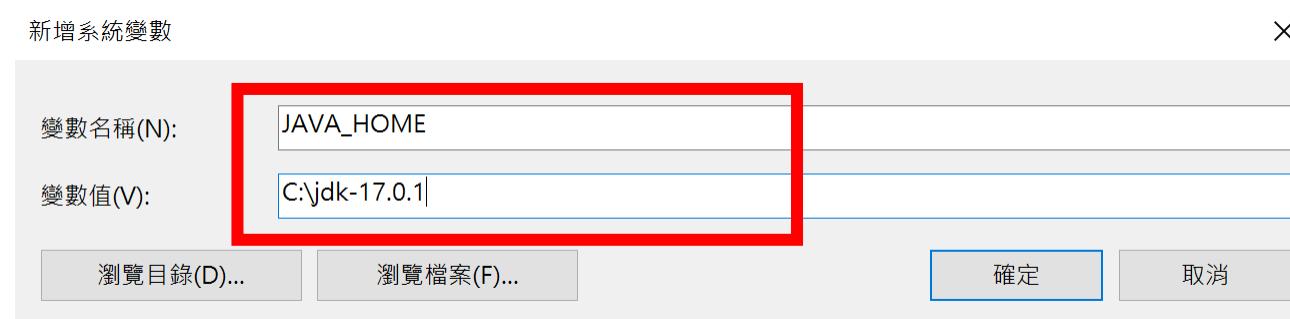
JDK安裝與環境變數設定 (圖解)

- 對”系統變數”做設定



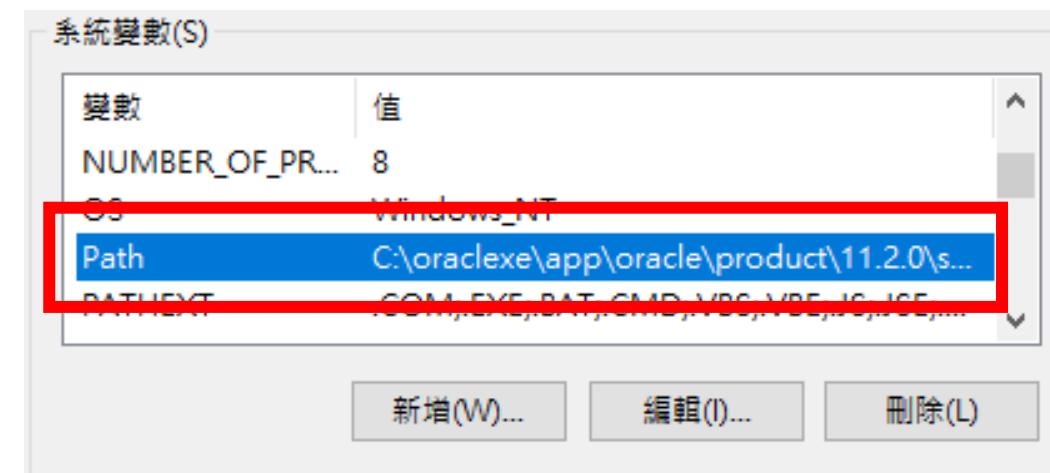
JDK安裝與環境變數設定 (圖解)

- 1. 點選”新增”
- 2. 變數名稱請輸入 : JAVA_HOME
- 3. 變數值請輸入JDK安裝的資料夾路徑，如C:\jdk-17.0.1



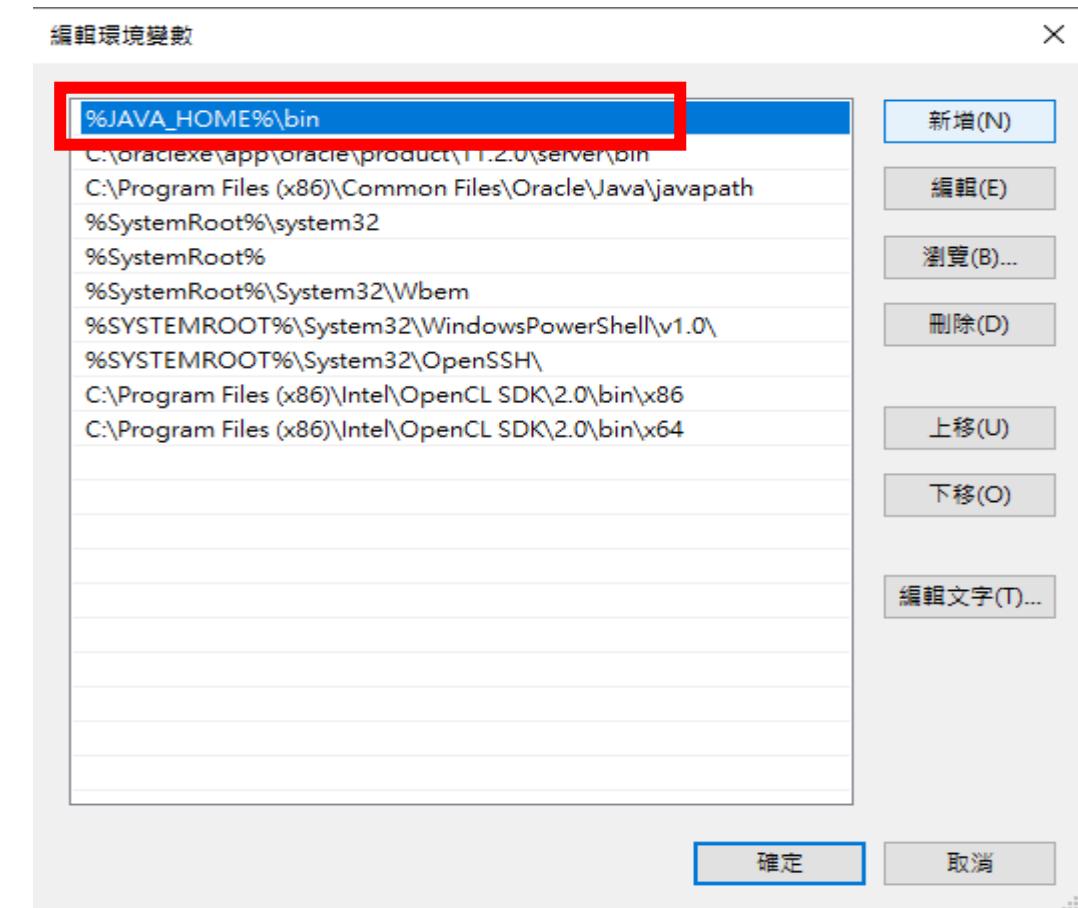
JDK安裝與環境變數設定 (圖解)

- 找到Path後，點選"編輯"



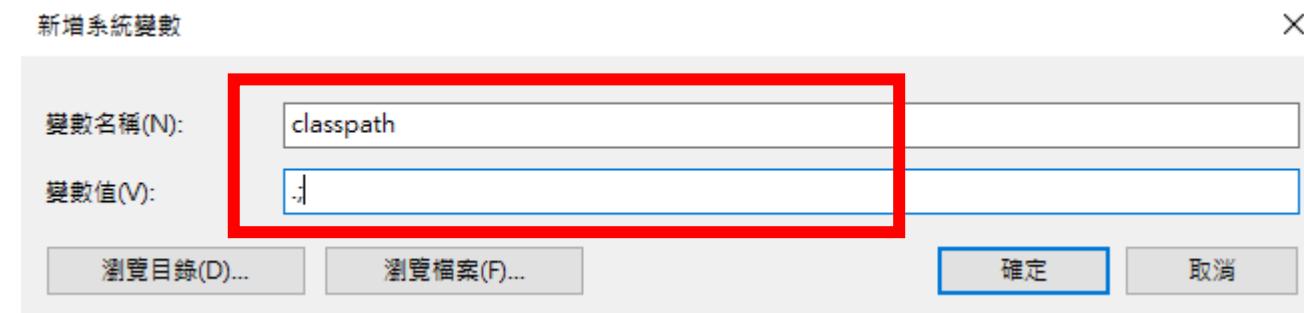
JDK安裝與環境變數設定 (圖解)

- 1. 點選”新增”，並輸入：%JAVA_HOME%\bin
- 2. 把這個路徑透過”上移”，移到第一個位置



JDK安裝與環境變數設定 (圖解)

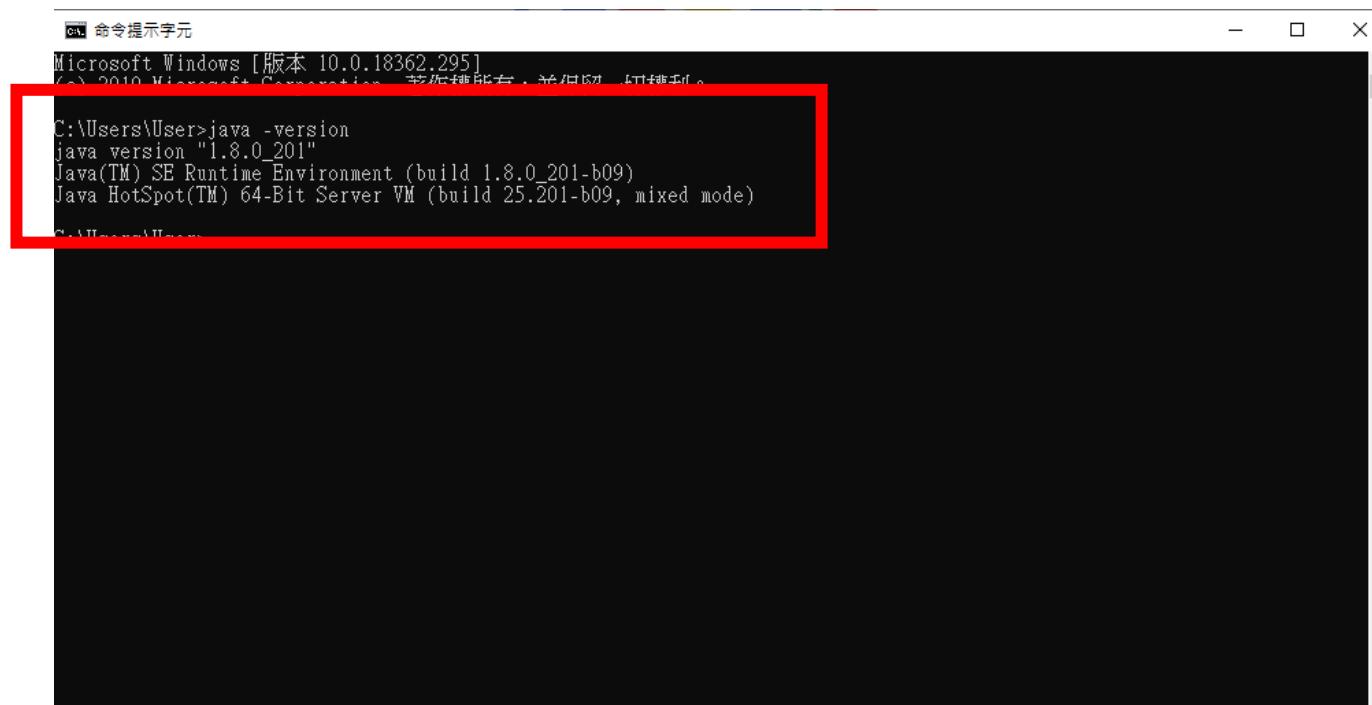
- 1. 點選”新增”
- 2. 變數名稱 : classpath
- 2. 變數值 : ..; (於介紹類別路徑時說明)



JDK安裝與環境變數設定 (圖解)

- 在cmd工具裡可以此指令測試java版本資訊

指令 : java -version

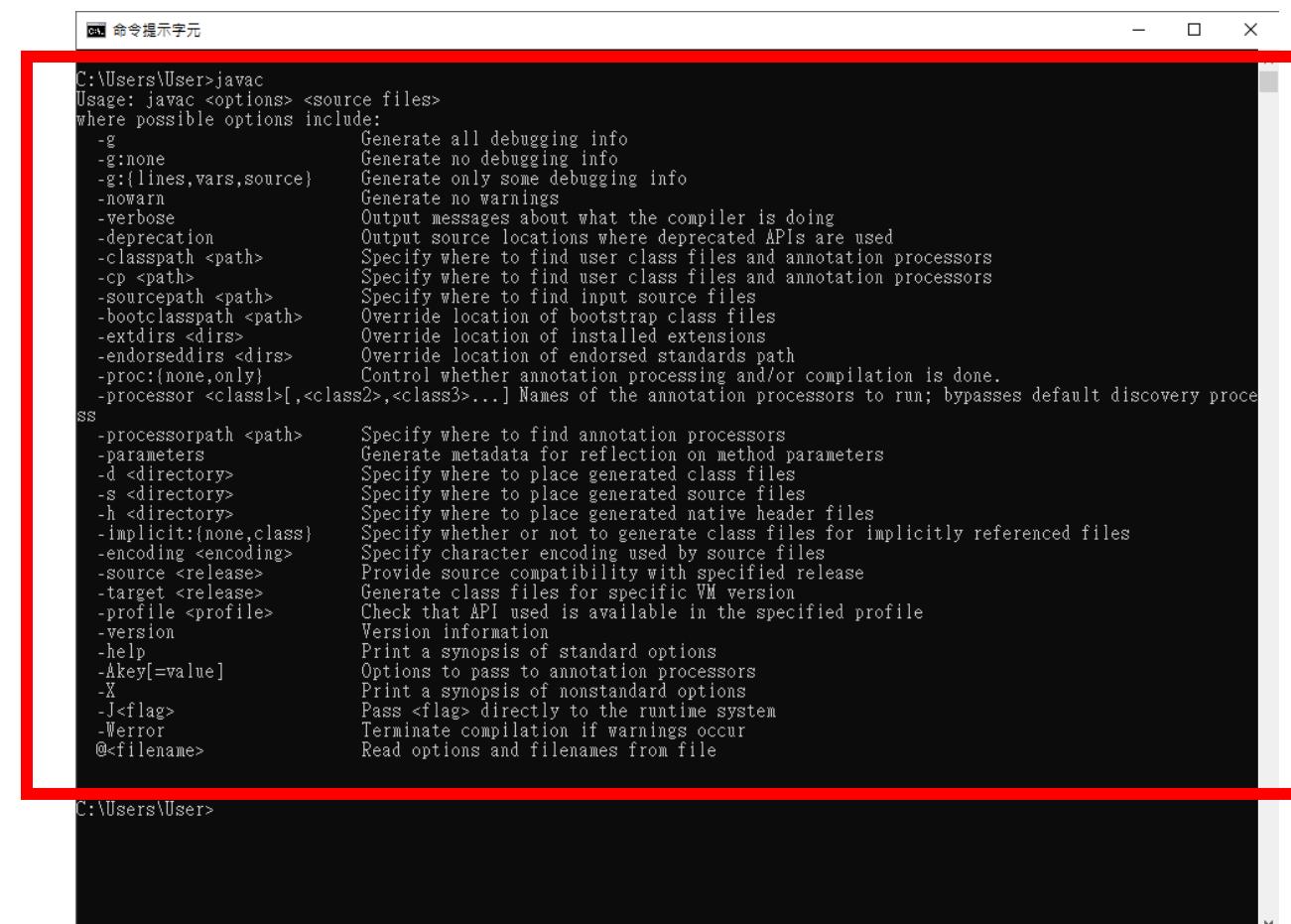


```
命令提示字元
Microsoft Windows [版本 10.0.18362.295]
(c) 2019 Microsoft Corporation. 皆有所有權。保留所有權利。
C:\Users\User>java -version
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
```

JDK安裝與環境變數設定 (圖解)

- 接著下此指令測試環境變數是否設定成功

指令 : javac



```
C:\Users\User>javac
Usage: javac <options> <source files>
where possible options include:
  -g                                     Generate all debugging info
  -g:none                                Generate no debugging info
  -g:{lines,vars,source}                  Generate only some debugging info
  -nowarn                                Generate no warnings
  -verbose                               Output messages about what the compiler is doing
  -deprecation                           Output source locations where deprecated APIs are used
  -classpath <path>                      Specify where to find user class files and annotation processors
  -cp <path>                             Specify where to find user class files and annotation processors
  -sourcepath <path>                      Specify where to find input source files
  -bootclasspath <path>                   Override location of bootstrap class files
  -extdirs <dirs>                         Override location of installed extensions
  -endorseddirs <dirs>                   Override location of endorsed standards path
  -proc:{none,only}                       Control whether annotation processing and/or compilation is done.
  -processor <class1>[,<class2>,<class3>...] Names of the annotation processors to run; bypasses default discovery proce
ss
  -processorpath <path>                   Specify where to find annotation processors
  -parameters                            Generate metadata for reflection on method parameters
  -d <directory>                          Specify where to place generated class files
  -s <directory>                          Specify where to place generated source files
  -h <directory>                          Specify where to place generated native header files
  -implicit:{none,class}                 Specify whether or not to generate class files for implicitly referenced files
  -encoding <encoding>                   Specify character encoding used by source files
  -source <release>                      Provide source compatibility with specified release
  -target <release>                      Generate class files for specific VM version
  -profile <profile>                     Check that API used is available in the specified profile
  -version                               Version information
  -help                                  Print a synopsis of standard options
  -Akey[=value]                           Options to pass to annotation processors
  -X                                     Print a synopsis of nonstandard options
  -J<flag>                             Pass <flag> directly to the runtime system
  -Werror                               Terminate compilation if warnings occur
  @<filename>                           Read options and filenames from file

C:\Users\User>
```

程式開發工具

- 文字編輯輔助工具
 - Notepad++ (<http://notepad-plus-plus.org/zh/>) 免費
 - UltraEdit (<http://www.ultraedit.com/>)
- 整合開發環境 (Integrated Development Environment, IDE)
 - NetBeans (<http://www.netbeans.org/>) Oracle官方建議使用IDE
 - Eclipse (<http://www.eclipse.org/>) 免費並為本課程所使用
- 建議初學者先使用文字編輯工具，瞭解Java編譯與執行方式後，從撰寫錯誤與找bug過程中成長，再行使用IDE以增加生產力與開發效率

Hello World (課堂實作)

- 撰寫你的第一支Java程式：HelloWorld.java

```
public class HelloWorld { ←  
    public static void main(String[] args) { ←  
        System.out.println("Hello World!");  
    } ←  
} ←
```

- Java的檔案名稱必須以class的名稱來命名，所以檔名要儲存為
HelloWorld.java

Java 程式基本形式

- 基本的 Java 類別架構：

```
<modifier>* class <class name> {  
    <attribute _declaration>*  
    <constructor _declaration>*  
    <method _declaration>*  
}
```

- 一個類別(class)的成員分為變數與方法兩種成員

Java 程式基本形式(續)

- 類別(class)宣告：
 - <modifier>* class <class_name> {...}
 - 修飾存取方式 class 類別名稱 {...}
- 屬性變數宣告與初始化：
 - <modifier>* <type> <name> [= <initial_value>];
 - 修飾存取方式 資料型態 變數名稱 [= 初始值];
- 方法(method)宣告：
 - <modifier>* <return_type> <name> (<argument>*) {...}
 - 修飾存取方式 傳回值型態 方法名稱 (傳入參數) {...}
- 註解：單行→ //..... 多行→ /*.....
.....
*/

程式執行的進入點 - main方法

- HelloWorld.java範例

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

- main方法遵循先前所提的語法，比較特別的地方為：

- main方法包含**兩個修飾子public與static**，兩者都一定要有
- main方法並不回傳值，因此**傳回值型態為void**
- main方法其本身方法名稱為**main**
- main方法可以接受0 ~ 多個字串作為參數從(**String[] args**)傳入，可以讓你在命令列執行時，將參數傳入程式碼中 (詳細於陣列array內容說明)

模組4

Java基本資料 型別

- 4-1：
基本資料字面常數
- 4-2：
八大基本型別介紹
- 4-3：
字元與Unicode編碼

什麼是字面常數

- 多數人把字面常數稱之為Literal
- 代表在程式碼裡固定資料的寫法格式
- Java字面常數可分為以下幾種：
 - 整數字面常數
 - 浮點數字面常數
 - 布林字面常數
 - 字元字面常數

整數字面常數

- 根據整數長度大小，Java提供了四種整數資料類型：
 - **byte, short, int, long**
- Java整數字面常數可以用十進位、八進位與十六進位制表示(Java 7以後可以用二進位制表示)
- 十進位即為我們平常使用表示，如100，12345
- 八進位需為0開頭，如0123，0567
- 十六進位需為0x或0X開頭，如0x8E，0X8E
- 二進位需為0b或0B開頭，如0b1001，0B1001
- 為了增加閱讀性，Java 7開始，可以使用底線(_)來分隔整數定數
 - 如會計人員會使用1,000,000表示一百萬，在Java裡我們可以用1_000_000作為表示

浮點數字面常數

- 根據浮點數長度大小，Java提供了兩種浮點數資料類型：
 - **float, double**
- 1.23, 3.14等為我們常見的浮點數字面常數，若是在後面加上f或F，則可以讓此浮點數成為一個**float**類型資料，如1.23f或1.23F
- Java浮點數字面常數另有科學記號表示法，如1.23e-10(1.23×10^{-10})，較常見於工程或是電腦領域使用，e大小寫均可，代表10的次方數

布林字面常數

- Java提供了一種布林值定數資料類型：
 - **boolean**
- Java使用**true**與**false**來代表布林值定數
- 換作日常生活就是我們常說的「真」與「假」、「對」與「錯」、「Yes」與「No」
- 對電腦來說，1就是**true**，0就是**false**

字元字面常數

- Java提供了一種字元值定數資料類型：
 - **char**
- 字元值定數為一個Unicode字元，或是在一對單引號裡的特殊字元(又稱轉義序列(Escape Sequence))
- 一般字元表示方式如：'a', 'o', '%', '我', 'A'
- 轉義序列表示方式如：'\n' (換行), '\\' (\), '\" (")
- Java允許使用Unicode表示法，如 '我'在Unicode編號為6211，則可以使用'\u6211'表示

字元字面常數 – 轉義序列

- 轉義序列(Escape Sequence)，也有人稱為跳脫字元

➤ \'	: 單引號
➤ \"	: 雙引號
➤ \\	: 反斜線
➤ \n	: 換行
➤ \t	: tab鍵
➤ \b	: 倒退一格
➤ \f	: 換頁
➤ \r	: return鍵 (Enter鍵)
➤ \u	: Unicode碼表示

基本資料型態 (Primitive Data Types)

範例：[TestPrimitiveType.java](#)

型態名稱	大小	範圍	說明	初始值
整數型態				
byte	8 bits	$-2^7 \sim 2^7-1$ (-128 – 127)	位元組整數	0
short	16 bits	$-2^{15} \sim 2^{15}-1$	短整數	0
int	32 bits	$-2^{31} \sim 2^{31}-1$	整數	0
long	64 bits	$-2^{63} \sim 2^{63}-1$	長整數	0L
浮點數型態				
float	32 bits	$-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$	7位小數	0.0F
double	64 bits	$-1.8 \times 10^{308} \sim 1.8 \times 10^{308}$	15位小數	0.0(D)
其它型態				
boolean	1 bit	true, false	布林	false
char	16 bits / Unicode格式	$0 \sim 2^{16}-1$ (0-65535) (\u0000 - \uffff)	字元	'\u0000'

基本資料型態其初始值指定

型態名稱	初值設定
整數型態	
byte	byte i = 1;
short	short i = 1;
int	int i = 1;
long	long i = 1L;
浮點數型態	
float	float i = 1.1F;
double	double i = 1.1; double i = 1.1D;
其它型態	
boolean	boolean b = true; boolean b = false;
char	char c = 'A'; char c = '\u0041'

編碼法 (Encoding)

- 編碼法(Encoding)：將文字編碼成數字，再以0與1表示這些數字
- 電腦世界常見的編碼法：

ASCII

American Standard Code for Information Interchange
(美國國家標準資訊交換碼)
長度為1 byte
拉丁字母(歐美國家常用)
範例：'A' (= 65) · 'a' (= 97)

Big5

大五碼：大千、倚天、國喬、零壹
正體中文(台灣、香港常用)
長度為2 bytes
範例：'乙' = 1010010001000001 (0xA441)

Unicode

萬國碼(Universal Code)
世界上所有字母(全世界通用)
長度為2 bytes (UTF-8, UTF-16)或4 bytes (UTF-32)
範例：'乙' = 1000111001011001 (0x4E59)

編碼法 (Encoding)

- UTF-8編碼介紹：
 - Universal Transformation Format (通用轉換格式)
 - 原ASCII使用1 byte儲存 (u0000 ~ u07FF)
 - 拉丁文、希臘文、希伯來文與阿拉伯文使用2 bytes儲存 (u0080 ~ u07FF)
 - 中、日、韓 (簡稱CJK)漢字或字母使用3 bytes儲存 (u0800 ~ uFFFF)
 - 其它字母使用4 bytes儲存 (u10000 ~ u10FFFF)
 - **因為省空間且與ASCII相容，故為現今最常使用的編碼**
- UTF-16無法與ASCII相容，需經過轉換，所以較少用
- UTF-32為最標準的Unicode格式，一律使用4 bytes儲存，除非要顯示罕用語言與字母，否則不太使用

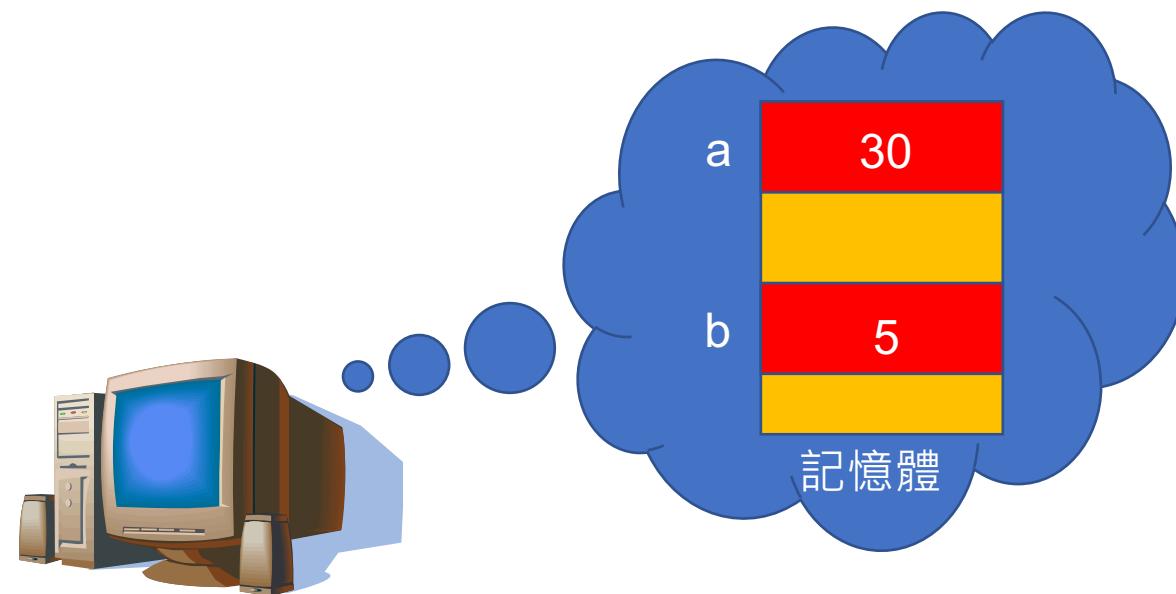
模組5

Java變數與常數

- 5-1：**
變數宣告與命名規則
- 5-2：**
變數種類與有效範圍
- 5-3：**
常數宣告與命名規則

變數機制目的

- 凡計算前必先記憶！
- 每公升油為 a 元，共加了 b 公升，我們可以計算出 $a \times b$ 元，假設一公升為30元，共加了5公升...
- 對電腦來說，如下圖：



變數命名規則

- 每個變數名字都代表著記憶體中某個記憶體位址
- 變數名稱：
 - 可以用**A-Z, a-z, 0-9, _**(底線), **\$**，長度不限制
 - 第一個字不可以是數字
 - 大小寫不同 (case-sensitive)
- 不能是關鍵字 (**keyword**) 或稱作保留字
- 變數、方法名稱以小寫開頭，類別名稱為大寫開頭
 - 如果名稱由不同字組成，建議後面字的首字元為大寫，如**myName**

Java語言關鍵字與保留字

- Java Programming Language Keywords

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

- goto與const也為關鍵字，但不可以在Java裡使用 (JDK 13開始多了yield關鍵字)
- Reserved literal words : **null**, true and false

變數宣告與初始化

- 變數被使用前皆需要有初值，否則編譯時會有錯誤訊息

➤ 基本的宣告方式：

- <資料型態> <變數名稱>;

- 如：`int i; int i, j, k;`

➤ 宣告之後，再指定初始值 (區域變數才可以)

- 如：`int i;`

- `i = 0;`

➤ 宣告同時指定初始值：

- <資料型態> <變數名稱> = <初始值>;

- 如：`int i = 0; int i = 0, j = 1, k = 2;`

變數種類整理

- 區域變數 (Local variables)：
 - 宣告在方法(method)或程式區塊(block)內
 - 區域變數只能在它們被宣告的同方法內存取
 - 又稱為automatic, temporary或stack variables
- 實體變數 (Instance variables)：
 - 宣告在方法之外，類別之內，**且沒有static修飾子**
 - 實體變數可被類別內任何非static方法存取
 - 又稱member variables(成員變數)、attribute variables(屬性變數)
- 注意：
 - 加上static修飾子為類別變數，也稱為靜態變數 (詳見修飾子的static部份)

常數宣告目的與規則

- 若程式裡所使用的資料不會有異動或修改的前提下，可宣告成為常數來保證資料的不變性
- 常數(Constant)**通常全部大寫**，且習慣使用**底線 _**來分開字組
 - 例如：`MY_NAME`
- 一個**變數**宣告為**final**，表示這個變數在初始值化後，不得再變更其值，也就是常數(Constant)
 - 例如：`final double PI = 3.14;`

模組6

Java運算子功能

- 6-1：
各種運算子功能介紹
- 6-2：
運算子先後順序
- 6-3：
資料型別晉升與轉換

運算子與運算元

- 運算子 (Operator)可對一個以上的運算元(Operand)進行運算動作
- 運算子執行運算之後將回傳值，而其回傳值型態視運算元而定

運算子名稱	
算術運算子	+, -, *, /, %
遞增遞減運算子	++, --
指定運算子	=, +=, -=, *=, /=, %=
關係運算子	<, <=, >, >=, ==, !=
條件運算子	&&, , !
位元運算子	&, , ^, ~
移位運算子	<<, >>, >>>
三元運算子	? :

算術運算子

範例：[TestArithmeticOP.java](#)

- 算術運算子 (Arithmetic Operators)又稱為二元運算子

算術運算子	用法	說明
+	$a + b$	1+1為數字相加 “1” + 1為串接相加
-	$a - b$	a 減 b
*	$a * b$	a 乘 b
/	a / b	a 除 b ($5.0/2 = 2.5$; $5/2 = 2$)
%	$a \% b$	取 a 除以 b 後的餘數 ($7\%2 = 1$; $9.6\%3.5 = 2.6$)

- 若兩個運算元的位階不相等，則運算完後的回傳值會與位階高者相同
- 若兩個運算元為基本型別，至少會轉換成int型別

遞增遞減運算子

範例：[TestInDecrementOP.java](#)

- 遞增(++)與遞減(--)運算子通常又稱為一元運算子
- 遞增與遞減運算子分為**後置型**與**前置型**

遞增遞減運算子	用法	說明
(後置型)++	a++	$a = a + 1$
(後置型)--	a--	$a = a - 1$
(前置型)++	++a	$a = a + 1$
(前置型)--	--a	$a = a - 1$

- 後置型 (a++)：先取值後再遞增；前置型 (++a)：先遞增後再取值

- 如：`int a = 3;`

`System.out.println(a++); // 3`

`System.out.println(a); // 4`

`System.out.println(++a); // 4`

`System.out.println(a); // 4`

指定運算子

範例：[TestAssignOP.java](#)

- 指定運算子 (assignment operators)是將**右邊**運算完成的結果**指定給左邊**的變數保存起來
 - “=”並非數學上”等於”的意思，而是”**指定**”的意思
 - 因此等號右邊的型別位階不可以超過等號左邊的型別位階
 - **位階高低順序**：**double > float > long > int > short > byte**

指定運算子	用法	相當於
=	a = 2	a = 2
+=	a += 2	a = a + 2
-=	a -= 2	a = a - 2
*=	a *= 2	a = a * 2
/=	a /= 2	a = a / 2
%=	a %= 2	a = a % 2

關係運算子

範例：[TestRelationalOP.java](#)

- 關係運算子 (Relational operators)用來比較兩個變數的值，其回傳值結果是一個布林值(true / false)

關係運算子	用法	回傳true/false結果
<	$a < b$	a 是否小於 b
\leq	$a \leq b$	a 是否小於等於 b
$=\!=$	$a == b$	a 是否等於 b
$\!=\!$	$a != b$	a 是否不等於 b
\geq	$a \geq b$	a 是否大於等於 b
>	$a > b$	a 是否大於 b

條件運算子

範例：[TestConditionalOP.java](#)

- 條件運算子 (Conditional operators)是將兩個布林值合併起來的運算子，運算結果仍為布林值

條件運算子	用法	回傳true/false結果
<code>&&</code>	<code>a && b</code>	<code>a</code> 和 <code>b</code> 都是 <code>true</code> 才回傳 <code>true</code>
<code> </code>	<code>a b</code>	<code>a</code> 和 <code>b</code> 只要有一方為 <code>true</code> 就回傳 <code>true</code>
<code>!</code>	<code>!a</code>	傳回相反的布林值

位元運算子(補充)

範例：[TestBitOP.java](#)

- 位元運算子 (Bitwise operators) : &(and), |(or), ^(xor)可用在整數的位元運算
- ~ (位元反轉運算子) : 用於整數，將位元1變成0，0變成1

~	0	1	0	0	1	1	1	1
<hr/>								
	1	0	1	1	0	0	0	0

0	0	1	0	1	1	0	1	
<hr/>								
&	0	1	0	0	1	1	1	1

0	0	1	0	1	1	0	1	
<hr/>								
^	0	1	0	0	1	1	1	1

0	0	1	0	1	1	0	1	
<hr/>								
	0	1	0	0	1	1	1	1

移位運算子(補充)

範例：[TestShiftOP.java](#)

- 移位運算子 (Shift operators)用在整數型態的位元移位運算

移位運算子	用法	功能
<code>>></code>	<code>a >> b</code>	將a向右移動b個位元 (真正負值)
<code><<</code>	<code>a << b</code>	將a向左移動b個位元 (真正負值)
<code>>>></code>	<code>a >>> b</code>	將a向右移動b個位元 (不真正負值向右移位，以0位元補進)

- 移位運算子的右邊參數，若超過型態本身的位元數，則以餘數作為移位的次數

三元運算子

範例：[TestTernaryOP.java](#)

- 三元運算子 (Ternary operators)是一個簡略的if – else敘述

三元運算子	用法	功能
<code>? :</code>	<code>a ? b : c</code>	如果條件a為true，執行運算式b 如果條件a為false，執行運算式c

- 例：

```
if ( x > y )
```

```
    a = x + 100;
```

```
else
```

相當於： `a = (x > y) ? x + 100 : y + 100;`

```
    a = y + 100;
```

運算子優先順序

Operators	Associative
<code>++ -- ~ !</code>	R to L
<code>* / %</code>	L to R
<code>+ -</code>	L to R
<code><< >> >>></code>	L to R
<code>< > <= >= instanceof</code>	L to R
<code>== !=</code>	L to R
<code>& ^ </code>	L to R
<code>&& </code>	L to R
<code><boolean_expr> ? <expr1> : <expr2></code>	R to L
<code>= += -= *= /= %= <<= >>= >>>= &= ^= =</code>	R to L

晉升與型別轉換

範例：[TestCast.java](#)

- 晉升 (Promotion)
 - 指較小的資料型別(等號的右邊)自動晉升為較大的資料型別(等號的左邊)
- 型別轉換 (Typecasting)
 - 指較大的資料型別轉換成為較小的資料型別
 - 必須使用強制轉換
 - 語法：(target type) value

例：int x = 1;

```
double y = 2.2;
```

```
y = x + 1; // 晉升
```

```
x = (int)y + 1; // 型別轉換
```

模組7

流程控制 –

選擇結構

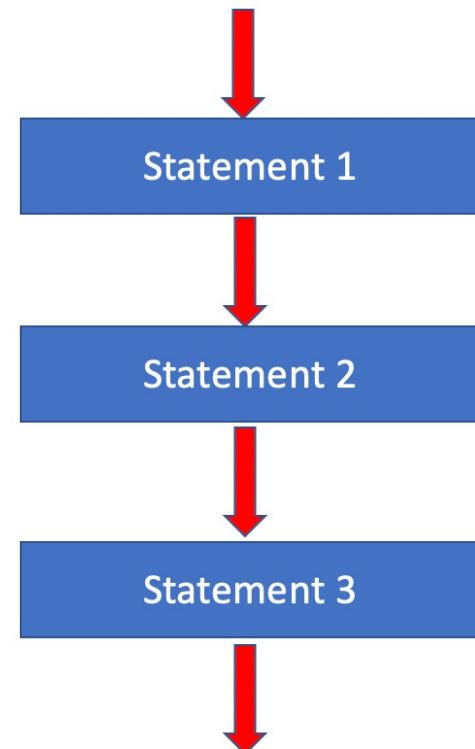
7-1 :
if ... else

7-2 :
switch ... case

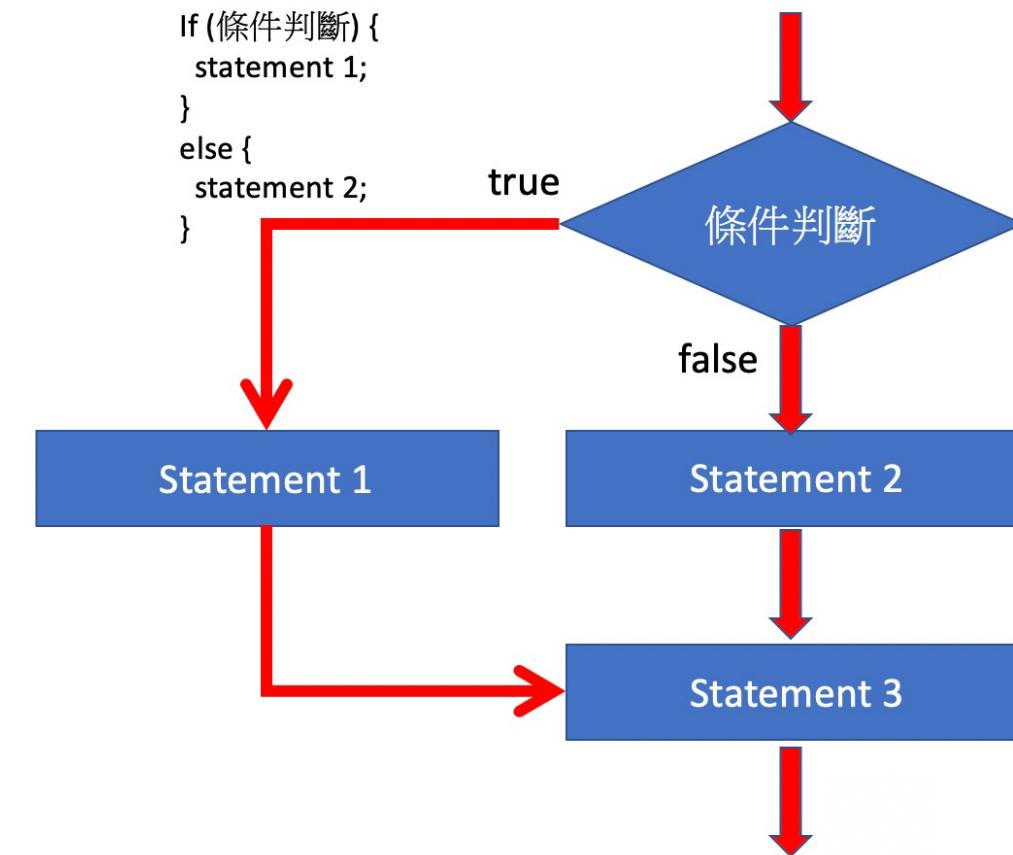
7-3 :
巢狀選擇結構

程式流程

1. 循序結構(sequence)

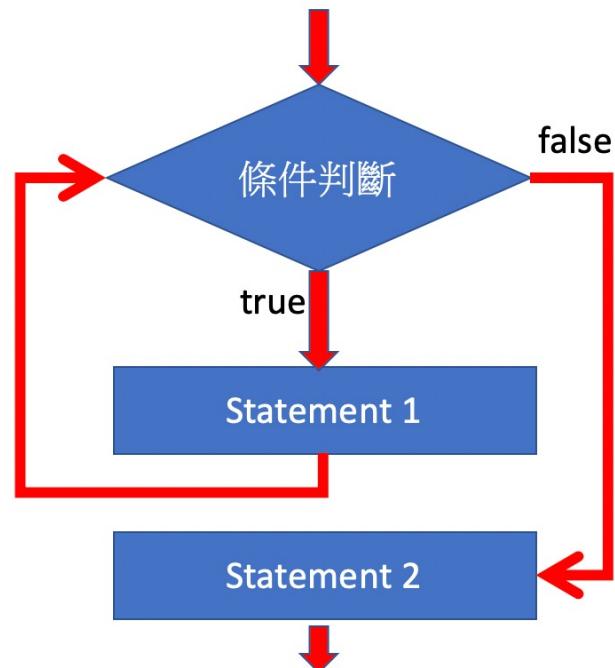


2. 選擇結構(selection)



程式流程(續)

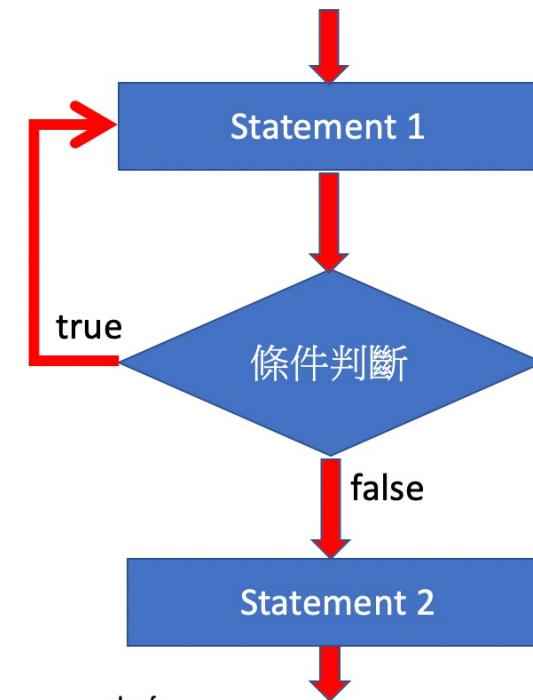
1. 重複結構(前測式) – for, while



```
for (初值設定; 條件判斷; 計次) {  
    statement 1;  
}
```

```
while (條件判斷) {  
    statement 1;  
}
```

2. 重複結構(後測式) – do-while



```
do {  
    statement 1;  
} while (條件判斷);
```

選擇性敘述

- 選擇性敘述用來決定某一個或是多個敘述要不要被執行
 - 單向選擇 : if
 - 雙向選擇擇一 : if else
 - 多重選擇擇一 : if...else...if...else
 - switch...case
- 語法 :
 - 見下一頁
 - 大括號內若只有一條敘述，則大括號可省略
- 通常會結合關係運算子 : <, <=, >, >=, ==, != 一起使用

if ... else statement

範例：[TestIfElse1.java](#)

單向選擇

```
if (條件判斷) {  
    敘述1;  
    ...  
}
```

雙向選擇擇一

```
if (條件判斷) {  
    敘述1;  
    ...  
}  
else {  
    敘述2;  
    ....  
}
```

多重選擇擇一

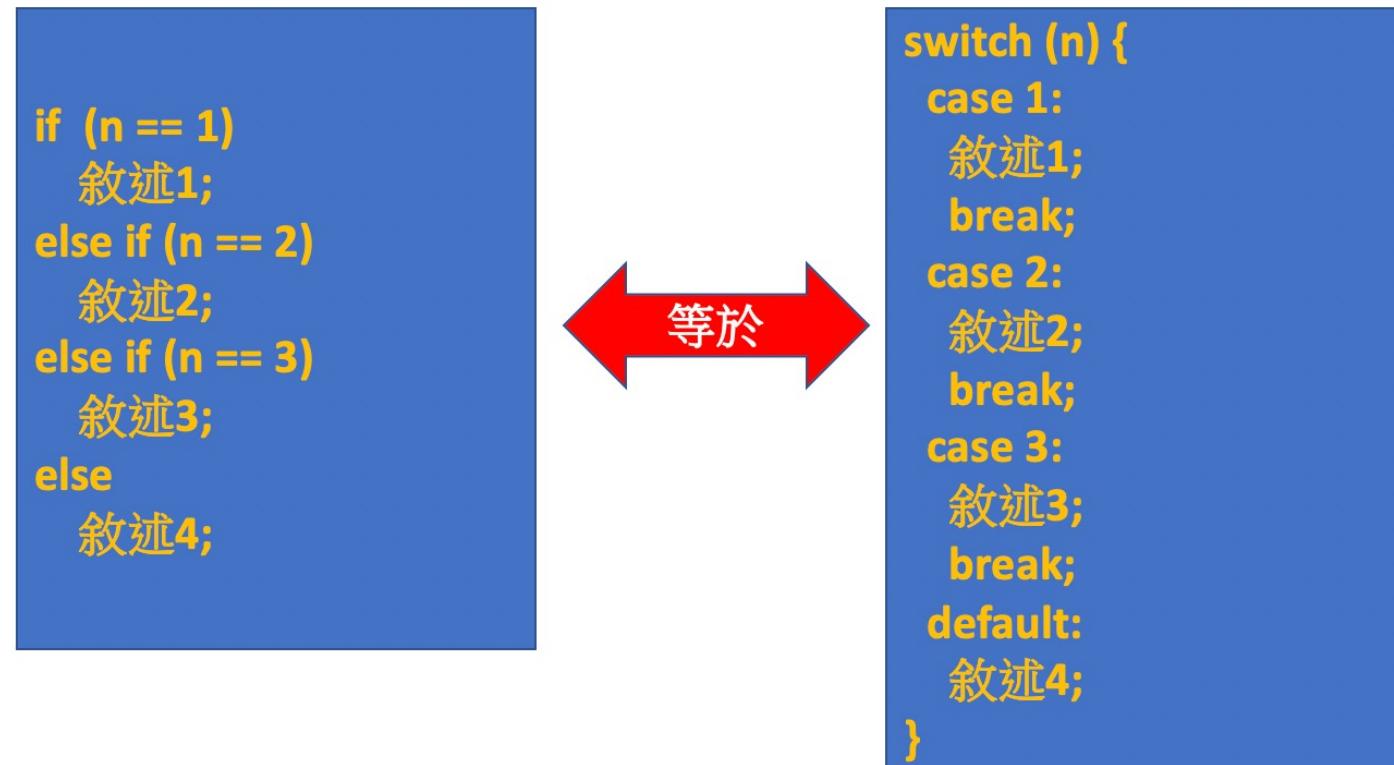
```
if (條件判斷) {  
    敘述1;  
    ...  
}  
else if (條件判斷) {  
    敘述2;  
    ....  
}  
else if (條件判斷) {  
    敘述n;  
    ...  
}  
else {  
    敘述;  
    ...  
}
```

- 請建立一個**TestBMI.java**，並計算自己的BMI值後輸出，另加入判斷結果為過瘦、正常或是過重
 - 提示一：BMI公式為體重(kg) / 身高²(m)
 - 提示二：BMI<18.5為過瘦， $18.5 \leq \text{BMI} < 24$ 為正常、 $\text{BMI} \geq 24$ 為過胖
 - 提示三：運算子 + if – else判斷

switch ... case statement

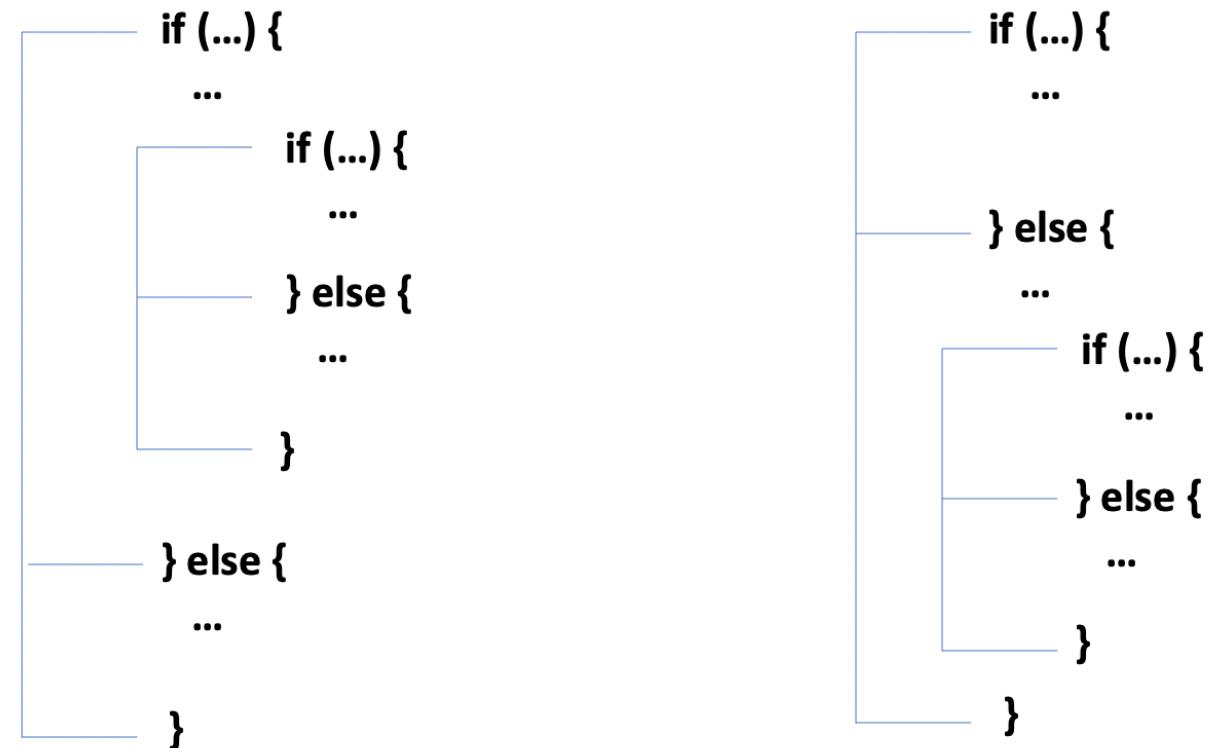
範例：[TestSwitchCase.java](#)

- switch case變數(n)只可為**整數**、**字元**，不可為浮點數 (**JDK 7以後，可以比對字串**)
- 若省略break敘述，則會執行下一個case中的敘述



Nested if – else statement

- 較複雜的情況，我們可以使用巢狀 if – else 敘述



關係/條件運算子與 if else

範例：[TestIfElse2.java](#)

- &&與||

- 當左式的條件可決定最終的結果時，便停止檢查右式的條件

```
if (gender.equals("女"))
    if (age <= 29)
        System.out.println("我請她看電影^_^");
    else
        System.out.println("謝謝再聯絡");
else
    System.out.println("謝謝再聯絡");
```

- 因此以上程式片段可改寫成下列程式片段

```
if (gender.equals("女") && age <= 29)
    System.out.println("我請她看電影^_^");
else
    System.out.println("謝謝再聯絡");
```

模組8 流程控制 – 重複結構

- 8-1：
迴圈設計要素介紹**
- 8-2：
三種迴圈比較**
- 8-3：
`break, continue`關鍵字**

loop statement

- Java有三種迴圈：**for, while, do...while**
- 迴圈有三要素：**初值設定、條件判斷、計次**。執行方式說明如下：
 - 初值設定：
 - 初值設定是為了設定控制迴圈執行的變數**起始值**，並於進入迴圈時，一開始執行的程式運算，只執行一次
 - 條件判斷：
 - 條件判斷為**是否重複執行迴圈的依據**
 - 如果條件判斷結果為**true**，則進入迴圈(繼續執行)；如果條件判斷結果為**false**，則跳離迴圈(中斷執行)
 - 計次：
 - 每經過一次迴圈，就會執行一次計次運算式，通常計次運算，多為設定控制迴圈執行變數的**遞增或遞減運算式**

三種迴圈的語法

範例：[TestWhile.java](#)
[TestDoWhile.java](#)

for迴圈：

```
for (初值設定；條件判斷；計次){  
    敘述  
}
```

while迴圈：

```
while (條件判斷){  
    敘述  
}
```

do...while迴圈：

```
do {  
    敘述  
} while (條件判斷);
```

- 如果執行前已可確定次數，通常會使用for迴圈，不確定次數，通常會使用while或do...while迴圈
- for, while執行0 ~ n次，do...while執行1 ~ n次
- 注意無窮迴圈的問題：無窮迴圈：while(true) {...}

while與do...while的比較

- 1.

```
int i = 100;
while (i <= 10) {
    System.out.println(i);
    i++;
}
```
- 2.

```
int i = 100;
do {
    System.out.println(i);
    i++;
} while (i <= 10);
```
- 3.

```
int count = 1;
while (count++ < 5)
    System.out.println("count = " + count);
// 2, 3, 4, 5 why?
```

數字1加到10時，三種迴圈的比較

範例：[Test3TenAdd.java](#)• 4. **for**

```
int sum = 0;  
for (int count = 1; count <= 10; count++)  
    sum += count;
```

6. **do...while**

```
int sum = 0;  
int count = 1;  
do {  
    sum += count;  
    count++;  
} while (count <= 10);
```

• 5. **while**

```
int sum = 0;  
int count = 1;  
while (count <= 10) {  
    sum += count;  
    count++;  
}
```

雙層迴圈的練習(巢狀迴圈)

範例：[NineNineLoop.java](#)

- 九九乘法表

```
public class NineNineLoop {  
    public static void main(String[] args) {  
        int i, j;  
        for (i = 1; i <= 9; i++) {  
            for (j = 1; j <= 9; j++) {  
                System.out.print(i + "*" + j + "=" + i*j + "\t");  
            }  
            System.out.println();  
        }  
    }  
}
```

- 請建立一個**Test4Numbers.java**，可輸出0~100裡4的倍數
- 請建立一個**TestNineNine.java**程式，可輸出九九乘法表
 - 一：使用**for**迴圈 + **while**迴圈
 - 二：使用**for**迴圈 + **do while**迴圈
 - 三：使用**while**迴圈 + **do while**迴圈

迴圈內的break與continue

- break的作用是**跳離**迴圈
- continue的功能是**跳過**continue以下的敘述，回到迴圈的起始點
- break與continue通常會配合 if 敘述與標籤作使用(見範例)
- 無窮迴圈經常用在「不知道要跑幾次」時使用，並常搭配跳離迴圈的break使用
- break只能在**迴圈**與 **switch case** 內使用

範例：
`TestContinue.java`
`TestBreakContinue1.java`
`TestBreakContinue2.java`

模組9 方法設計與應用

- 9-1：
方法宣告與注意事項
- 9-2：
傳入參數與回傳值
- 9-3：
方法覆載機制

方法宣告

- 定義：
 - 可重複使用的程式碼片段
- 範例：
 - 如 public static void main(String[] args) {...}

main方法為程式的進入點
- 宣告：
 - <modifier>* <return_type> <name> (<parameter>*) {...}
 - 修飾存取方式 回傳值型別 方法名稱 (傳入參數) {...}

方法呼叫

- **呼叫不同類別的方法：**
 - 利用「.」運算子來呼叫不同類別中的方法
 - 例：myPen.showInfo();

- **呼叫相同類別中的方法：**
 - 同一類別中，可從calling method直接叫用worker method
 - 例：

```
public void p(String s) {  
    System.out.println(s);  
}
```



```
public void showInfo() {  
    p("Brand is : " + brand);  
    p("Price is : " + price);  
}
```

傳遞參數與回傳值

範例：
Calculator.java
TestCalculator.java

- worker method
 - 可能需要有參數傳進來
 - 如有回傳值，則不再是void，且需加return

- calling method
 - 傳遞worker method要的參數型別與值
 - 如有回傳值，則可以用適當的型別去接

```
public class Calculator {  
    public int sum(int x, int y) {  
        return x + y;  
    }  
}
```

```
public class CalculatorTest {  
    public static void main(String[] args) {  
        Calculator myCal = new Calculator();  
        int sum = myCal.sum(1, 2);  
        System.out.println(sum); //3  
    }  
}
```

方法覆載機制

範例：[TestOverloading.java](#)

- 基本觀念
 - Overloading 讓我們可以用相同的方法名稱來呼叫相似功能的方法
 - 程式設計者為功能類似的方法提供相同的方法名稱時，Java 會自動依據參數的數量及不同的資料型別，自動呼叫對應的方法，如：
 - public void println(int i)
 - public void println(float f)
 - public void println(String s)
- 注意事項
 - 必須注意的是，Overloading 的方法無法根據回傳值型態的不同而區別，如以下為重複宣告的錯誤：
 - void method(int i)
 - int method(int i) // 重複宣告
 - String method(int i) // 重複宣告

方法覆載程式碼片段

```
1 public class TestOverloading{  
2  
3     //圓形面積  
4     public static double areaMeasure(double radius){  
5         return radius* radius*3.14;  
6     }  
7     //長方形面積  
8     public static double areaMeasure(double height, double width){  
9         return height*width;  
10    }  
11    //梯形面積  
12    public static double areaMeasure(double upper, double bottom, double height){  
13        return (upper+bottom)*height/2;  
14    }  
15  
16    public static void main(String args[]){  
17        double i = areaMeasure(3.0);  
18        double j = areaMeasure(3.0,4.0);  
19        double k = areaMeasure(3.0,4.0,5.0);  
20        System.out.println("圓形面積=" +i);  
21        System.out.println("長方形面積=" +j);  
22        System.out.println("梯形面積=" +k);  
23    }  
24 }
```

方法覆載與Java API

- Overloading & Java API
 - 許多在Java API中的方法都是Overloading的
 - 例如System.out.println(...)，使用上非常方便就是因為Overloading的關係
 - 註：println(...)是PrintStream類別的方法

void	<code>println()</code> Terminates the current line by writing the line separator string.
void	<code>println(boolean x)</code> Prints a boolean and then terminate the line.
void	<code>println(char x)</code> Prints a character and then terminate the line.
void	<code>println(char[] x)</code> Prints an array of characters and then terminate the line.
void	<code>println(double x)</code> Prints a double and then terminate the line.
void	<code>println(float x)</code> Prints a float and then terminate the line.
void	<code>println(int x)</code> Prints an integer and then terminate the line.
void	<code>println(long x)</code> Prints a long and then terminate the line.
void	<code>println(Object x)</code> Prints an Object and then terminate the line.
void	<code>println(String x)</code> Prints a String and then terminate the line.

模組10 認識物件

- 10-1：
理解類別與物件關係
- 10-2：
建立物件進行操作
- 10-3：
使用類別作為資料型別

類別與物件

- 五字箴言：**所見即物件**
- 只要是物件就一定有此兩項：
 - 屬性 (attribute) 或稱為特徵 (Characteristics)
 - 行為 (behavior) 或稱為操作 (Operation)
- Java透過類別(class)實現物件的概念，讓程式設計師能更具體化與直覺的方式進行資料處理
- 類別組成成員：
 - 資料成員 (Data Member → 變數 Variable)
 - 方法成員 (Method Member → 方法 Method)

類別與物件

- 一隻筆為物件的概念：



屬性 (Attributes) 有顏色, 有廠牌, 有價格

行為 (Behavior) 能寫, 能摔, 能丟...

類別與物件

- 先設計類別才能產生物件，**一個物件是由某類別產生的一個實體 (instance)**



筆類別



筆物件



類別與物件

- 使用同一個類別產生不同的物件實體 (instance)

Attributes: brand, price, color...

Method: write, resupply...



brand: 萬寶龍
price: 14,800
color: 黑色



brand: S.K.B
price: 10
color: 藍色

實體化與初始化物件

範例：
Pen.java
PenTest.java

- 流程
 - 宣告
 - <類別名稱> <變數名稱>
 - 如：Pen myPen;
 - 實體化物件
 - 欲產生該物件真正的記憶體空間，必須以 new 關鍵字建立
 - 如：new Pen();
 - 初始化物件
 - 用 = (指定運算子)指派該物件至物件參考變數
 - 如myPen = new Pen();
 - 注意！
 - 物件參考變數(Object Reference Variables)是一個儲存物件在記憶體中位址的變數

```
1 public class PenTest {  
2  
3     public static void main(String[] args) {  
4         Pen p = new Pen();  
5         p.setBrand("Mont Blanc");  
6         p.setPrice(14800);  
7     }  
8 }
```

操作物件屬性與呼叫方法

- 操作資料：最普遍的作法是利用「.」運算子來操作物件的值
 - 如：`myPen.brand = "SKB";`
 - 如：`yourPen.price = 12000.0;`
- 呼叫方法：最普遍的作法是利用「.」運算子來呼叫物件的方法
 - 如：`myPen.showInfo();`
 - 如：`yourPen.showInfo();`

```
1 public class PenTest {  
2  
3     public static void main(String[] args) {  
4         Pen myPen = new Pen();  
5         myPen.brand = "SKB",  
6         myPen.price = 10.0;  
7  
8         Pen yourPen = new Pen();  
9         yourPen.brand = "MontBlanc";  
10        yourPen.price = 12000.0;  
11  
12        myPen.showInfo();  
13        yourPen.showInfo();  
14    }  
15 }
```

實體變數(屬性)預設初始值

- Java會在編譯時給實體變數預設初始值，內容如下：

變數型態	值
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0(D)
boolean	false
char	'\u0000'
類別型態	null

實體變數與區域變數比較

- 內容如下：

	宣告	初始值	存取	生命週期(scope)
區域變數	宣告在方法裡面或是流程控制的區域裡面	沒有預設初始值, 存取之前, 程式設計師要主動給予初始值。可以先宣告後再給值	只能在自己所宣告的區域內使用, 不能跨區直接使用	如同名稱, 跟著自己所屬的區域或方法, 執行時存活, 當該區域或方法執行完畢, 此變數即被釋放
實體變數	宣告在方法外, 類別裡面, 而且沒有 static 關鍵字	宣告後, Java 會自動給予預設初始值, 根據資料型別有不同的初始值。若是想要指定自己的初始值, 只能在宣告的同時指定。	只要在同個類別裡, 可以跨不同方法使用(該方法不可以有 static 關鍵字), 透過物件參考變數存取	如同名稱, 跟著所屬的物件實體創建而存在, 只要該物件實體還在, 此實體變數就會隨著物件存活著, 直到該物件消失才結束

再看類別與物件關係

- Java 程式設計師藉由類別來定義一個物件實體所該具有的屬性與行為
- 再藉由該類別來產生物件實體，進而在程式裡進行操作
- 所以對於該物件實體資料，我們就使用類別作為變數宣告！
- **結論：類別就是物件的資料型別 (type)**
 - 如 **Pen** myPen = new Pen();
 - 如 **Student** david = new Student();

- 請新增一個**Student**類別，並宣告一個屬性為**score**，型別為**int**，另宣告兩個方法分別為**play()**與**study()**，功能如下：
 - **public void play(int hours)**
每休息一個小時，**score**就會減1
 - **public void study(int hours)**
每讀書一個小時，**score**就會加1
- 該類別完成後，請在**main**方法裡創建兩個**student**物件，並藉由呼叫**play()**與**study()**並取得分數是否正確

模組11

物件導向概論

- 11-1：
物件參考變數特性介紹
- 11-2：
理解傳值與傳參考
- 11-3：
物件導向程式語言特性

物件參考(reference)變數

- 物件參考變數(Object Reference Variables)是一個儲存物件在記憶體中位址的變數

不同的記憶體空間

```
Pen myPen1 = new Pen();  
Pen myPen2 = new Pen();
```

相同的記憶體空間

```
Pen myPen1 = new Pen();  
Pen myPen2 = myPen1;
```

- 在記憶體中儲存物件參考變數

- 物件參考變數儲存記憶體位址 (memory address)
- 基本型別變數儲存值 (value)

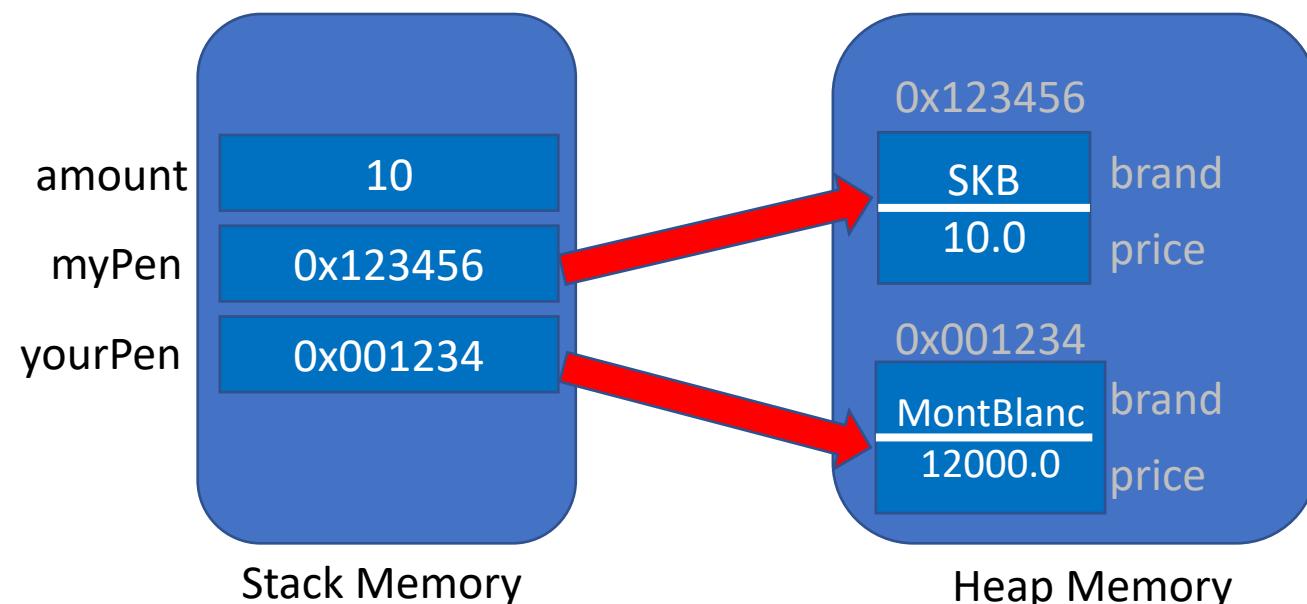
- 注意：

- Java的Pass by value, Pass by reference或是只有Pass by value，在定義上一直有所爭議，將於上課時說明

變數於記憶體運作機制

- 變數儲存在記憶體示意

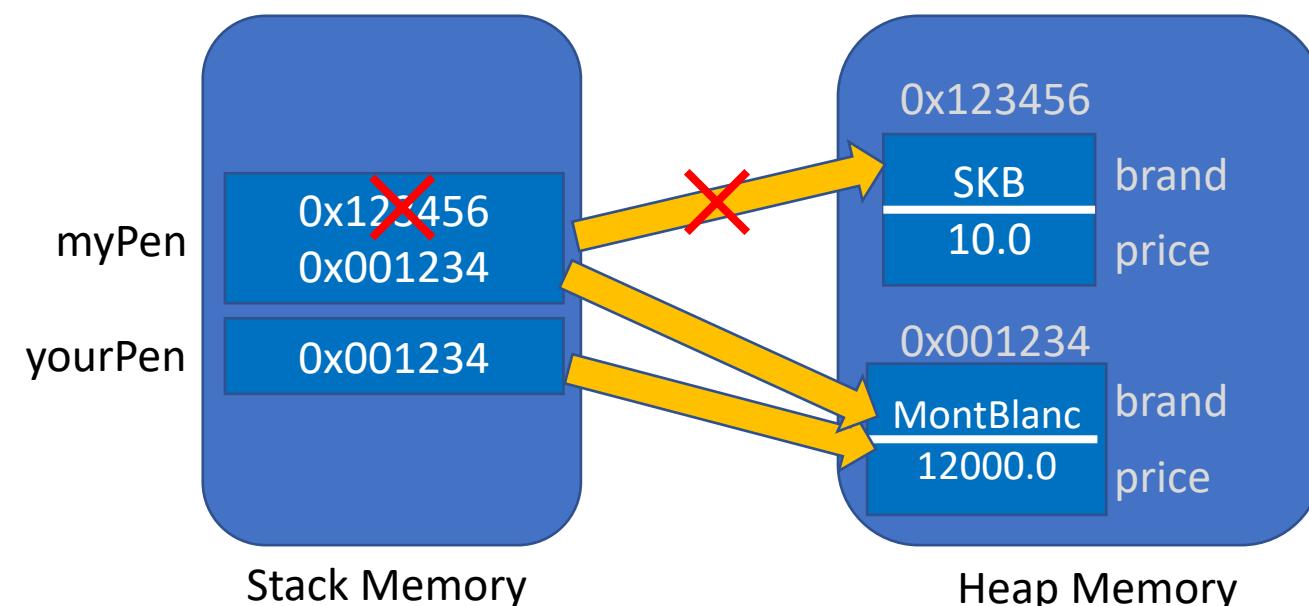
```
public static void main (String[] args) {  
    int amount;  
    amount = 10;  
    Pen myPen = new Pen();  
    Pen yourPen = new Pen();  
}
```



物件參考(reference)變數指定(assign)行為

- 將物件參考變數指定給另一個參考變數

```
Pen myPen = new Pen();
Pen yourPen = new Pen();
myPen = yourPen;
```



傳值與傳址(參考)

範例：PassArgTest.java

- 傳值 Pass by value

- 如果傳入的參數為**基本資料型別**時，在方法內對參數的改變將不影響原來方法外變數的值

- 傳址 Pass by reference

- 如果傳入的參數為**物件參考變數或陣列**時，可修改指向物件的變數值，也可使用指向物件的其它方法與變數

```
1 public class PassArgTest {  
2  
3     static void passValue(double value) {  
4         value = 20.0;  
5     }  
6  
7     static void passReference(Pen reference) {  
8         reference.price = 20.0;  
9     }  
10  
11    public static void main(String[] args) {  
12        double price = 10.0;  
13        passValue(price);  
14        System.out.println(price); // 10  
15  
16        Pen myPen = new Pen();  
17        myPen.price = 10.0;  
18        passReference(myPen);  
19        System.out.println(myPen.price); // 20  
20    }  
21 }
```

物件導向程式語言 (OOP)

- 物件導向程式語言必定有以下三種特性：
 - 封裝 (Encapsulation)
 - 依類別成員存取權限分為private, default, protected與public
 - 繼承 (Inheritance)
 - 子類別可繼承父類別的成員，並可以修改或是新增自有成員
 - 多型 (Polymorphism)
 - 父類別指向子類別物件，並對應到子類別適用的方法
- OOP使用**訊息傳遞(Message Passing)機制**，透過物件接受訊息、處理訊息、傳送訊息來實現功能

整理與探討

- Java透過類別(class)來定義屬性與方法，並用該類別產生的物件實體(object instance)進行功能上的實現與架構設計
- 封裝、繼承與多型為物件導向程式語言的三大特性，我們必須藉由類別來達成以上三種概念，由此可知類別對Java的重要性有多大，我們可以說這是以類別為基礎(class-based)的程式設計
 - 封裝(Encapsulation)讓Java實現隱藏資料與提昇資料存取安全性
 - 繼承(Inheritance)讓父類別所定義的成員可以給子類別使用，子類別更能自行修改父類別定義或是進行擴充，以表現出自身的特質
 - 多型(Polymorphism)讓程式設計師可用同樣方式去引用不同類別的物件，這對我們來說可以易於使用，並進一步到專案維護與功能擴充

模組12 陣列 (1)

- 12-1：
認識陣列與使用目的
- 12-2：
宣告陣列與元素存取
- 12-3：
操作一維陣列

陣列(Array)的定義

- 陣列是由**一群相同資料型態的變數所組成的一種資料結構**
- 比較一個變數與一個陣列
 - 一個變數：`int x = 0;` `Pen myPen = new Pen();`
 - 一個陣列：`int x[] = new int[3];` `Pen myPen[] = new Pen[3];`
 - 程式進入點main方法可以接受零至多個字串當作參數(`String args[]`)傳入，`String args[]`其實就是一個字串陣列

陣列(Array)宣告

- 陣列必須用**new**關鍵字來分配陣列的儲存空間，所以：
 - 陣列也是一種Reference資料型態
 - 陣列的指定運算，也是傳遞陣列的記憶體位址(memory address)
 - 注意：在**new**宣告的同時**必須指定長度且不可再更改**
- 陣列宣告時，中括號可置於陣列參考的前面或後面
 - `int x[];` 或 `int[] x;`
 - `int x[], y[];` 或 `int[] x, y;`

陣列(Array)宣告(續)

- 一維陣列

```
int x[] = new int[3];  
x[0] = 10;  
x[1] = 20;  
x[2] = 30;
```



```
int x[] = {10, 20, 30};
```

```
String s[] = new String[3];  
s[0] = "one";  
s[1] = "two";  
s[2] = "three";
```



```
String s[] = {"one", "two", "three"};
```

```
Pen p[] = new Pen[3];  
p[0] = new Pen();  
p[1] = new Pen();  
p[2] = new Pen();
```



```
Pen p[] = {new Pen(), new Pen(), new Pen()};
```

陣列(Array)元素的存取

- 取得陣列的長度：
 - 語法：陣列名稱.length (如myArray.length)
 - 注意1：一維陣列為元素**個數**
 - 注意2：二維陣列為**列數**
 - 注意3：length後面不可以加上小括弧，因為此處的length並不是方法，而是陣列的一個屬性
(跟String類別的length()不同)
- 取得陣列的元素
 - 可藉由索引值(index)存取陣列中儲存的資料值
 - 注意：**索引值從0開始**
- 陣列使用new關鍵字分配好儲存空間後，**所有元素都會自動賦予初始值**

陣列宣告的預設初始值

- 陣列宣告(沒給自訂初始值時)：

變數型態	值
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0(D)
boolean	false
char	\u0000'
類別型態	null

一維陣列使用

範例：[TestOneDimArray.java](#)

- 以下示範計算一維陣列中所有整數的總合

```
public class TestOneDimArray {  
  
    public static void main(String[] args) {  
  
        int[] intArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
        int sum = 0;  
  
        for (int i = 0; i < intArray.length; i++)  
  
            sum += intArray[i];  
  
        System.out.println("總合為：" + sum);  
  
    }  
  
}
```

請同學們觀察：

- for迴圈的各個要素
- 陣列元素取得的語法(搭配索引值)

模組13 陣列 (2)

- 13-1：
多維陣列
- 13-2：
陣列指定與傳遞
- 13-3：
陣列元素進階操作

多維陣列觀念

- 多維陣列(Java的多維陣列是**陣列的陣列**)
 - `int xx[][] = new int [4][5];`
- 多維陣列也可以如一維陣列般，一邊宣告一邊給初值

2x3陣列：

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

3x2陣列：

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

- 非矩形(non-rectangular)的多維陣列

```
int xx[][] = new int[4][];  
xx[0] = new int [3];  
xx[1] = new int [4];  
xx[2] = new int [5];  
xx[3] = new int [5];
```

二維陣列使用

範例：[TestTwoDimArray.java](#)

- 以下示範計算**二維陣列**中所有整數的總合

```
public class TestTwoDimArray {  
  
    public static void main(String[] args) {  
  
        int[][] intArray = { {1,2,3,4,5}, {6,7,8,9,10} };  
  
        int sum = 0;  
  
        for (int i = 0; i < intArray.length; i++) {  
  
            for (int j = 0; j < intArray[i].length; j++)  
  
                sum += intArray[i][j];  
  
        }  
  
        System.out.println("總合為：" + sum);  
  
    }  
}
```

請同學們觀察：

- 巢狀for迴圈流程
- 二維陣列的length與一維陣列length
- 二維陣列元素取得的語法(搭配索引值)

- 請分別建立x, y, z三個3x3的int陣列，然後把x和y陣列的加總存放到z陣列裡，再將結果顯示於螢幕上
- x和y陣列中的數字：
 - 請用亂數產生介於0 ~ 30之間的整數
 - 亂數之取得可參考 `java.lang.Math`的靜態方法 `random()`
 - `public static double random()`
 - 傳回亂數值其範圍為0.0 ~ 1.0

陣列的指定

範例：[TestAssignArray1.java](#)

- 以下示範陣列的**指定運算(=)**

```
public class TestAssignArray {  
  
    public static void main(String[] args) {  
  
        int[] intArray1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
        int[] intArray2 = intArray1; //將intArray1指定給intArray2  
  
        for (int i = 0; i < intArray2.length; i++)  
            intArray2[i] = 0; //將intArray2的所有元素改成0  
  
        for (int i = 0; i < intArray1.length; i++)  
            System.out.println(intArray1[i]); //列印原來intArray1所有元素，也都跟著變成0  
    } }
```

請同學們觀察與思考：

- int[] intArray2 = intArray1; 意思為何？
- intArray1的初始資料在哪被修改？

陣列的傳遞

範例：[TestAssignArray2.java](#)

- 以下示範傳遞陣列的**記憶體位址(memory address)**

```
public class TestAssignArray2 {  
  
    static void passReference(int[] intArray) {  
  
        for(int i = 0; i < intArray.length; i++)  
  
            intArray[i] = 0;  
    }  
  
    public static void main(String[] args) {  
  
        int[] iArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
        passReference(iArray);  
  
        for (int i = 0; i < iArray.length; i++)  
  
            System.out.println(iArray[i]);  
  
    }  
}
```

請同學們觀察與思考：

- passReference(iArray); 意思為何？
- iArray的初始資料在哪被修改？

陣列元素的排序、搜尋與複製 (Arrays類別)

範例：[TestArrays.java](#)

- 陣列的排序：
 - static void **Arrays**.sort(欲排序的陣列名稱)，可讓該陣列元素由小到大排序
- 陣列的搜尋：
 - static int **Arrays**.binarySearch(陣列名稱, 欲搜索的值)
 - 使用二分搜索法，搜索陣列內某個值的位置 (回傳int)
 - 執行搜索前，必須先將該陣列排序，如果欲搜尋的值不在該陣列裡，則回傳負值
- 陣列的複製：【**JDK 6新增方法：copyOf()**】
 - static 複製的陣列型別 **Arrays**.copyOf(欲複製的陣列名稱，複製的長度)
 - 複製出的新陣列可以不用預先初始化(不用new)，直接回傳(複製出)一個新的陣列

模組14

Java字串

- 14-1：
字串不可變與字串池
- 14-2：
字串比較(==與equals)
- 14-3：
字串常用方法

字串不可變與字串池

- 在HelloWorld.java裡出現的「Hello World!」時，Java用String物件進行處理的
- String物件的幾個特性如下：
 - 不可變的 (immutable)字串：
 - String一旦宣告後，即不能在原所在記憶體位置改變字串內容
 - 使用String類別任何方法時，傳回的字串都會放在新的記憶體空間
 - String s1 = new String("Hello");
 - 有自己的獨立記憶體空間
 - String s1 = "Hello";
 - 為加快程式執行，Java會把此類字串放在**字串池(String Pool)**裡
 - 程式裡若有多個變數，都使用相同的字串常數(如="Hello")，則均會使用相同的記憶體空間(即字串池String Pool)

字串不可變與字串池 (圖解)

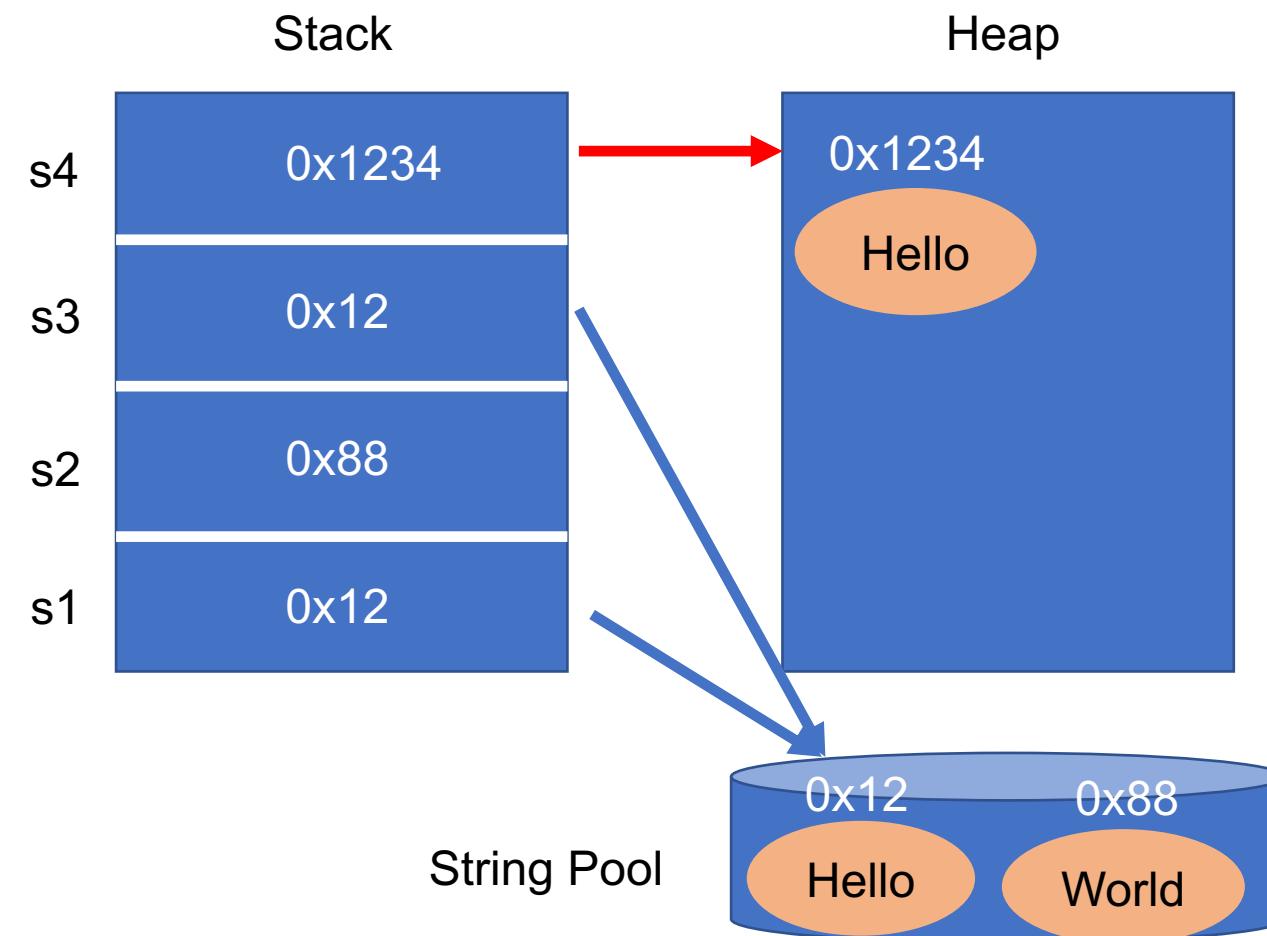
程式碼片段如下：

```
String s1 = "Hello";
```

```
String s2 = "World";
```

```
String s3 = "Hello";
```

```
String s4 = new String("Hello");
```



Java字串的==與equals

範例：[TestString.java](#)

- String的比較：
 - 比較字串內容時，**並非**使用==，因為==在Java字串中，比較的是記憶體位址（指是否佔用相同的記憶體空間）而不是內容
 - 比較字串內容時，應該使用String物件本身提供的一個方法，
`public boolean equals(Object anObject)`
- 比較如下：

<code>String s1 = "Hello";</code>		<code>String s1 = "Hello";</code>	
<code>String s2 = "Hello";</code>		<code>String s2 = new String("Hello");</code>	
<code>System.out.println(s1 == s2);</code>	// true	<code>System.out.println(s1 == s2);</code>	// false
<code>System.out.println(s1.equals(s2));</code>	// true	<code>System.out.println(s1.equals(s2));</code>	// true

列舉String常用方法

- **public char charAt(int index)**：透過索引值取得字串內某一個字元 (注意字元位置是從0開始算起)
- **public int length()**：傳回字串長度，也就是字元的數量 (注意空白也算進去)
- **public boolean isEmpty()**：如果字串長度為0，則回傳true，否則回傳false (JDK 6.0新增方法)
- **public String substring(int beginIndex)**：擷取從開始索引值的字元至結尾字元的字串
- **public String substring(int beginIndex, int endIndex)**：
擷取從開始索引值的字元至結束索引值的字元之間的字串 (注意結束索引值的字元不取)
- **public int compareTo(String anotherString)**：比較的方式是由左至右，依照字元ASCII值比較大小
 - 若回傳值=0，表示兩個字串相等
 - 若回傳值>0，表示左邊字串大於右邊字串
 - 若回傳值<0，表示左邊字串小於右邊字串
- 註：JDK6以前，測試是否為空字串(`String s = ""`)時，須使用`if(s.length() == 0)`；JDK6以後可使用`if(s.isEmpty())`來測試，方便又美觀！

模組15 varargs機制

- 15-1：
main方法命令列參數
- 15-2：
ArrayIndexOutOfBoundsException (預備知識)
- 15-3：
不固定引數機制

main方法參數

範例：[TestMainArray.java](#)

- 程式進入點main方法可以接受零至多個字串當作參數傳入，String[] args 其實就是一個字串陣列，如下所示：
 - java TestMainArray c a t
- 程式中各參數值分別以字串陣列型態 **args[0]**, **args[1]**, **args[2]**...存取

```
public class TestMainArray {  
  
    public static void main(String[] args) {  
  
        System.out.println("貓的英文是：" + args[0] + args[1] + args[2]);  
  
    }  
  
}
```

陣列索引值元素存取問題

- 程式碼裡若是main方法的命令列參數，但執行時卻未提供資料，就會發生ArrayIndexOutOfBoundsException，因為該字串陣列沒有元素，索引值的存取超過該陣列的範圍
- 因為陣列索引值從0開始，所以初用陣列的學習者很容易發生此執行上的例外。例外迴圈次數設計錯誤導致，或是條件設計不良所造成
- 為後面的”例外處理”觀念先有所準備！

varargs

範例：[AddInt.java](#)

- varargs(不固定引數個數 / 可變引數個數)：
 - 方法內可使用【...】點號，宣告【可變數目的參數】
 - 可變參數宣告必須放在參數列的最後面
 - 方法中最多只能有1個不固定參數的宣告，不能有2個或2個以上的不固定參數宣告
- 例如：
 - `void methodTest1(int x, String... args) {...}`
 - 或 `void methodTest2(String... args) {...}`
 - 呼叫methodTest2方法時就可變化如：`methodTest2("xx");` 或 `methodTest2("xx", "yy");`

模組16

使用封裝 (1)

Encapsulation

16-1：
封裝觀念與目的

16-2：
Java存取修飾字

16-3：
封裝設計優點

封裝目的

- 程式設計師設計類別時，即可指定是否讓其它類別可以存取此類別的資料成員或是方法成員
- Java資料封裝的基本就是**類別**
- 封裝做到資料隱藏與存取限制，當我們將設計好的資料與方法包在物件裡，都必須透過該物件的成員方法來對資料進行存取動作，其它程式無法直接對此物件的資料存取
- Java使用了**private, default, protected**與**public**四種存取修飾子做為封裝權限的等級



包裹有封好，寫code沒煩惱！

存取修飾關鍵字

- **public**
 - Visible to the world
 - 所有類別皆能存取
- **protected**
 - Visible to the package and **all subclasses**
 - 相同套件裡的類別或所有其子類別都可以存取
- **default (預設)**
 - Visible to the package
 - 相同套件裡的類別皆可存取
- **private**
 - Visible to the class only
 - 只有該類別內部才可存取

存取修飾關鍵字(續)

- 用表格來看存取修飾子

	the Same Class	the Same Package	Subclass	Universe
public	✓	✓	✓	✓
protected	✓	✓	✓	
default	✓	✓		
private	✓			

- 存取修飾子開放等級由大至小：
 - public → protected → default → private

存取修飾關鍵字(續)

- 存取修飾子適用場合：

	類別	實體變數	方法	建構子
public	✓	✓	✓	✓
protected	-	✓	✓	✓
default	✓	✓	✓	✓
private	-	✓	✓	✓

- 注意1. 類別只有public與default兩種修飾子可使用
- 注意2. 方法變數(區域變數)不能使用存取修飾子，因為沒有意義

封裝設計思維

- 思考方向：
 - 個人
 - 商業
- 運用分析：
 - 類別設計
 - 資料、方法存取控制
- 參考後續說明與情境範例

模組17

使用封裝 (2)

Encapsulation

- 17-1：
封裝範例 – 基本
- 17-2：
封裝範例 – 進一步探討
- 17-3：
變數有效範圍

一般類別封裝設計

範例：
Pen.java
PenTest.java

- 一般而言，設計類別時，建議實體變數(instance variable)設定為private權限，再透過方法來存取資料
- 想想看：筆的資料成員 – price，若是沒做好存取限制，會有什麼後果？

```
1 public class Pen {  
2     private String brand;  
3     private double price;  
4  
5     public String getBrand() {  
6         return brand;  
7     }  
8     public void setBrand(String brand) {  
9         this.brand = brand;  
10    }  
11    public double getPrice() {  
12        return price;  
13    }  
14    public void setPrice(double price) {  
15        this.price = price;  
16    }  
17}  
  
1 public class PenTest {  
2  
3     public static void main(String[] args) {  
4         Pen p = new Pen();  
5         p.setBrand("Mont Blanc");  
6         p.setPrice(14800);  
7  
8         System.out.println(p.getBrand());  
9         //輸出結果為Mont Blanc  
10        System.out.println(p.getPrice());  
11        //輸出結果為14800.0  
12  
13        // 以下編譯失敗，brand與price都設定為private  
14        // 故出現Pen.brand is not visible  
15        // 與Pen.price is not visible 訊息  
16        // System.out.println(p.brand);  
17        // System.out.println(p.price);  
18    }  
19}
```

封裝設計議題

範例：
PenNG.java
PenTestNG.java

- 未使用封裝有什麼情況？

你 完 了!!!!!!

```
1 public class Pen {  
2     //預設牌子為"無牌"  
3     public String brand = "No brand";  
4     //預設售價為0  
5     public double price = 0.0;  
6     //使用show方法顯示出牌子與售價  
7     public void show() {  
8         System.out.println("Brand is:" + brand);  
9         System.out.println("Price is:" + price);  
10    }  
11 }  
12  
13 public class PenTest {  
14  
15     public static void main(String[] args) {  
16         Pen p = new Pen();  
17         //設定牌子為SKB  
18         p.brand = "SKB";  
19         //設定售價為100塊(但你沒注意到多了個負號！！！)  
20         p.price = -100;  
21         //顯示此鋼筆的資訊  
22         p.show();  
23     }  
24 }
```



getter/setter方法

範例：
PenGood.java
PenTestGood.java

- 解決之道三部曲：
 1. private → 2. public getXXX → 3. public setXXX

```
1 public class Pen {  
2     private double price = 0.0;  
3  
4     public double getPrice() {  
5         return price;  
6     }  
7     public void setPrice(double price) {  
8         if (price > 0) {  
9             this.price = price;  
10        } else {  
11            System.out.println("請確認售價設定");  
12        }  
13    }  
14    public void show() {  
15        System.out.println("Price is:" + price);  
16    }  
17 }
```

加入此判斷
問題即可解決

封裝設計總結

- 對於理想的程式碼來說，類別中絕大部份甚至是全部的變數(Variable)都會是用**private**修飾子
- 這表示它們無法直接被自己所屬以外的類別修改或查詢，只能藉由自己類別中的方法來修改或是查看
- 這些方法(如getter/setter方法)應該包含程式碼和運作邏輯以確保變數不會被設定成不適當的值

模組18 建構子 Constructor

- 18-1：**
建構子使用目的與注意事項
- 18-2：**
this關鍵字
- 18-3：**
建構子覆載設計

建構子

- 建構子名稱需與所屬類別名稱相同
- 建構子宣告：
 - [modifier] constructor_name ([parameters]) {...}
 - 一個類別可以有多個建構子
 - 一個建構子可以傳入零至多個參數
 - 建構子類似方法，可以有存取修飾子
 - 建構子沒有宣告回傳型別，加了回傳型別即成為一般方法
- 必須使用 **new** 關鍵字呼叫建構子產生物件，並初始化該物件的實體變數
- 注意：Java會自動給一個不帶參數的建構子，一旦宣告其他建構子，Java會自動將此預設建構子移除

建構子範例

範例：PenConstructor.java

```
1 public class Pen {  
2     public String brand;  
3     public double price;  
4  
5     public Pen (String brandXXX, double priceXXX) {  
6         brand = brandXXX;  
7         price = priceXXX;  
8     }  
9  
10    public static void main(String[] args) {  
11        Pen p = new Pen("SKB", 10);  
12        System.out.println(p.brand);  
13        System.out.println(p.price);  
14    }  
15 }
```

this關鍵字

範例：PenConstructorThis.java

- this關鍵字
 - 用來代表執行時的當前物件
 - 易混淆的程式碼範圍加以區分
 - 建構子覆載設計使用
(參考下一個知識點)

```
1 public class Pen {  
2     public String brand;  
3     public double price;  
4  
5     public Pen (String brand, double price) {  
6         this.brand = brand;  
7         this.price = price;  
8     }  
9  
10    public static void main(String[] args) {  
11        Pen p = new Pen("SKB", 10);  
12        System.out.println(p.brand);  
13        System.out.println(p.price);  
14    }  
15 }
```

- 建立一個class，名為Animal.java
- 此類別有兩個成員變數分別為age(年紀 - 型別int)、weight(體重 - 型別float)
- 此類別需宣告有參數的建構子
- 有一成員方法名為speak()，用以列印上述兩個值
- 在main()裡藉由建構子產生一個Animal，同時初始化該物件的年紀和體重分別為2歲、5.0公斤，並呼叫speak方法得到此Animal的成員變數值

建構子覆載(Overloading)

範例：[PenConstOverload.java](#)

- 可以藉由 `this` 關鍵字呼叫同類別裡的另一個建構子
- 建構子第一個敘述只要有`this()`，則進行其它建構子呼叫執行

```
2     public String brand;
3     public double price;
4
5     public Pen (String brand, double price) {
6         this.brand = brand;
7         this.price = price;
8     }
9
10    public Pen (double price) {
11        this("SKB", price);
12    }
13
14    public Pen (String brand) {
15        this(brand, 10);
16    }
17
18    public Pen () {
19        this("SKB", 10);
20    }
```

模組19 **static**關鍵字

- 19-1：
比較**static**變數與實體
變數
- 19-2：
static運作機制
- 19-3：
如何使用**static**方法

static關鍵字

範例：
Count.java
TestCount.java

- 實體變數和方法若是宣告為 **static**，則此變數和方法即成為**類別變數**(或稱**靜態變數**)和**類別方法**(或稱**靜態方法**)
- 宣告為**static**的變數和方法，不是由任何此類別的物件單獨擁有，而是屬於**此類別的所有物件共同擁有**
 - 補充1：實體變數由物件各自獨立維護，彼此不受干擾
 - 補充2：**static**類別變數是屬於類別的變數，但卻可以由該類別所創造(**new**)出來的物件共享共用
 - 補充3：儲存類別變數和方法的記憶體空間為**global**，與儲存物件的記憶體空間是分開的
 - 補充4：使用**static**變數和**static**方法的方式有兩種：
 - 經由類別的任何實體來呼叫 (不好也不鼓勵使用)
 - **經由類別的名稱來呼叫 (較好的方式)**

static機制

- 當類別第一次被載入JVM時，在任何實體被建構之前，靜態的變數與方法就會先被載入，所以：
 - static方法裡不可使用this
 - 宣告為靜態static方法，不可以存取該類別中non-static的變數和方法，只可以存取該類別中static的變數和方法
 - 宣告為non-static的方法，可以存取該類別中non-static的變數和方法，也可以存取該類別中static的變數和方法

```
1 public class Count {  
2     //產品序號  
3     private int serialNumber;  
4     public static int getSerialNumber() {  
5         return serialNumber; //編譯失敗，static方法裡不得存取non-static變數  
6     }  
7 }
```

static 區塊

範例：
Count2.java
TestCount2.java

```
1 public class Count2 {  
2     //產品序號  
3     private int serialNumber;  
4     public int getSerialNumber() {  
5         return serialNumber;  
6     }  
7     //產品數量  
8     private static int counter;  
9     static {  
10        counter = 0;  
11        System.out.println(  
12            "起始數量：" + counter + "\n");  
13    }  
14    public static int getTotalCount() {  
15        return counter;  
16    }  
17    //建構元  
18    public Count2() {  
19        counter++;  
20        serialNumber = 1000+ counter;  
21    }  
22 }
```



static 程式區塊裡的程式在載入類別時
會先執行一次

static方法與main方法

範例：TestStaticMethod.java

```
public class TestStaticMethod {  
    public static void main(String[] args) {  
        System.out.println("請畫三角形");  
        int count = 9;  
        drawTriangle(count);  
        System.out.println("畫得不錯喔！");  
    }  
  
    public static void drawTriangle(int count) {  
        int i, j;  
        for (i = 1; i <= count; i++) {  
            for (j = 1; j <= i; j++)  
                System.out.print("*");  
            System.out.println();  
        }  
    }  
}
```

注意：
main方法裡呼叫static方法的操作



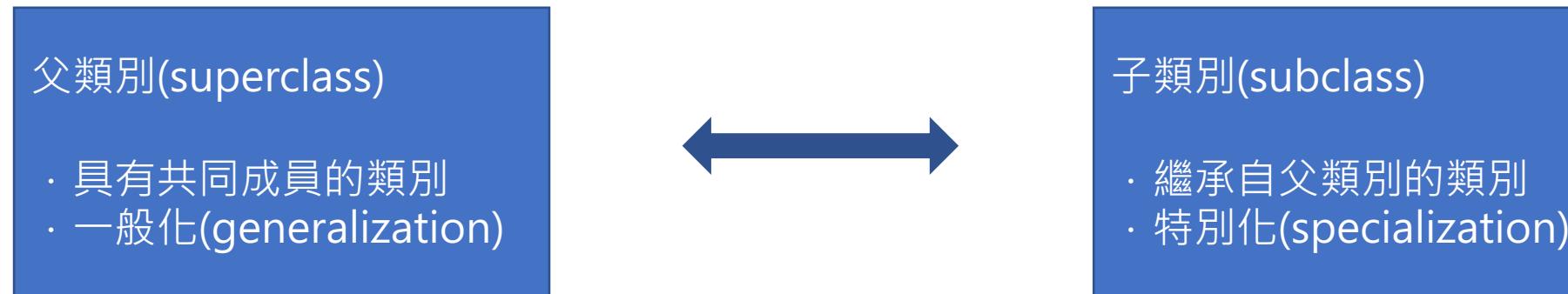
請畫三角形
*
**

模組20 繼承 Inheritance

- 20-1：
繼承目的
- 20-2：
語法與注意事項
- 20-3：
is a與has a的觀念建立

繼承基本概念

- 一個類別可以繼承它的父類別所有狀態及行為，我們即把此稱之為**子類別**(subclass)延伸(extends)自**父類別**(superclass)



繼承優點與好處

- 繼承主要目的就是提高程式的重複使用性
- 子類別將會繼承到父類別中所有可以存取的成員，包括變數與方法
 - **共同資料只要描述一次**
 - 如正職員工(FullTime)與工讀生(PartTime)都有共同資料，如姓名、電話、地址、性別等...
 - **處理共同資料的成員方法也只要描述一次**
 - 如每一個子類別可以使用定義在父類別的成員方法，如員工方法的getName(); getSalary(); 等
- 子類別繼承父類別功能之後：
 - 還可以再加入新方法 (method)
 - 也可以覆寫 override) 從父類別而來的方法，建立一個專屬子類別自己的運作邏輯

繼承語法與注意事項

- 語法：
 - **class SubClassName extends SuperClassName**
 - **class FullTimeEmployee extends Employee {**
 private double monthlySalary; //月薪
 }
- 注意：
 - Java不支援多重繼承，一個子類別只能**extends**一個父類別
 - 建構子(Constructor)無法被繼承
 - **java.lang.Object**類別為所有類別的共同父類別

is a與has a

- 當B繼承自A，以「B **is a** A」表示
- 如：
 - `class FullTimeEmployee extends Employee {
 private double monthlySalary; //月薪
}`
- 則：
 - FullTimeEmployee **is a** Employee
 - FullTimeEmployee **has a** monthlySalary

模組21 使用繼承

- 21-1：
方法覆寫(override)機制
- 21-2：
super呼叫父類別方法
- 21-3：
super呼叫父類別建構子

覆寫目的與規定

- 目的：
 - 子類別繼承父類別後，不滿意父類別定義的方法，子類別可以在繼承後**重新改寫**，即為overriding
- 規定：
 - 子類別宣告覆寫(overriding)方法時，方法名稱、參數個數、參數型別與回傳值(註1)皆須跟父類別裡被覆寫的方法相同
 - 註1：JDK 1.5開始，如回傳型態是類別，則**可以是原方法回傳值型別的子類別**
 - 註2：**存取修飾子的等級不可以小於原方法**(指可以一樣或是更加寬廣) (參考存取修飾子內容)
 - 註3：子類別覆寫父類別定義有throws的方法時，**不得**比父類別被覆寫方法的Exception還要高階 (參考例外處理章節)
 - 註4：後面提到 final 與 static 關鍵字時補充

覆寫存取修飾字規則

- 子類別覆寫(override)父類別的方法時，其存取控制的範圍不可小於原方法
(指的是可以相同或是開放等級更高)
- 例：

```
public class Father {  
    protected void doSomething() {...}  
}  
public class Son extends Father {  
    private void doSomething() {...}          //失敗，private小於protected  
    void doSomething() {...}                  //失敗，default小於protected  
    protected void doSomething() {...}        //成功，與父類別同存取等級  
    public void doSomething() {...}           //成功，大於父類別存取等級  
}
```

再看final關鍵字

- 一個**類別**宣告為**final**，表示這個類別不能被繼承
 - public final class String {...}
 - public final class Math {...}
- 一個**方法**宣告為**final**，表示這個方法不能被覆寫(Override)
- 一個**變數**宣告為**final**，表示這個變數在初始值化後，不得再變更其值，也就是常數(Constant)
- 一個**物件參考變數**宣告為**final**，表示這個變數在初始值化後，不得再指向另一個物件

呼叫父類別的方法

- 子類別透過 **super.** 可以呼叫上一層類別的方法
 - 語法：**super.methodName();**
 - 其中的**methodName()** 為上一層類別的方法
 - 無法越級呼叫

```
1 public class FullTimeEmployee extends Employee {  
2     private double monthlySalary; //月薪  
3  
4     public void display() {  
5         super.display();  
6         System.out.println("月薪=" + monthlySalary);  
7     }  
8 }
```

呼叫父類別的建構子

- 子類別透過建構子，用 **super(...)** 將共同的建構子參數傳給父類別
(指共同的資料應使用父類別的建構子)
- 物件產生時，**建構子呼叫的順序為先父類別再子類別**，所以：
 - 建構子中**若有出現 super(...)**，一定要放在**第一個敘述位置**
 - 建構子中**若未出現 super(...)**，Java預設會有一個隱形的 **super()**
【呼叫父類別不帶參數的建構子】**預設自動放在第一個敘述的位置**

呼叫父類別的建構子

```
public class Employee
    → public Employee (int empno, String ename) {
        this.empno = empno;
        this.ename = ename;
    }
}

public class FullTimeEmployee extends Employee
    private double monthlySalary; //月薪

    → public FullTimeEmployee (int empno, String ename, double monthlySalary) {
        super(empno, ename);
        this.monthlySalary = monthlySalary;
    }
}

public class Manager extends FullTimeEmployee
    private double bonus; //獎金

    public Manager (int empno, String ename, double monthlySalary, double bonus) {
        super(empno, ename, monthlySalary);
        this.bonus = bonus;
    }
}
```

範例：
Employee.java
FullTimeEmployee.java
Manager.java
TestFullTimeEmployee.java
TestManager.java

- 建立一個class，名為Elephant並延伸自Animal類別
- 此類別有一個成員變數為name(名字 – 型別String)
- 此類別需宣告有參數的建構子
- 有一覆寫成員方法名為speak()，用來印出父類別的兩個成員變數和自己的成員變數
- 在main()裡透過建構子產生兩個物件：
 - 其一為Animal，其年紀和體重分別為3歲、8.0公斤
 - 另一為Elephant，其年紀、體重和種類名稱分別為8歲、1200.0公斤、大象
- 列印上述兩種Animal的值

模組22

多型

Polymorphism

22-1：
is-a與多型機制

22-2：
多型宣告語法

22-3：
操作型別轉換

is a與多型(Polymorphism)

- 當B繼承自A，以「B **is a** A」表示，我們對is a會用「是一種」的關係來看待。因此，所謂多型(Polymorphism)就是運用類別間繼承關係，使父類別(superclass)當成子類別(subclass)物件的通用型態
- 如：
 - class FullTimeEmployee extends Employee
 - 正職員工「**是一種**」員工
- 甚至：
 - class Manager extends FullTimeEmployee
 - 經理「**是一種**」正職員工
 - 經理也可以視為「**是一種**」員工

型別多樣化

- 回憶一下基本型別的晉升(promotion)關係：
 - double weight = 60; // OK
 - long uid = 10000; // OK
- 只要符合類別間的繼承關係，在宣告參考變數時，子類別(**位階低**)物件實體可以升級成父類別(**位階高**)型別表示，如：
 - Employee e1 = new FullTimeEmployee(); // OK
 - Employee e2 = new Manager(); // OK
 - Employee e3 = new PartTimeEmployee(); // OK

類別型別轉換(Cast)

- 回憶一下基本型別的強制轉型(Cast)關係：
 - float weight = 65.0f; // OK
 - int i1 = (int)10.0; // OK
 - String s1 = (String) 10; // NG
- 父類別參考變數若是要轉型回子類別，則需要靠強迫轉型(Cast)，但是會在執行時期檢查是否能夠轉回適當的子類別型別
- 如：Employee **e1** = new FullTimeEmployee();
 - FullTimeEmployee f = (FullTimeEmployee)**e1**; // OK
 - Manager m = (Manager)**e1**; //執行發生java.lang.ClassCastException

模組23 多型操作

23-1：
instanceof關鍵字

23-2：
多型資料一致性操作

23-3：
方法動態繫結設計

有關instanceof

- **instanceof** 運算子常被用來作為轉型操作前的檢查，可避免執行時發生ClassCastException
 - 語法：物件參考變數 instanceof 類別名稱
 - 說明：檢查左邊參考的物件是否可以轉型為右邊的類別型別，如果可以回傳true，否則為false
- Employee **e1** = new FullTimeEmployee();
 - System.out.println(**e1** instanceof FullTimeEmployee); // true
 - System.out.println(**e1** instanceof Manager); // false
 - System.out.println(**e1** instanceof PartTimeEmployee); // false

一致性操作

- 對於不同的物件實體，我們找出這些物件實體可通用的型別作為宣告，實現資料操作上的一致性，可以讓程式碼變得更加簡潔，也易於日後資料的擴充設計與維護
- 例如：

```
public class TestPolymorphism2 {  
    public static void main(String[] args) {  
        Employee[] e = new Employee[3];  
        e[0] = new FullTimeEmployee(7002 , "peter", 40000.0 );  
        e[1] = new Manager(7003 , "merry", 50000.0 , 10000.0);  
        e[2] = new PartTimeEmployee(7004 , "John" , 1000.0, 8);  
        for (int i = 0; i < e.length; i++)  
            System.out.println(e[i].getSalary());  
    }  
}
```

動態繫結(dynamic binding)

- 用父類別的型別(參考)，指向子類別的物件，並對應到子類別overriding的方法
 - 根據指向的子類別物件是什麼，再呼叫此子類別裡對應的overriding方法

```
Manager m = new Manager(7003, "David", 50000.0, 10000,0);  
double salary = m.getSalary();
```

```
Employee e = new Manager(7003, "David", 50000.0, 10000.0);  
double salary = e.getSalary();
```

- 以上getSalary()在最後執行時，都是子類別Manager的方法
但父類別的getSalary()還是不可省去，否則無法進行對應造成錯誤

動態繫結(dynamic binding)

```
public class Employee
    private int empno;
    private String ename;
    → public double getSalary() { return 0; }
```

```
public class FullTimeEmployee extends Employee
    private double monthlySalary; //月薪
    → public double getSalary() { return monthlySalary; }
```

```
public class Manager extends FullTimeEmployee
    private double bonus; //獎金
    → public double getSalary() {
        double monthlySalary = super.getSalary();
        return monthlySalary + bonus;
    }
```

範例：
EmployeePoly.java
FullTimeEmployeePoly.java
ManagerPoly.java
PartTimeEmployeePoly.java
TestPolymorphism2.java

動態繫結(dynamic binding)

```
public class Employee
    private int empno;
    private String ename;
    → public double getSalary() { return 0; }
```

```
public class PartTimeEmployee extends Employee
    private double hourlyPay; //時薪
    private int workHour;    //工時
    → public double getSalary() {
        return hourlyPay * workHour;
    }
```

- 產生一個class，名為PolyAnimal.java
- 程式同上一個課堂練習
- 在main()裡透過多型來製作

模組24

抽象機制與目的

Abstract

- 24-1：
抽象機制與目的
- 24-2：
抽象類別與抽象方法
- 24-3：
實作抽象方法

抽象類別 (abstract class)

範例：[TestAbstract.java](#)

- 抽象方法沒有方法主體，且必須加上**abstract**修飾子
 - `public abstract void myMethod();`
- 抽象類別不一定要有抽象方法，但具有抽象方法的類別，一定要宣告為抽象類別
 - `public abstract class MyClass {...}`
- 一個類別只要加上**abstract**修飾子(即使它裡面不含任何**abstract**方法)，它就無法產生實體，只能透過繼承來建立延伸子類別，而該類別若繼承了抽象父類別，除非它實作了抽象父類別當中的所有抽象方法，否則它仍然只是個抽象類別
- 使用時機 (上課詳述)
 - 在建立類別時，若有方法尚未決定如何設計內容主體時，就可將此方法加上**abstract**修飾子成為抽象方法，之後再由繼承的子類別來實作

模組25

介面

Interface

25-1：
介面語法

25-2：
介面方法與資料特性

25-3：
介面與資料型別

介面 (Interface)

- Java使用介面(interface)的主要五大功能：多重繼承、定義規格、貼標籤、型別轉換、降低相依性
- **多重繼承**
 - Java只能單一繼承，而介面可以實現物件導向中的多重繼承
(替代C++中的多重繼承)
 - class 子類別 **extends** 父類別 **implements** 介面1, 介面2, ... {...}
 - class 子類別 **implements** 介面1, 介面2, ... {...}
- 預先**定義規格**給實作此介面的所有子類別
 - 介面可說是一種所有方法皆為抽象方法的抽象類別，所以子類別必須實作介面的所有抽象方法
 - 而介面跟介面之間是可以再繼承(**extends**)的

介面 (Interface)

範例：[TestInterface.java](#)

- 介面裡面宣告的資料，預設由編譯器自動加入以下三個修飾關鍵字：
 - **public static final**
 - 這代表在介面裡宣告的資料實為常數
- 介面裡面宣告的方法，預設由編譯器自動加入以下兩個修飾關鍵字：
 - **public abstract**
 - 這代表在介面裡宣告的方法強制為抽象方法
- 因為介面做為定義規格的用途，所以對任何實作介面的類別來說，資料需為一致，也就是所謂的”標準”(既然是實作同一個介面，理所當然地從介面得到的資料都是相同的)。但方法是可以在各個實作類別裡自行完成

介面 (Interface)

- Java使用介面(interface)的主要目的(續)：
- 因為介面對Java來說是個規格較特殊的類別(class)，所以：
 - 介面也是一種參考型別，也就是說介面提供了另一種彈性，使子類別在繼承原父類別的特性之外，也能具有其他型別的特性
 - 因為一個物件可以實作多個介面，所以每一個父介面都可以當作此物件的(父)多型之一，也因此用介面來幫物件作型態轉換將是一件容易的事情

模組26

介面設計與多型

26-1：
介面與多型

26-2：
介面與程式相依性關係

26-3：
空介面

介面 (Interface)

- 介面與多型比起繼承又更有彈性，我們將類別之間從is-a的關係，轉化成**有共同行為**的關係，例如鳥類、飛機與超人雖然沒有現實生活上的”是一種”關係，但他們**都具備了”飛”的行為**，所以也能達到資料一致性的操作
- 降低相依性的觀念範例，見比較性範例：
 - 低凝聚性 - 高相依性：[Pencil.java](#), [InkBrush.java](#), [WorkWithPens.java](#),
[WriteBusinessTest.java](#)
 - **高凝聚性 - 低相依性**：[IWritable.java](#), [Pencil2.java](#), [InkBrusch2.java](#), [WorkWithPens2.java](#),
[WriteBusinessTest2.java](#)
- 相依性也可稱為耦合性(Coupling)，而凝聚性也稱為內聚性(Cohesion)

空介面 (Tag interface)

- 沒有定義任何方法的介面叫做空介面
- 一個類別可以implements某個空介面，就不需要實作任何方法，但該類別的物件實體即已經成為該介面的一個合法實體
- java.lang.**Cloneable** 和 java.io.**Serializable**是比較有名的兩個空介面
 - 一個類別implements前者java.lang.Cloneable空介面時，該類別的物件才可以做**物件的複製**
 - 一個類別implements後者java.io.Serializable空介面時，該類別的物件才可以做**物件的序列化**
(將物件永久儲存(persistence)，稱做序列化)

修飾子適用的場合

修飾子	類別	實體變數	方法	建構子	程式區塊
public	✓	✓	✓	✓	-
protected	-	✓	✓	✓	-
default	✓	✓	✓	✓	-
private	-	✓	✓	✓	-
final	✓	✓	✓	-	-
static	-	✓	✓	-	✓
abstract	✓	-	✓	-	-

模組27

套件介紹

package

- 27-1：
套件管理機制
- 27-2：
Java常用套件
- 27-3：
套件與編譯/執行關係

Java原始檔案格式

- 在Java檔案中可能會出現三個稱為編譯單元(compilation units)的元素，這些元素皆非必要，但，如果有這些元素，則一定要依以下順序出現：
 - 1. package 宣告
 - 2. import 引用敘述
 - 3. class 類別
- 例如：
 - 1. package myPackageName;
 - 2. import yourPackageName1.*;
 - 3. import yourPackageName2.*;
 - 4. class MyClass {...}

套件(package)

- Java提供套件(Package)的機制，它就像一個管理容器，可以將所定義的名稱區隔管理在package底下，而不會有類別名稱相互衝突的情況發生
- Java的package被設計為與檔案系統結構相對應，而以檔案管理的觀點著手，將性質相似之類別集合在一起
 - 例如java.sql表示名稱java的目錄底下有一個子目錄名叫sql，其內存放的都是Java資料庫連線相關的類別檔
- 套件宣告：
 - 宣告於原始檔案的第一行
 - package 套件名稱(myPackageName);
 - package com.ibm;

Java API與常用套件

- Java標準API裡有許多已經設計好的類別與其相關的內容方便我們程式設計師可以更輕鬆快速實現所需要的功能和應用，所以對於常用的類別應該熟記相關套件
- 如：
 - java.lang
 - java.io
 - java.net
 - java.sql
 - java.util
 - java.util.function (Java 8新增)

套件編譯與執行

- 所有屬於 myPackageName 類別庫的.class 檔案都必須儲存在 myPackageName 資料夾下，若不使用 package 告知，Java 預設會將類別檔置於目前工作環境所在的目錄中
- 因為 source 檔(.java 檔)與.class 檔不一定要放在同一個目錄下，所以使用 package 告知時，必須在編譯時透過 -d option，指定類別檔要置於哪個目錄之中

編譯：`javac -d . HelloWorld.java`

【註：「.」指編譯後的 class 檔置於目前的目錄位置】

執行：`java packageName.HelloWorld`

【註：要在原來的目錄下執行】

模組28 **import & classpath**

28-1：
import使用目的
28-2：
static import功能
28-3：
類別路徑設定與說明

import (引用) 套件

- 關鍵字 (keyword) 中的 **import** 可用來引入 API 中的功能，或是自行定義的套件 (package)
- Java 會自動引用的兩個套件：
 - `java.lang.*`：常用的類別，如 `String` 類別已置於此套件中
 - 目前工作環境所在的 package
- 若使用上述之外的其它套件，則必須用 **import** 引用敘述
 - 如：**import java.xxx.*;**
 - 註：不包含其子目錄的類別
如 `import java.xxx.yyy.*;` 是引用不同的套件

import (引用) 套件或特定類別

範例：TestImport.java

- 引用套件中**所有類別**：

➤ **import java.sql.*;**

```
Date date = new Date(...);
```

- 引用套件中**特定的類別**：

➤ **import java.sql.Date;**

```
Date date = new Date(...);
```

- 如不使用**import**敘述，則必須使用類別長名稱

➤ **java.sql.Date date = new java.sql.Date(...);**

➤ **java.util.Date date = new java.util.Date(...);**

static import

範例：[TestStaticImport.java](#)

- 靜態引用套件 【JDK 5加入的功能】
 - 靜態引用可導入類別內的所有static fields與static methods，亦即使用這些static members無需再指定其類別名稱
 - 使用 * 星號字元可導入類別內所有靜態成員
- 如：
 - `import static java.lang.Math.*;`
.....
`r = sin(PI * 2); //等於 r = Math.sin(PI * 2);`
- 避免過度使用static import功能，否則容易造成混淆而不利於維護

類別路徑(classpath)

- classpath可以讓Java應用程式在編譯和執行時，可以找到要用的相關類別
- 根據JDK文件說明，Java以下面三類classpath順序，依序找尋所需的class

1. Bootstrap classes(Core classes)：

- Java2 Platform核心類別函式庫
- 現有，已置於%JAVA_HOME%\jre\lib\rt.jar檔案，JDK預設會自動載入，不必額外再作設定

2. Extension classes

- Java2 Platform擴充的類別函式庫
- 指的是%JAVA_HOME%\jre\lib\ext目錄下的jar檔或zip檔
- JDK預設會自動載入此目錄內所有的jar檔或zip檔，不必額外再作設定

類別路徑(classpath)

3. Users classes

- 是使用者自己寫的類別函式庫(third-party的類別函式庫也可)，必需額外作設定，JDK才會載入類別
- 是指我們在環境變數classpath設定路徑下的classes或jar檔

如：在作業系統的環境變數，預先新增classpath變數：.;C:\myLib\xxx.jar;C:\myLib\yyy.jar;C:\myClass;

或如：在命令列中：`set classpath = %classpath%;C:\myLib2\zzz.jar;C:\myClass2;`

或如：在編譯及執行時：

```
javac –classpath "%classpath%;C:\myLib2\zzz.jar;C:\myClass2;" HelloWorld.java
```

```
java –classpath "%classpath%;C:\myLib2\zzz.jar;C:\myClass2;" HelloWorld
```

(或-cp)

JDK 6可以一次簡化為
C:\myLib*

模組29

Object類別

29-1 :
equals方法

29-2 :
toString方法

29-3 :
finalize與clone方法

所有Java類別共同父類別 - Object

- Java的所有類別，全部繼承自java.lang.Object類別
- 若一類別無繼承任何類別，則Java會自動用Object類別作為此類別的父類別
- Object類別常用的方法：
 - 為何Java要在(共同父類別)Object類別預備這些方法，其重要事項與觀念在上課時說明
 - boolean equals(Object obj)
 - String toString()
 - protected void finalize()
 - final void wait() notify() notifyAll() 【屬於執行緒的部份】

模組30

包裝類別介紹

Wrapper Class

- 30-1：
包裝類別說明
- 30-2：
包裝類別常用方法
- 30-3：
自動裝箱 / 拆箱功能

包裝類別(Wrapper Class)

- Java每一個基本資料型態，都有一個相對應的Wrapper類別(包裝類別)

基本資料型態(Primitive Type)	Wrapper Class(包裝類別)
整數型態	
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
浮點數型態	
float	java.lang.Float
double	java.lang.Double
其它型態	
boolean	java.lang.Boolean
char	java.lang.Character



為java.lang.Number的子類別

使用包裝類別

範例：[TestWrapper.java](#)

- boxing：將基本型別，置入相對應的包裝類別物件裡
 - `Integer i = new Integer(1);`
- unboxing：從相對應的包裝類別物件取其值
 - 使用 `xxxValue()`方法 如：`int x = i.intValue();`
 - 使用 `static method parseXXX(String s)` 如：`int i = Integer.parseInt("1");`
 - 使用 `static method valueOf(String s)` 如：`Integer i = Integer.valueOf("1");`
- 比較兩個物件是否相等
 - 使用 `boolean equals(Object obj)`

記得其它如
float, double...等
使用均以此類推

自動裝箱 / 拆箱機制 (Auto-boxing / unboxing)

範例：[TestAutoboxing.java](#)

- Autoboxing/Unboxing
 - 如果我們想將像是 int 的基本資料型別放到 Collection 中的話要怎麼辦呢？

- **Autoboxing(自動裝箱)：**
基本資料型別自動轉為包裝型態(Wrapper Types)，如int轉Integer

- **Unboxing(自動拆箱)：**
包裝型態自動轉為基本資料型別，例如Integer轉int

```
33 public class Autoboxing1 {  
34  
35 public static void main(String[] args) {  
36     Integer i1 = 1; //boxing  
37     int i2 = i1;    //unboxing  
38  
39     int sum1 = i1 + i2;  
40     Integer sum2 = i1 + i2;  
41     System.out.println(sum1);  
42     System.out.println(sum2);  
43 }  
44 }
```

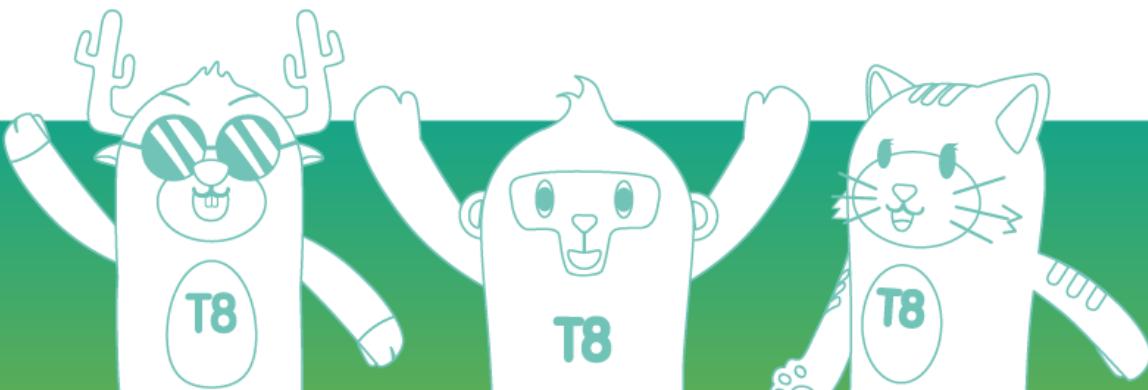
Java物件導向程式設計(實務應用)

授課講師

吳冠宏

教材編寫

吳冠宏



緯
育 *TibaMe*

即學 · 即戰 · 即就業

<https://www.tibame.com/>

課程大綱

- ◆ 模組1：例外機制
- ◆ 模組2：例外處理操作 (1)
- ◆ 模組3：例外處理操作 (2)
- ◆ 模組4：進階例外處理
- ◆ 模組5：改良例外處理機制
- ◆ 模組6：File類別
- ◆ 模組7：循序存取媒體I/O (1)
- ◆ 模組8：循序存取媒體I/O (2)

課程大綱

- ◆ 模組9：循序存取媒體I/O (3)
- ◆ 模組10：循序存取媒體I/O (4)
- ◆ 模組11：循序存取媒體I/O (5)
- ◆ 模組12：物件輸入與輸出
- ◆ 模組13：認識集合
- ◆ 模組14：使用集合物件 (1)
- ◆ 模組15：使用集合物件 (2)
- ◆ 模組16：泛型機制

課程大綱

- ◆ 模組17：加強迭代操作
- ◆ 模組18：集合進階操作 – 排序
- ◆ 模組19：集合進階操作 – 唯一性
- ◆ 模組20：集合與資料結構
- ◆ 模組21：多執行緒設計 (1)
- ◆ 模組22：多執行緒設計 (2)
- ◆ 模組23：控制執行緒與優先安排
- ◆ 模組24：多執行緒同步

課程大綱

- ◆ 模組25：多執行緒溝通
- ◆ 模組26：死結問題
- ◆ 模組27：數字與文字資料
- ◆ 模組28：Regular Expression
- ◆ 模組29：日期時間資料 (1)
- ◆ 模組30：日期時間資料 (2)
- ◆ 模組31：日期時間資料 (3)
- ◆ 模組32：System類別

課程大綱

- ◆ 模組33 : Runtime類別
- ◆ 模組34 : 列舉類型
- ◆ 模組35 : 內部類別 (1)
- ◆ 模組36 : 內部類別 (2)
- ◆ 模組37 : Java8 – Lambda語法
- ◆ 模組38 : Java8 – 函式介面
- ◆ 模組39 : Java8 – Stream API (1)
- ◆ 模組40 : Java8 – Stream API (2)

課程大綱

- ◆ 模組41 : Java8 – Stream API (3)
- ◆ 模組42 : Java8 – 日期/時間API
- ◆ 相關補充附件



授課講師介紹

授課講師

吳冠宏

簡歷

Oracle Certified Java Trainer

OCPJP 8

OCPWCD 6

聯絡方式

vladylo98@gmail.com

專長

Java SE,

Java EE,

Android APP

Design Pattern

學習本課程須知

先備知識

- A. 已經上過Java物件導向程式設計基礎課程
- B. 具備基礎物件導向設計觀念
- C. 對基礎物件導向程式碼有閱讀理解的能力

學習目標

- A. 熟悉Java程式例外處理操作
- B. 藉由集合操作關聯到資料結構的認知
- C. 瞭解多執行緒程式設計與注意事項
- D. 對Java常用資料類型處理作法的認識
- E. 對Java各版本機制功能使用與特性有所認識與接觸

學習本課程須知

學習方式

- A. 上課前預習
- B. 下課後複習
- C. 課堂練習
- D. 閱讀與理解範例程式碼
- E. 課後作業練習

須完成哪些作業或考試

- A. 各模組對應作業題目

模組1 例外機制 **Exception**

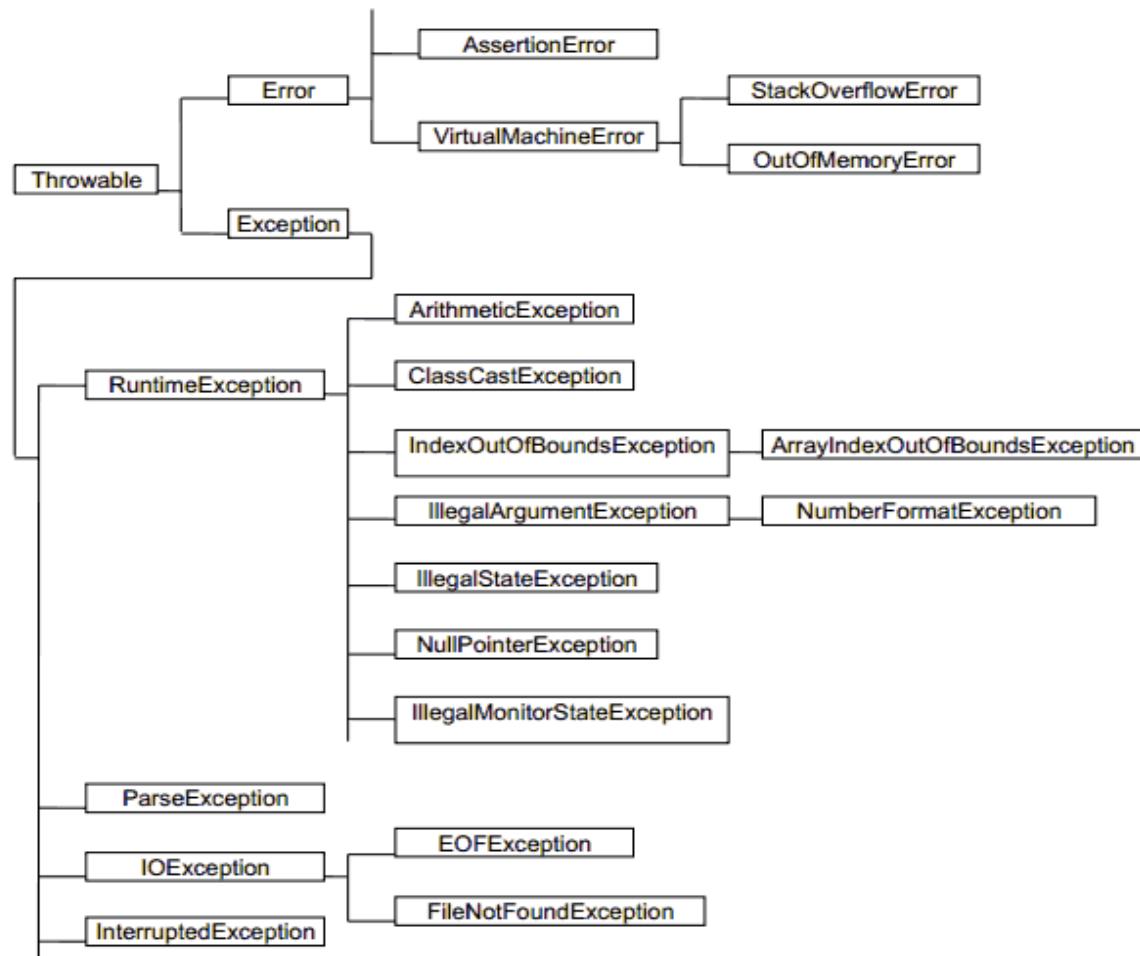
- 1-1：
Exception與Error差異
- 1-2：
例外API類別層級架構
- 1-3：
try...catch使用

Exception物件

- Java程式執行時，如果發生異常狀況，可借助例外處理
 - 程式發生(產生)例外時，視為產生了一個**Exception**物件
- Java的例外處理可將一般**正常處理程序**與**錯誤處理程序**做到**分開敘述**，讓程式可讀性提高
 - 5個關鍵字：**try**、**catch**、**finally**、**throws**與**throw**
 - 在沒有支援例外處理的程式語言裡，錯誤必須自行檢查，然後再手動處理，可能要用到很多**if...else**，非常麻煩
- 適合使用的時機：如資料庫連結失敗、找不到檔案、除以0、參考變數為空值、資料格式不相符、陣列索引超出範圍等...

例外的類別層級架構

Throwable 的階層架構圖



try - catch

範例：[TestTryCatch.java](#)

- Java的例外處理：

```
try {  
    正常處理程序的程式碼...;  
} catch (MyException e) {錯誤處理程序的程式碼...;}  
catch (Exception e) {錯誤處理程序的程式碼...;}
```

- try {}：將**正常處理程序**的程式碼置於 try {} 的程式區塊中
- catch (MyException e) {}：當捕捉到例外物件時將**錯誤處理程序**的程式碼置於 catch {} 的程式區塊中
- try {} 程式區塊後應緊接著 catch {} 程式區塊
- try {} 程式區塊後可接多個 catch {} 程式區塊，至類別位階**高者**需置於類別位階**低者**的**後面**

模組2 例外處理操作 (1)

- 2-1：
finally關鍵字
- 2-2：
受檢與非受檢例外
- 2-3：
常見Runtime Exception

try – catch - finally

範例：
[TestTryCatchFinally.java](#)

- 將**一定要執行的程式碼**放在 **finally { }** 的程式區塊裡
- **finally { }** 是無論發生什麼情況**皆會執行**的程式區塊，可用來釋放有限的資源，例如關閉(close)資料庫連線、檔案讀取等
- **finally { }** 置於**所有catch { }**的後面

```
try {  
    正常處理程序程式碼...;  
} catch (MyException e) {  
    錯誤處理程序程式碼...;  
} catch (Exception e) {  
    錯誤處理程序程式碼...;  
} finally {  
    一定要執行的程式碼...;  
}
```

Checked / Unchecked Exception

- Java有兩種不同型態的例外(Exceptions)
 - 執行時期的例外
 - 這一類的例外，不一定要處理
 - 也稱為不必檢查的例外 (**unchecked exceptions**)
 - 例如 **RuntimeException** 及其子類別
 - 非執行時期的例外
 - 這一類的例外，一定要處理
 - 也稱為必須檢查的例外 (**checked exceptions**)
 - 例如 **IOException**、**SQLException**...等

常見Runtime Exception介紹

例外(Exception)的類別名稱	例外狀況說明
ArithmaticException	分母為 0 時
NullPointerException	呼叫到一個空值 null 變數
IllegalArgumentException NumberFormatException	傳入的參數型態不符 傳入的數字型態不符(前者的子類別)
IndexOutOfBoundsException ArrayIndexOutOfBoundsException	索引值超過物件上限 索引值超過陣列上限(前者的子類別)
ClassCastException	類別的轉型(Casting)失敗
SecurityException	違反命名安全原則

模組3 例外處理操作 (2)

- 3-1 :
throws關鍵字
- 3-2 :
throw關鍵字
- 3-3 :
取得與理解例外訊息

throws關鍵字使用方式

- 在方法定義時，可使用 **throws** 關鍵字將可能發生的例外，丟出給呼叫此方法的程式去處理，用法如下：
 - void method() **throws** MyException {...}
 - public static int parseInt(String s) **throws** NumberFormatException {...}
 - public int read() **throws** IOException {...}
 - public static Connection getConnection() **throws** SQLException {...}
- 對於**checked exceptions**，在呼叫有 **throws** 關鍵字的方法時，**必須**將該方法置於下列兩者之一：
 - 將該方法置於 **try { } 程式區塊中**
 - **(或)將該方法置於定義有 throws 關鍵字的方法中**
 - 此為再透過**throws**丟出例外，然後再由下一個呼叫者來處理

throws關鍵字使用方式

範例：TestThrows1.java

```
1 public class Test3_throws {  
2     String[] strs = {"Hello1", "Hello2", "Hello3"};  
3  
4     public void printStrs(int i) throws Exception {  
5         System.out.println(strs[i]);  
6     }  
7  
8     public static void main(String[] args) {  
9         int i = 0;  
10        Test3_throws t3 = new Test3_throws();  
11        while(i < 4) {  
12            try {  
13                t3.printStrs(i);  
14            } catch (ArrayIndexOutOfBoundsException e) {  
15                System.out.println("1-已超出陣列的長度");  
16            } catch (Exception e) {  
17                System.out.println("2-發生Exception");  
18            }  
19            i++;  
20        }  
21    }  
22 }
```

將該方法置於 try {}
程式區塊中

throws關鍵字使用方式

範例：TestThrows2.java

```
1 public class Test4_throws {  
2     String[] strs = {"Hello1", "Hello2", "Hello3"};  
3  
4     public void printStrs(int i) throws Exception {  
5         System.out.println(strs[i]);  
6     }  
7  
8     public static void main(String[] args) throws Exception {  
9         int i = 0;  
10        Test4_throws t4 = new Test4_throws();  
11        while(i < 4) {  
12            t4.printStrs(i);  
13            i++;  
14        }  
15    }  
16 }
```

將該方法置於定義有
Throws關鍵字的方法中

throw關鍵字使用方式

- 可使用 throw 關鍵字，將**方法內的例外手動丟出**
- throw的指令格式：
 - throw 「一個可被丟出的物件」
 - 該物件必須是 `java.lang.Throwable` 類別的子類別
- 多用於方法設計與自訂例外時使用

throw關鍵字使用方式

範例：TestThrowDemo.java

```
1 public class Test5_ThrowDemo {  
2     public static double method(double i, double j) throws ArithmeticException {  
3         double result;  
4         if (j == 0) {  
5             throw new ArithmeticException("喂！除到0！算數錯誤！");  
6         }  
7         result = i / j;  
8         return result;  
9     }  
10    public static void main(String[] args) {  
11        try {  
12            System.out.println(method(1, 0));  
13        } catch (ArithmeticException e) {  
14            System.out.println(e.getMessage());  
15            //或  
16            e.printStackTrace();  
17        }  
18    }  
19 }  
java.lang.ArithmetricException: 喂！除到0！算數錯誤！  
at Test5_ThrowDemo.method(Test5_ThrowDemo.java:5)  
at Test5_ThrowDemo.main(Test5_ThrowDemo.java:13)
```

喂！除到0！算數錯誤！



例外訊息取得相關方法

範例：[TestStackTrace.java](#)

- 取得錯誤訊息的方法 (Throwable 類別所定義)

取得錯誤訊息的方法	說明
String getMessage()	Returns the detail message string of this throwable
void printStackTrace()	Prints this throwable and its backtrace to the standard error stream
void printStackTrace (PrintStream s)	Prints this throwable and its backtrace to the specified print stream 至指定的輸出設備
String toString()	Returns a short description of this throwable 簡短描述

模組4 進階例外處理

- 4-1：
自訂例外
- 4-2：
Exception與覆寫
- 4-3：
例外處理執行流程

自訂例外設計

範例：[MyException.java](#)
[TestMyException.java](#)

- 要自訂例外類別時：
 - 必須繼承 `Throwable` 或 `Exception` 或 `RuntimeException` 之一
 - 自訂的例外類別，通常會包含兩個建構子：
 - `public 建構子名稱 () {}`
 - `public 建構子名稱 (String message) {`
 `super(message);`
 `}`
- 程式內一樣可利用 `throw` 關鍵字，將例外拋給負責處理此例外的 `catch {}` 區塊處理

- 請建立一個正立方體Cube.java檔案，並定義邊長屬性(double length)，建構子(Constructor)與getter/setter方法
- 產生一個cube物件並同時傳入邊長值，若是值為0或負數，則拋出自行定義的例外CubeException，並顯示「正立方體邊長不得為0或是負數」的訊息
- 若是傳入邊長的值沒有問題，則顯示體積

例外拋出方法與覆寫關係

- 子類別覆寫其父類別定義有 **throws** 的方法時，子類別所 **throws** 的 **Exception** 必須與父類別被覆寫方法的 **Exception 一樣或是更低階**

1. public class BaseClass {
 public void method() throws IOException {}
2. public class OK_A extends BaseClass {
 public void method() throws IOException {}}
3. public class OK_B extends BaseClass {
 public void method() {}}
4. public class NG_C extends BaseClass {
 public void method() throws Exception {} // Error!

Exception flows

```
public class ExceptionFlow {  
    public static void main(String[] args) {  
        try {  
            method();  
            System.out.println("output 0");  
        } catch (Exception1 e1) {  
            System.out.println("output 1");  
        } catch (Exception2 e2) {  
            System.out.println("output 2");  
        } finally {  
            System.out.println("output 3");  
        }  
        System.out.println("output 4");  
    }  
}
```

請同學思考以下三種情況輸出結果：

1. method()執行一切正常
2. method()執行發生Exception1
3. method()執行發生Exception3
(Exception3與1, 2沒有繼承關係)

Exception flows

範例：[ExceptionFlow2.java](#)

- 若method()執行成功，未發生任何例外錯誤，則程式輸出結果為 output 0, output 3, output 4
- 若method()發生 **Exception1** 的例外錯誤，則程式輸出結果為 output 1, output 3, output 4
- 若method()發生 **Exception1, 2 以外**的例外錯誤，則程式輸出結果只有 output 3

模組5 改良例外處理

- 5-1：
例外多重捕捉
- 5-2：
例外類型拋出精確指定
- 5-3：
Assertion機制

例外類型多重捕捉

範例：[TestMultiCatch.java](#)

- 以往catch區塊只能處理一個例外，Java 7開始一個catch區塊可以處理一個以上的例外類型，可以有效地精簡程式碼過於冗長的撰寫
- 注意：catch括號內的例外類型不可以有繼承關係**

```
28 // before, 一個catch區塊只能處理一個例外類型
29 try {
30     methodA("A");
31     methodB("B");
32 } catch (ExceptionB e) {
33     e.printStackTrace();
34 } catch (ExceptionA e) {
35     e.printStackTrace();
36 }
```



```
38 // Java 7, 一個catch區塊可以處理一個以上的例外類型，可以精簡程式碼
39 try {
40     methodA("A");
41     methodB("B");
42 } catch (ExceptionA | ExceptionB e) {
43     e.printStackTrace();
44 }
```

改良重新拋出例外的類型檢查

範例：
[TestImprovedRethrow.java](#)

- 編譯器可以更精確地分析需拋出的例外類型，在方法宣告的throws子句中可指定更多明確的例外型別

```
5  static class ExceptionA extends Exception { }
6  static class ExceptionB extends Exception { }
7  // before
8  public static void methodA(String exceptionName) throws Exception {
9      try {
10          if (exceptionName.equals("A")) {
11              throw new ExceptionA();
12          } else {
13              throw new ExceptionB();
14          }
15      } catch (Exception e) {
16          throw e;
17      }
18  }
19
20 // Java 7
21 public static void methodB(String exceptionName) throws ExceptionA, ExceptionB {
22     try {
23         if (exceptionName.equals("A")) {
24             throw new ExceptionA();
25         } else {
26             throw new ExceptionB();
27         }
28     } catch (Exception e) {
29         throw e;
30     }
31 }
```

斷言機制 (補充)

範例：[TestAssertion.java](#)

- 什麼是Assertion
 - 用來維護程式使之更堅固(robust)，零錯誤
 - Assertion通常用來檢查一些關鍵的值，避免這些值有錯誤時，讓程式無法繼續執行
- Assertion語法
 - **assert <boolean_expression>**：
 - 當boolean_expression為 false 時，會丟出AssertionError，程式即中斷
 - **assert <boolean_expression> : <detail_expression>;**
 - 當boolean_expression為 false 時，會執行後面的運算式，最常用為字串，以說明錯誤的原因
 - 如：assert obj != null：“這物件不得為null”；
 - 如：assert k != 0：“k值不得為0”；
- 執行：`java -ea TestAssertion`

模組6

File類別

- 6-1：
File類別與建構子
- 6-2：
File類別常用方法
- 6-3：
檔案絕對路徑/相對路徑

File類別

- java.io.**File**類別：
 - **File**這個類別為檔案及目錄提供了對應的Java物件
 - 可以用此類別來建立、移除檔案，或變更檔案的屬性...等
- 使用**File**建構子時：
 - 並未實際在檔案系統中建立檔案
 - 也未讀寫或修改該檔案內容
 - 該檔案可以已經存在或事後才建立

File類別建構子

- 建構子：
 - **public File(String pathname)**
 - `File myDir = new File("C:\\myDir");`
 - `File myFile = new File("C:\\myDir\\myFile.txt");`
 - **public File(String parent, String child)**
 - `File myFile = new File("C:\\myDir", "myFile.txt");`
 - **public File(File parent, String child)**
 - `File myDir = new File("C:\\myDir");`
 - `File myFile = new File(myDir, "myFile.txt");`

以上檔案路徑內的 【\\】 也可以使用 【/】 取代

File類別常用方法

範例：[TestDir.java](#)

方法	說明
boolean exists()	如果是檔案或目錄存在就回傳true，沒有就false
boolean isFile()	如果是檔案就回傳true，不是就false
boolean isDirectory()	如果是目錄就回傳true，不是就false
String[] list()	包含File所描述的所有檔案與目錄的名稱，回傳字串陣列
File[] listFiles()	包含File所描述的所有檔案與目錄的File物件，回傳File物件陣列
String getAbsolutePath()	回傳檔案或目錄的絕對路徑
String getPath()	回傳建構File物件時提供的路徑
String getParent()	回傳包含File的目錄名稱
String getName()	回傳檔案或目錄的名稱，所得名稱是路徑最後一個名稱
long length()	回傳檔案的長度

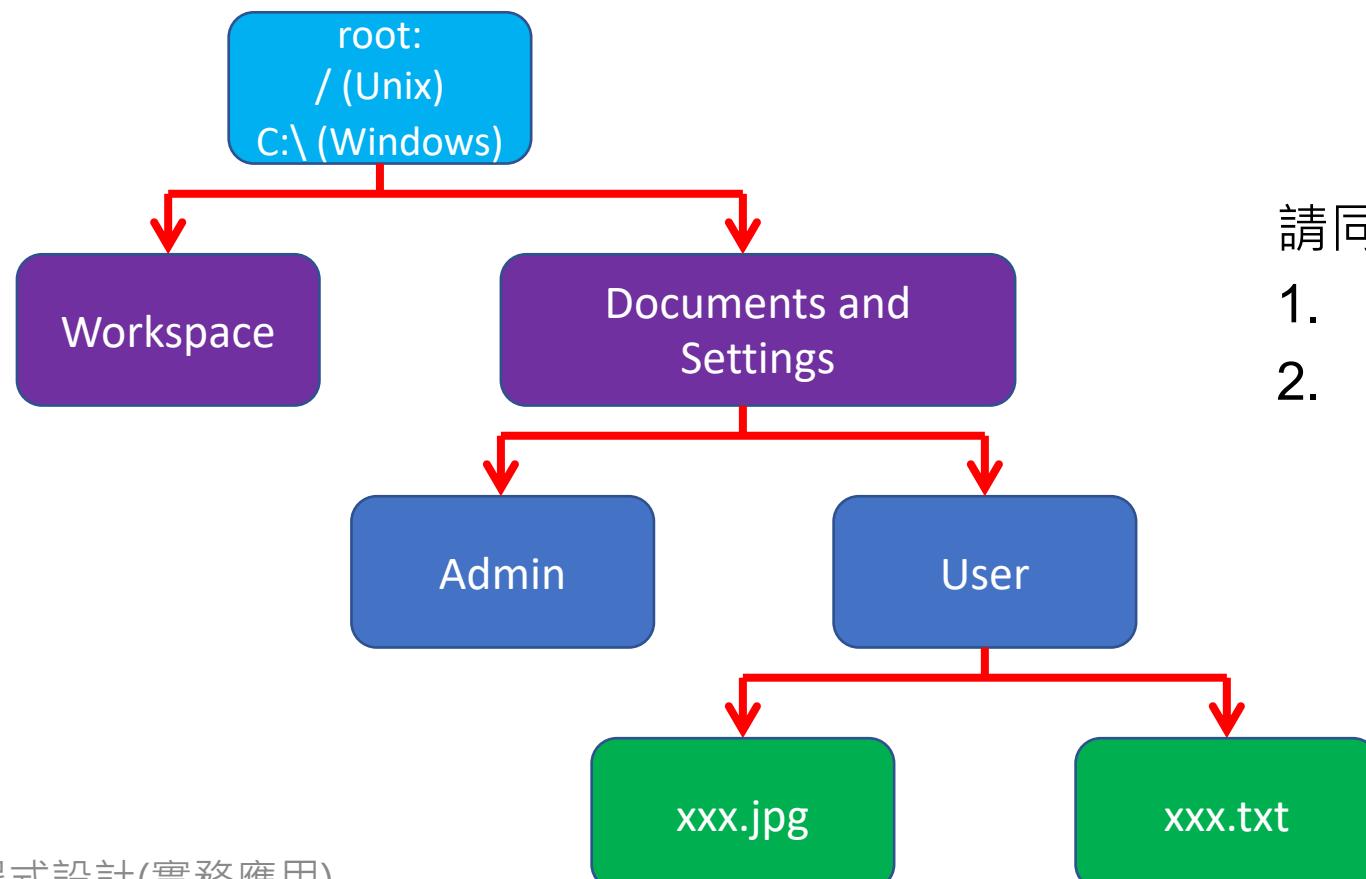
File類別常用方法

範例：[TestFile.java](#)

方法	說明
boolean canRead()	判斷檔案是否可被讀取
boolean canWrite()	判斷檔案是否可以覆寫(修改)
boolean setReadOnly()	設定檔案為唯讀
long lastModified()	傳回檔案最後一次的修改時間
boolean delete()	刪除檔案或目錄
boolean renameTo(File dest)	更改名稱兼移動到指定的路徑
boolean mkdir()	建立一個新資料夾 <pre>File file = new File("C:\\dir"); file.mkdir();</pre>
boolean mkdirs()	建立多層資料夾 <pre>File file = new File("C:\\dir2\\dir1"); file.mkdirs();</pre>
boolean createNewFile() throws IOException	建立新檔案

路徑示意圖

- “.”代表當前路徑，“..”代表上一層路徑，可用絕對路徑或相對路徑表示一個檔案或資料夾的位置
(註) 在eclipse開發環境裡，當前資料夾是指現在所屬的專案資料夾



請同學思考如何表示以下相對路徑：
1. 當前位置在Admin，要表示xxx.txt
2. 當前位置在User，要表示xxx.jpg

模組7

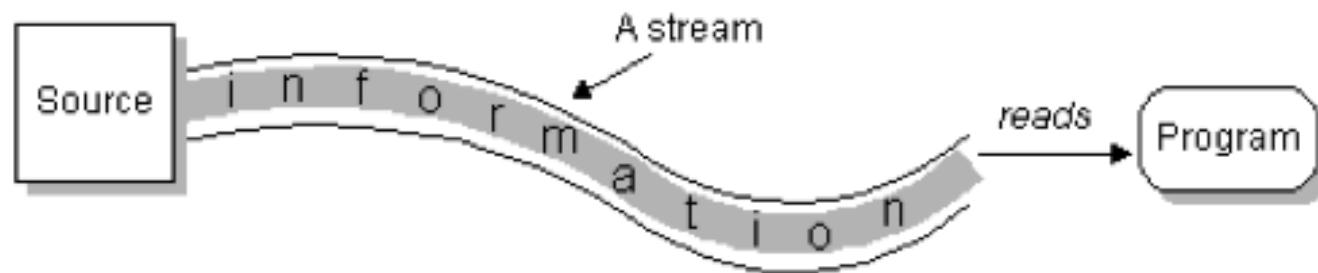
循序存取媒體

I/O (1)

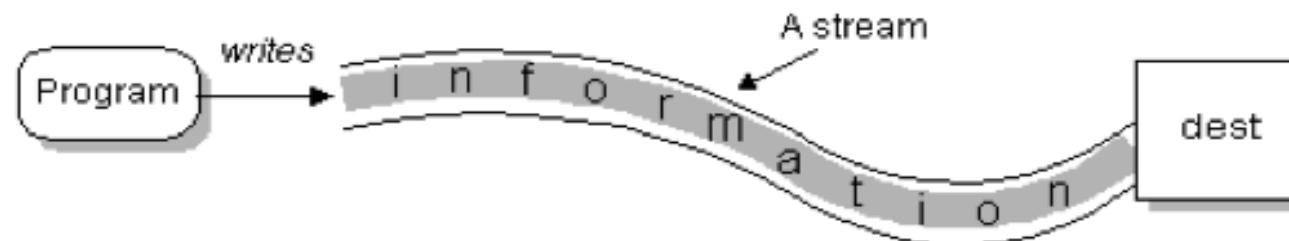
- 7-1：
理解輸入與輸出
- 7-2：
資料流API架構
- 7-3：
操作資料流 – 以檔案複製為例

資料流處理觀念

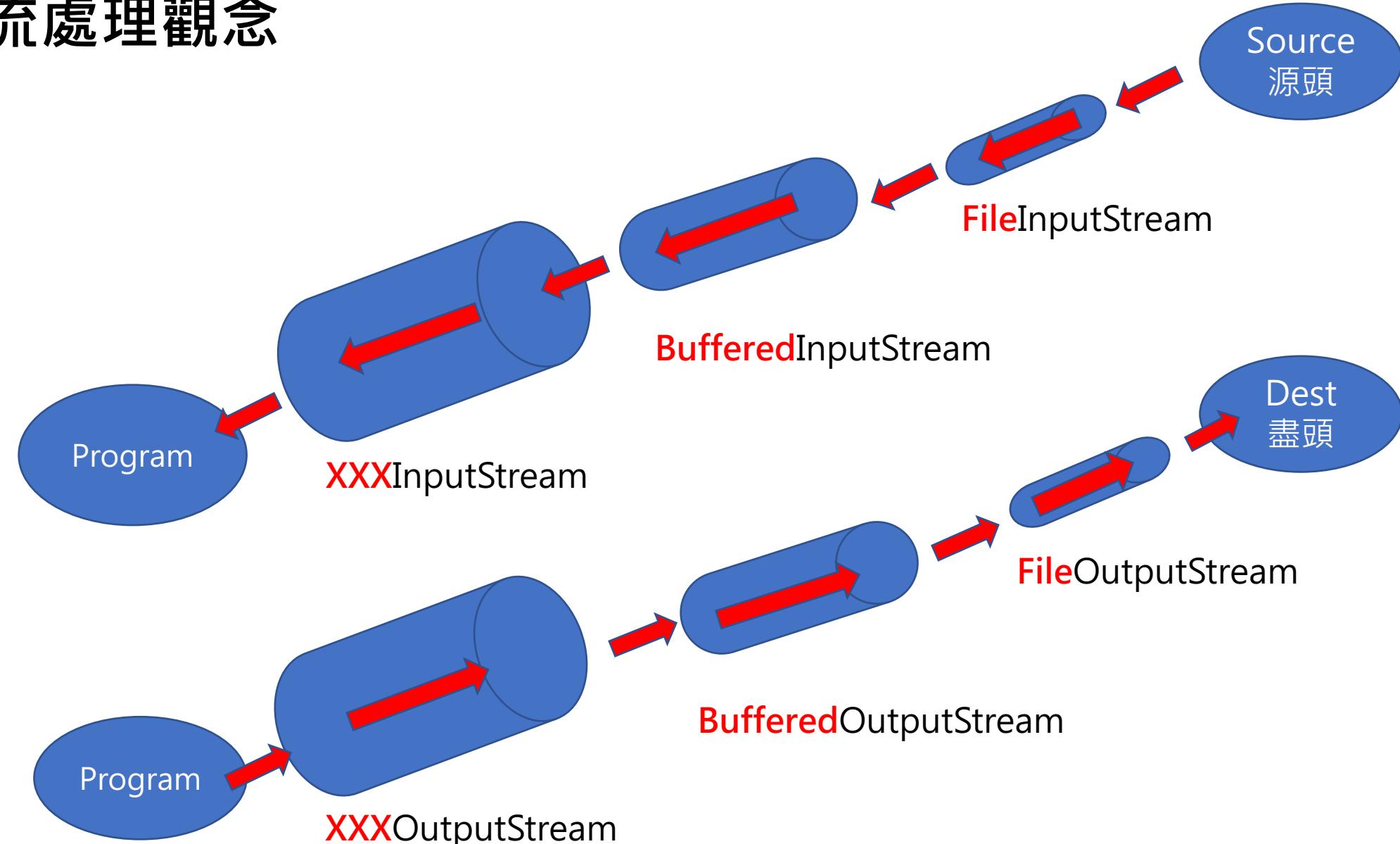
- 程式透過資料流(Stream)讀取一連串資料，來源可以是檔案、記憶體或是網路



- 程式也能透過資料流(Stream)將資料寫出到目的地，同樣可以是檔案、記憶體或是網路



資料流處理觀念



資料流API

- InputStream / OutputStream 及 Reader / Writer
 - Java的資料流類別內建於**四個抽象父類別**
- InputStream / OutputStream 型資料流
 - 存取是以8bits為基礎的byte，處理中文有困難
- Reader / Writer 型資料流
 - 存取是以16bits為基礎的char來處理Unicode
- I/O Stream 與 Reader / Writer 之間的資料傳輸
 - 網路 I/O 與 Console I/O是以byte為基礎的I/O

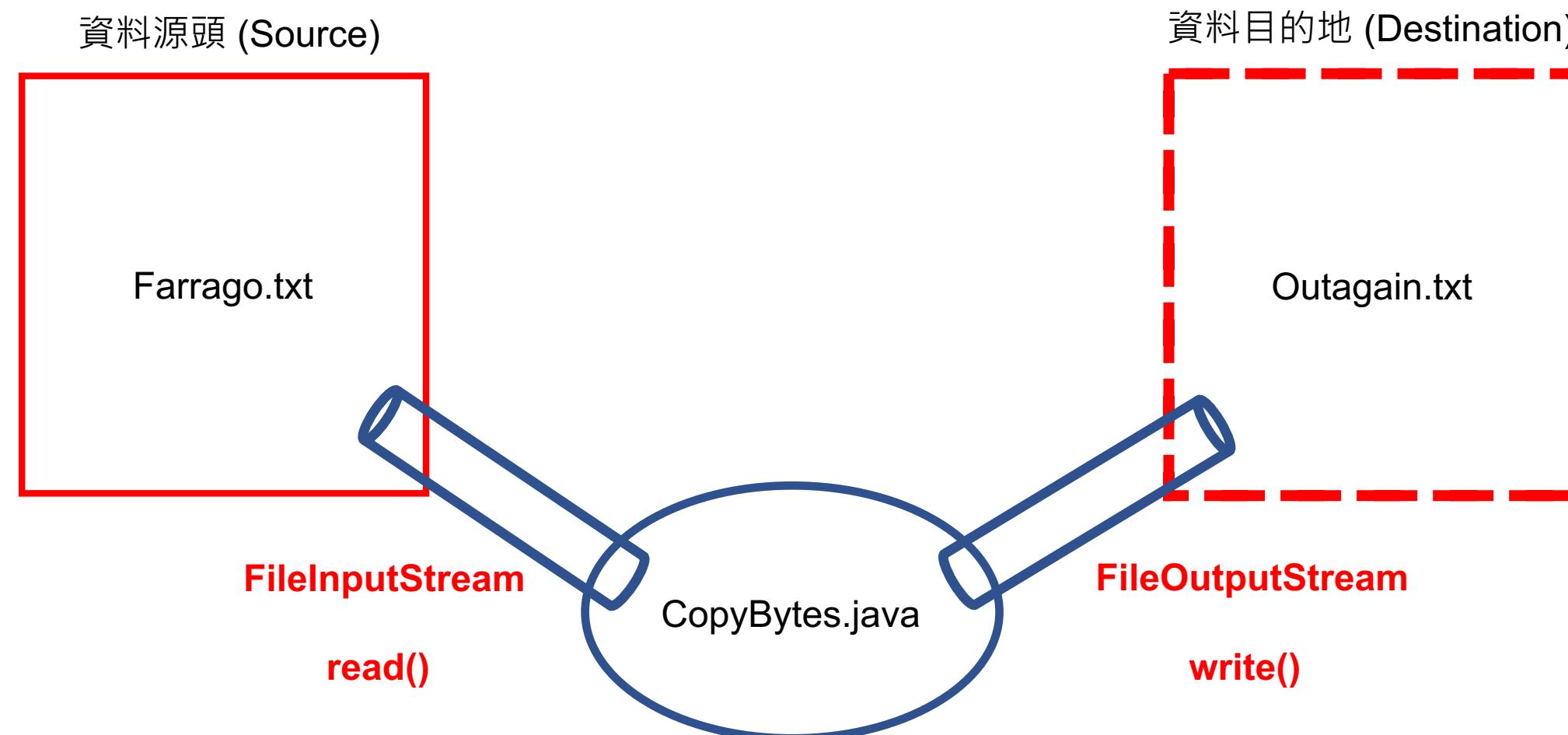
輸入父類別

- **InputStream**
 - int read()
 - 回傳值為檔案裡下一個byte資料，如回傳 -1 代表已到檔案末端
 - int read(byte[] buf)
 - 讀取檔案中下一段(buf.length個byte)資料，並放入陣列buf裡
 - 回傳值為實際讀取到的byte數量，如回傳 -1 代表已到檔案末端
 - int read(byte[] buf, int offset, int length)
 - 讀取檔案中下一段(buf.length個byte)資料，並放入陣列buf裡，從offset開始的位置
 - 回傳值為實際讀取到的byte數量，如回傳 -1 代表已到檔案末端
- **Reader** (方法說明同InputStream，byte資料改為**char字元**資料)
 - int read()
 - int read(char[] cbuf)
 - int read(char[] cbuf, int offset, int length)

輸出父類別

- OutputStream
 - void write(int b)
 - 將b的位元組(byte)資料寫至目的地
 - void write(byte[] buf)
 - 將陣列buf裡所有的位元組(byte)資料寫至目的地
 - void write(byte[] buf, int offset, int length)
 - 將陣列buf中從offset位置開始的length個位元組(byte)資料寫至目的地
- Writer (方法說明同OutputStream，byte資料改為**char字元**資料)
 - void write(int c)
 - void write(char[] cbuf)
 - void write(char[] cbuf, int offset, int length)

CopyBytes.java / Copy.java



模組8

循序存取媒體

I/O (2)

8-1：
I/O鍊

8-2：
Input/Output Stream

8-3：
緩衝(Buffer)功能

I/O Chain

- 為使用方便，Java依功能將I/O再分為：
 - 低階I/O：即**節點資料流(Node Stream)**
 - 高階I/O：即**處理(加工)資料流(Processing Stream)**
- **低階I/O類別**
 - 負責與媒體資料作存取
 - 大部份只提供**read, write**的陽春功能資料存取
- **高階I/O類別**
 - 主要作提供一些特殊的功能，如置入緩衝區、資料型態轉換及字元編碼等加工處理
 - 提供更多的讀寫方法(method)，如讀寫**int, double**或**String**等加工處理

I/O Chain注意事項

- 重要原則
 - **I/O鍊 (I/O Chain) :**
 - 在建立一個I/O前必須先用低階I/O類別來存取資料(如檔案)
 - 之後再使用高階I/O類別來控制低階I/O類別的動作
 - 此層層架構稱之為I/O鍊 (I/O Chain)
 - 高階I/O類別可再與其它高階I/O類別連結
 - 輸入類的資料流只能與輸入類的類別相連接
 - 輸出類的資料流只能與輸出類的類別相連接

InputStream Chain

範例：[InputStreamChain.java](#)

- 程式片段 (輸入)

1. 建立一個檔案輸入流

```
FileInputStream fis = new FileInputStream("輸入檔案名");
```

2. 建立一個高階I/O物件BufferedInputStream bis, 並連結至fis，將fis放到緩衝區

```
BufferedInputStream bis = new BufferedInputStream(fis);
```

3. 從緩衝區讀取資料，以減少CPU的I/O時間

```
bis.read();
```

- 程式架構圖



OutputStream Chain

範例：[OutputStreamChain.java](#)

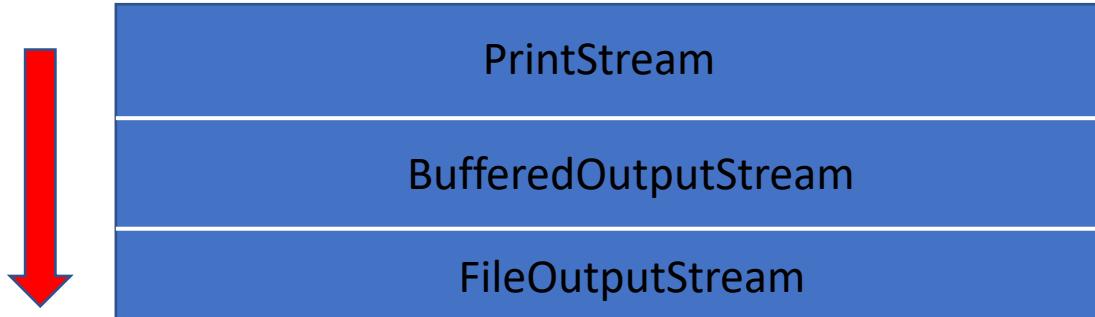
- 程式片段 (輸出)

```
FileOutputStream fos = new FileOutputStream("輸出檔案名");
BufferedOutputStream bos = new BufferedOutputStream(fos);
PrintStream ps = new PrintStream(bos);
```

```
ps.println("Hello World");
```

```
ps.close();
bos.close();
fos.close();
```

- 程式架構圖



緩衝(Buffer)區

- 想想看，假設你(妳)是一位便利商店的物流司機，今天車上的10箱飲料要用什麼方式送進門市裡面會是最省時省力的呢？



模組9

循序存取媒體

I/O (3)

- 9-1 : Reader / Writer
- 9-2 : 文字與位元資料流轉換
- 9-3 : 文字處理與編碼關係

Reader / Writer資料流

- Reader / Writer用法與 I/O Stream類似
 - Reader類別只能與Reader類別相連接
 - Writer類別只能與Writer類別相連接
 - 高階I/O類別亦可多層疊堆，使用高階I/O類別前，必須有一個低階I/O類別先處理媒體相關的存取動作
- 與I/O Stream不同則在於Reader / Writer是專門用於Unicode的字元處理

Reader Chain

範例：[ReaderChain.java](#)

- 程式片段 (輸入)

1. 建立一個檔案輸入流

```
FileReader fr = new FileReader("輸入檔案名");
```

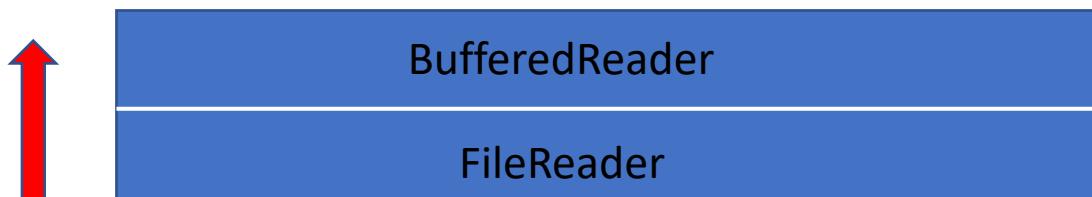
2. 建立一個高階I/O物件BufferedReader br, 並連結至fr，將fr放到緩衝區

```
BufferedReader br = new BufferedReader(fr);
```

3. 從緩衝區讀取資料，以減少CPU的I/O時間

```
br.readLine();
```

- 程式架構圖



Writer Chain

範例：[WriterChain.java](#)

- 程式片段 (輸出)

```
FileWriter fw = new FileWriter("輸出檔案名");
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter pw = new PrintWriter(bw);
```

```
pw.println("Hello World");
```

```
pw.close();
bw.close();
fw.close();
```

- 程式架構圖



InputStreamReader與OutputStreamWriter類別

- 網路I/O與Console I/O
 - 因為網路I/O與Console I/O是位元資料流，所以Reader / Writer不能夠直接存取網路I/O與Console I/O
 - 需要進行資料流的轉換，才能順利處理對應的資料內容
- InputStreamReader類別
 - An InputStreamReader is a bridge from byte streams to character streams
- OutputStreamWriter類別
 - An OutputStreamWriter is a bridge from character streams to byte streams

文字編碼與處理

- Unicode與UTF關係(補充)：
 - Unicode又叫統一碼、萬國碼、單一碼、標準萬國碼，產生的目的就是為了解決世界各國文字問題，像中文字在不同的地區(中國、香港、台灣)的寫法不大一樣，在編碼上當時就被歸成不同套，所以產生了不同的內碼轉換表，如GB2312、GBK、BIG5等。
 - Java對於文字都是採用Unicode編碼，Unicode對於任何字元都是使用2 bytes儲存 但這對英文語系國家只會覺得每個字元佔用2個位元組太浪費了... 因為英語、數字、符號這些在 ASCII 只使用了1 byte，所以本來用2 bytes的Unicode，基於節省儲存空間為目的，因此發展了Unicode的內部轉換格式，稱為 (Unicode Transformation Format，簡稱為UTF)
 - 如果一個僅包含基本7位ASCII文字的Unicode文件，每個字元都使用2 bytes的原Unicode編碼傳輸，其第一位元組的8位始終為0。這就造成了比較大的浪費。對於這種情況，可以使用UTF-8來進行演算，將字元轉換成一種可長可短的編碼，這樣可能節省大量的容量。對於網際網路傳輸資料尤其節省頻寬，所以成了電子郵件、網路檔案傳送最愛的一種編碼格式。

文字編碼與處理

- Unicode to UTF-8 (補充)：
 - ‘我’這個字元在Java裡面轉成int得到的整數為25105(十進位表示) 轉成Unicode的十六進位表示為 \u6211，它是如何對應到UTF-8的 E6 88 91？
 - 1. 利用小算盤將 6211 轉成二進位表示：110001000010001，但由於只有15個0與1，所以需要在最左邊補一個0。(Unicode編碼只要未滿16bits都是補0)
 - 2. 依照java API中，java.io.DataInput介面的說明將 0110001000010001 拆成三組位元
0110 001000 010001
 - 3. 010001 的前面補 10 成為 10010001，001000 的前面補 10 成為 10001000，0110 的前面補 1110 成為 11100110 就會得到下面三個位元組：11100110 10001000 10010001
 - 4. 以上三個位元組對應的十六進位表示就是：E6 88 91

模組10 循序存取媒體 I/O (4)

10-1：
網路資料流 – 簡易爬蟲

10-2：
java.net套件

10-3：
自訂緩衝與分段讀寫

爬蟲是什麼

- 又可稱為Web Crawler或Spider，主要就是藉由程式執行而自動瀏覽網路資料內容或是更進一步取得資料回來，最早是用來建立網路資料索引，以方便搜尋引擎執行的效能優化，現今多應用於資料爬取。
- 爬取資料行為是個灰色地帶，因為藉由程式偽裝成使用者(人)而取得該網站或web server所提供的資料。所以在商業用途上需多加確認與告知
- HTML 標籤結構觀念要清楚！
- 因為Java語言的特性，所以單純地使用Java API進行爬蟲功能實現，遇到動態網頁或是Javascript渲染過的內容，在爬取的支援性較差，需要再搭配第三方套件或工具實現較為方便

java.net

- **java.net**套件已被包在Java標準API裡，可以直接使用，裡面有許多對於網路程式設計所需要的介面、類別與相關方法，如URL, HttpURLConnection, Socket等。
- URL類別可以讓我們在Java程式建立一個對應指定的網路URL資源的物件
- HttpURLConnection類別可以讓我們藉由http通訊協定，對指定的網路URL資源進行存取設定與資料的輸入與輸出(結合資料流相關類別完成)
- 參考補充範例：**GetNatalieFromInternet.java** (內含自訂緩衝/分段讀寫操作)

模組11

循序存取媒體 I/O (5)

- 11-1：
Console I/O
- 11-2：
鍵盤輸入 - Scanner
- 11-3：
資料流特殊功能補充

文字主控介面I/O

- Console I/O
 - **System.in**、**System.out**、**System.err**三個不同資料流，不需 new 宣告即可使用
 - System.in
 - 標準輸入資料流，預設是**鍵盤**
 - 為**InputStream**的物件
 - System.out
 - 標準輸出資料流，預設是**螢幕**
 - 為**PrintStream**的物件
 - System.err
 - 標準錯誤輸出資料流，預設是**螢幕**
 - 為**PrintStream**的物件

Scanner類別(JDK 5)

範例：
[InputFromKeyboard.java](#)

- 鍵盤輸入範例

- 如JDK 5之前的版本要從鍵盤讀入整數值的作法

```
InputStreamReader isr = new InputStreamReader(System.in);
```

```
BufferedReader br = new BufferedReader(isr);
```

```
String s = br.readLine();
```

```
int n = Integer.parseInt(s);
```

- JDK 5以後的版本可改寫成？

```
Scanner sc = new Scanner(System.in);
```

```
String s = sc.next();
```

```
【int n = sc.nextInt();】
```

資料流功能補充

- 抽象父類別 `InputStream` 類別提供 `int available()`方法，以取得輸入資料流的資料大小(`number of bytes`)
- 輸出父類別`OutputStream`與`Writer`提供 `void flush()`方法，可以強制將緩衝區裡未滿的資料進行輸出，避免造成資料遺失的情況
- 建構子`FileOutputStream(String name, boolean append)`與建構子`FileWriter(String fileName, boolean append)`
 - 當`append = false`時，新增的資料將**覆蓋/取代**原始資料(預設為`false`)
 - 當`append = true`時，新增的資料將**附加**於原始資料之後

模組12

物件輸入與輸出

- 12-1：
物件輸入/輸出目的
- 12-2：
序列化(Serializable)
- 12-3：
序列化與分散式運算

物件輸出與輸入

範例：[ObjectInOut.java](#)

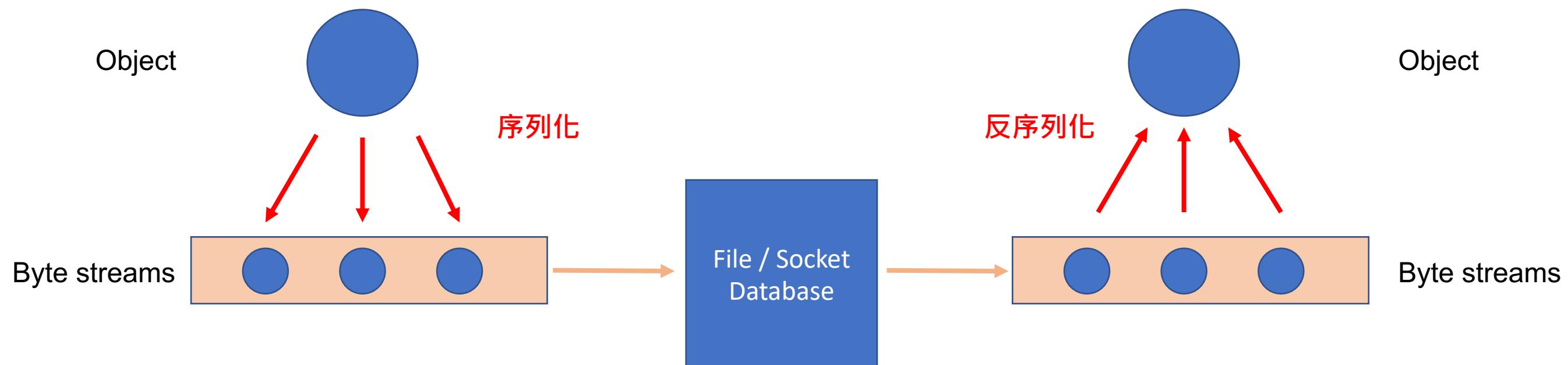
- 將物件寫出與讀入
 - 寫出物件：使用**ObjectOutputStream**類別寫出物件
 - 讀入物件：使用**ObjectInputStream**類別讀入物件
- 將物件寫出與讀入的建構子：
 - 寫出：**ObjectOutputStream**(*OutputStream out*) throws IOException
 - 讀入：**ObjectInputStream**(*InputStream in*) throws IOException
- 將物件寫出與讀入的方法：
 - 寫出：**void writeObject**(*Object obj*) throws IOException
 - 讀入：**Object readObject()** throws IOException, ClassNotFoundException
- 物件讀入注意事項：
 - 物件讀入順序必須與物件寫出順序相同
 - Object readObject()的回傳型態為Object，必須自行強迫轉型為原來寫出時的物件型態

序列化(Serializable)

- java.io.**Serializable**(空介面)：
 - 物件是動態產生的，欲將物件永久儲存時，稱做**persistence**
 - 欲將某物件的資料儲存(寫出)到OutputStream(檔案或socket)時，該物件必須實作Serializable空介面
 - 註1：Java類別預設是不實作Serializable介面的
 - 註2：父類別實作Serializable介面後，其子類別也等同於實作Serializable介面
 - 宣告為**transient**與**static**的資料成員不會被序列化
 - 如果某資料成員不想被**serialized**，程式設計者可以自行(主動)加上**transient**修飾子
 - 因為宣告為**static**的變數與方法，不是由任何此類別的物件單獨擁有，而是由屬於此類別所有物件共同擁有

序列化圖解(Serializable)

- 示意圖：



- 補充：`serialVersionUID`功能

I/O整理

- Java的資料流支援方式，功能強大又具自定的彈性
- 事實上，關於I/O方面的討論可以寫成一本書
- 一個無限制的鍊結機制讓你可混合一些類別來達成任何想要的I/O功能
- 可產生一些自己加強的資料類別，以便在一個輸入鍊或輸出鍊其中的任何地方做插入
- I/O奠定將來網路程式設計的基礎
- 物件永續性與序列化觀念奠定將來RMI (遠端方法呼叫 – Remote Method Invocation) 分散式運算基礎

模組13 認識集合

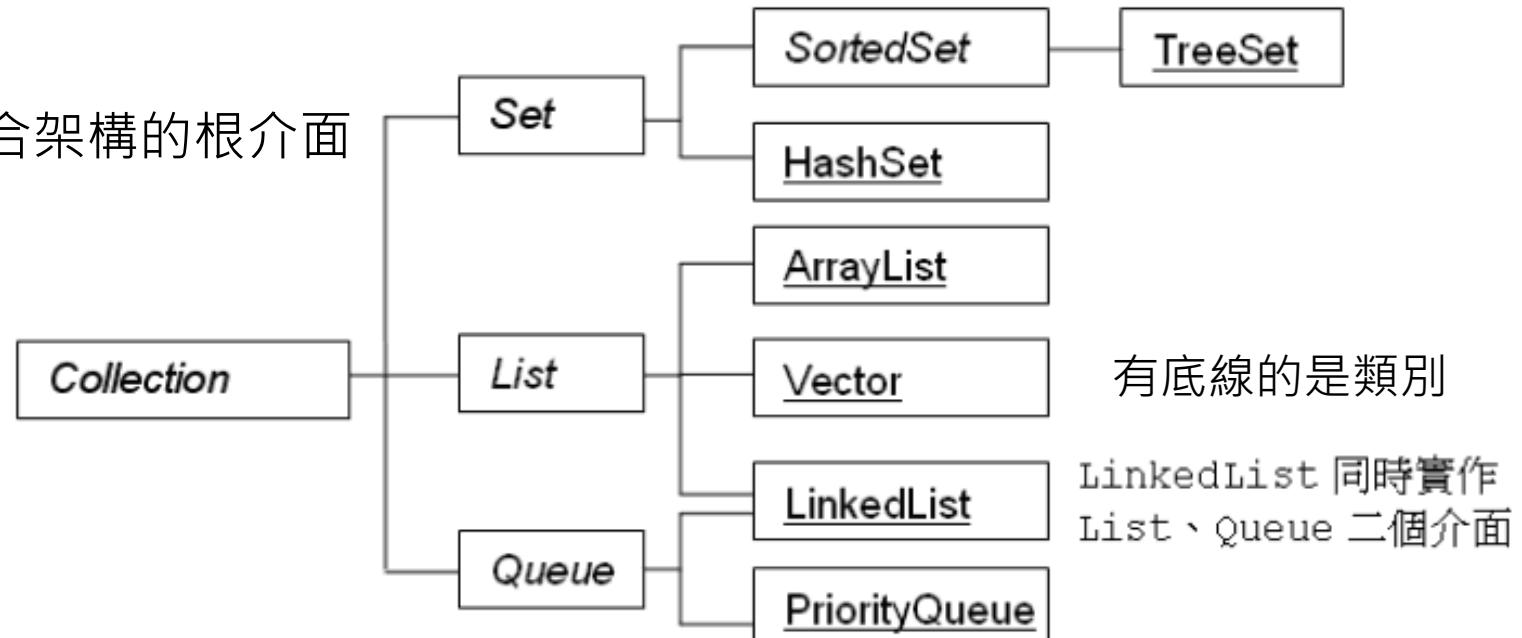
- 13-1：
區分集合與陣列
- 13-2：
集合API架構介紹
- 13-3：
List / Set集合特性介紹

集合與陣列比較

	集合	陣列
空間大小	建立時不必指定集合大小，無論將元素存入或移除，集合都會動態調整空間以符合需求；這是集合最大優點	建立時就必須指定陣列大小，而且之後無法改變。元素存取時，不可超過索引上限，否則會產生 ArrayIndexOutOfBoundsException
資料類型	可存放任何物件，但若存放基本類型，會先AutoBoxing成對應的物件後再存放。放入的物件還會自動轉型成Object類型，但若搭配泛型功能就可像陣列般限制元素的資料類型	可存放物件或基本類型，但必須符合陣列所宣告的資料類型
資料存取	要將元素取出，可使用 1. for-each 2. Iterator	要將元素取出，可使用 1. for迴圈搭配索引 2. for-each

集合架構介紹

- 認識集合物件
 - 集合物件是指一群相關聯的資料，集合在一起組成一個物件，存在裡面的資料，稱之為元素(element)
- 從JDK1.2開始，Java的集合物件以Collection介面與Map介面分作兩大類：
 - **Collection介面**：可持有各自獨立的物件
 - **Map介面**：持有成對的key-value物件
- java.util.Collection介面是存放單一物件集合架構的根介面



Collection相關物件常用方法

<<interface>> Collection	
boolean add (E obj)	指定物件新增至集合內
void clear()	將集合內所有元素清空
boolean isEmpty()	檢查集合是否為空容器
Iterator iterator()	取得 iterator 物件走訪集合內所有元素
boolean remove(Object obj)	將指定元素從集合中移除
int size()	回傳集合內的元素總數
boolean contains (Object obj)	檢查集合內是否有指定元素
Object[] toArray()	將集合內元素轉存為陣列

List / Set集合

- Collection介面最重要的兩個子介面：
 - 子介面**Set**：
 - 無特定順序，不允許重複(重複時不加入)
 - **HashSet**類別實作Set介面
 - 子介面**List**：
 - 有特定順序，允許重複
 - **ArrayList**類別實作List介面
- 子介面Queue參考模組20

註：Set沒有get方法可用

<<interface>>	
List	
void add(int index, E element)	指定物件新增至集合指定索引內
boolean add(E element)	指定物件新增至集合內
E get(int index)	取得指定索引的元素
E set(int index, E element)	指定物件取代集合指定索引的元素

模組14 使用集合物件 (1)

- 14-1：
使用ArrayList存取元素
- 14-2：
使用迭代器
- 14-3：
使用HashSet存取元素

使用ArrayList存取元素

範例：[TestArrayList.java](#)

```
3 public class TestArrayList {  
4     public static void main(String args[]) {  
5           
6         List list = new ArrayList();  
7         list.add(new Integer(12));  
8         list.add(new Long(34L));  
9         list.add(new Double(5.6));  
10        list.add("Hello");  
11        list.add("Hello"); // duplicate, is added  
12          
13        System.out.println("toString()=" + list);  
14        System.out.println("元素個數=" + list.size());  
15          
16        Iterator objs = list.iterator();  
17        while(objs.hasNext())  
18            System.out.println(objs.next());  
19          
20        // List家族可以用Iterator或for迴圈取值  
21        System.out.println();  
22        for (int i = 0 ; i < list.size() ; i++){  
23            Object obj = list.get(i);  
24            System.out.println(obj);  
25        }  
26    }  
27 }
```

迭代器Iterator

- 如何取得集合裡的元素
 - Collection介面提供 iterator()方法，回傳一個實作Iterator介面的迭代器物件
 - public Iterator iterator()
 - 利用此方法，可把collection裡的所有元素，轉換成可進行迭代的迭代器(Iterator)
 - 關於元素取得的順序是沒有任何保證的 (除非此collection是提供保證順序的特性)
- Iterator介面 (稱為迭代器介面，JDK1.2開始)
 - 此介面被用來擷取collection集合裡的所有元素(也包含了其子介面Set與List)
 - public boolean hasNext()
 - public Object next()
 - Iterator(迭代器)介面是為了取代JDK1.0的Enumeration(列舉)介面 (**參考集合附件**)

使用HashSet存取元素

範例：[TestHashSet.java](#)

```
3 public class TestHashSet {  
4     public static void main(String args[]) {  
5           
6             Set set = new HashSet();  
7             set.add(new Integer(12));  
8             set.add(new Long(34L));  
9             set.add(new Double(5.6));  
10            set.add("Hello");  
11            set.add("Hello");      // duplicate, not added  
12              
13            System.out.println("toString()=" + set);  
14            System.out.println("元素個數=" + set.size());  
15              
16            // Set家族只能用Iterator 取值  
17            Iterator objs = set.iterator();  
18            while(objs.hasNext())  
19                System.out.println(objs.next());  
20        }  
21    }
```

模組15 使用集合物件 (2)

- 15-1：
key – value結構
- 15-2：
以HashMap存取元素
- 15-3：
Map集合注意事項

Map介面

- **Map**是一種key/value的集合，每一筆資料皆有一對主鍵值(key)和內含值(value)

```
- HashMap map = new HashMap();  
map.put("key", "value");
```

Primary Key	Value
M001	David
M002	James
M003	Vincent

Map介面常用方法

<<interface>> Map	
Object put(Object key, Object value)	將指定的鍵與值放到Map裡
Object get(Object key)	依指定的鍵取得對應的值
Object remove(Object key)	依指定的鍵移除對應的鍵值組
void clear()	清空Map內所有的鍵值組
boolean containsKey(Object key)	檢查Map內是否有指定的鍵
boolean containsValue(Object value)	檢查Map內是否有指定的值
boolean isEmpty()	檢查Map是否為空容器
int size()	回傳Map內的鍵值組總數
Set keySet()	將Map內的所有鍵轉存成Set物件
Collection values()	將Map內的所有值轉存成Collection集合

使用HashMap

範例：[TestHashMap.java](#)

```
2  public class TestHashMap {
3      public static void main(String args[]) {
4          Map map = new HashMap();
5          map.put("one", new Integer(1));
6          map.put("two", "2");
7          map.put("three", new Float(3.0));
8
9          //取出所有的key,包裝為Set的型態
10         Set set = map.keySet();
11
12         Iterator it = set.iterator();
13         while(it.hasNext()){
14             Object myKey = it.next();
15             System.out.println(myKey +"="+ map.get(myKey));
16         }
17
18         /**
19          keySet() 方法來自Map介面,
20          所以所有Map家族成員都適用此方式取值 ,
21          因此Hashtable類別也當然適用
22         */
23
24     }
25 }
```

Map注意事項

- 若加入相同的主鍵值，則新的資料會取代舊的資料
- 使用主鍵值尋找內含值，若主鍵值不存在，則會回傳null
- **HashMap**類別實作Map介面
 - 取得Map集合中所有的key，可以使用keySet()，它會將Map裡所包含的所有key以Set介面的型態回傳
 - Set keySet()
 - 取得Map集合中所有的value，可以使用values()，它會將Map裡所包含的所有value以Collection介面的型態回傳
 - Collection values()

模組16 泛型機制

- 16-1：
泛型與集合關係
- 16-2：
使用與自訂泛型
- 16-3：
自訂泛型上下邊界

泛型(Generic Type)

- JDK1.5的泛型設計有以下優點
 - 可預先指定集合裡能存放的元素型別為何
 - 因此，只要不小心加入其它類別的物件，在編譯階段就會出現錯誤，使得原本放在執行階段才會檢查出來的問題，提昇到了編譯時期
 - 避免runtime時期的java.lang.ClassCastException的錯誤，也因型別已知，取出資料時可省略型別轉換(Cast)的麻煩
 - 另Java使用動態連結技術，並且有共同的Object祖先做為最根本的多型
- 在JDK1.4之前，實作Collection介面時，我們能夠處理Object物件，如add(Object o)或remove(Object o)方法
 - 如此雖可讓Collection介面變得一般化，但是因為在大部份的實際狀況下，我們放在Collection中的物件通常都屬同一個類別，取出時型別為Object還得作型別轉換(Cast)，較為麻煩也容易出錯

泛型使用比較

範例：
[BeforeGenericList.java](#)
[GenericList.java](#)

JDK 5以前作法

```
5  
6 public class BeforeGenericsList {  
7     public static void main(String[] args) {  
8         List data = new ArrayList();  
9         data.add("Hello");  
10        data.add("World");  
11  
12        Iterator it = data.iterator();  
13        while (it.hasNext()) {  
14            String str = (String)it.next(); //強制轉形  
15            System.out.println(str);  
16        }  
17    }  
18 }
```

JDK 5以後作法

```
6 public class GenericList {  
7     public static void main(String[] args) {  
8         List<String> data = new ArrayList<String>();  
9         data.add("Hello");  
10        data.add("World");  
11  
12        Iterator<String> it = data.iterator();  
13        while (it.hasNext()){  
14            String str = it.next(); //強制轉形,不再需要  
15            System.out.println(str);  
16        }  
17    }  
18 }
```

Set / Map與泛型

範例：GenericMap.java

- 較複雜的泛型，除了在List部份，Set和Map系列當然也支援泛型

```
7 public class GenericMap {  
8  
9 public static void main(String[] args) {  
10    Map<Integer , String> map = new HashMap<Integer , String>();  
11    for (int i = 0; i < 3; i++) {  
12        map.put(new Integer(i) , "number" + i);  
13    }  
14    System.out.println(map.get(new Integer(0)));  
15    System.out.println(map.get(new Integer(1)));  
16    System.out.println(map.get(new Integer(2)));  
17 }  
18 }
```

自訂泛型

範例：MyGeneric.java

- 程式設計師也允許可以對自己創建的類別加上泛型機制

```
7 public class MyGeneric {  
8  
9    public static void main(String[] args) {  
10        MyGenericType<String> myGeneric = new MyGenericType<String>();  
11        for (int i = 0; i < 3; i++) {  
12            myGeneric.add("number" + i);  
13            System.out.println(myGeneric.get(i));  
14        }  
15    }  
16}  
17  
18 class MyGenericType<Type> {  
19  
20    private List<Type> list;  
21  
22    public MyGenericType() {  
23        list = new Vector<Type>();  
24    }  
25    public void add(Type t) {  
26        list.add(t);  
27    }  
28    public Type get(int i) {  
29        return list.get(i);  
30    }  
31}
```

泛型上下邊界

範例：[GenericAdv.java](#)

- 泛型的進階設定
 - 泛型的設定可使用「？」搭配「**extends**」或「**super**」來增加泛型的彈性
 - 如：
 - <? extends Number>：代表可以是Number或Number的子類別
 - <? super Number>：代表可以是Number或Number的父類別
- 註：？在泛型機制裡，代表為「any type」的意思

模組17 加強迭代操作

- 17-1：
Iterable介面
- 17-2：
for-each語法蜜糖
- 17-3：
for-each與傳統**for**迴圈

Iterable介面 (JDK 5)

- 從JDK1.5開始，Collection介面增加了新的**泛型**(Generic Type)功能設計，並繼承JDK1.5的新介面Iterable
- Iterable介面(JDK1.5)：
 - 此介面只有一個iterator()方法，回傳Iterable介面
 - Iterable<T> iterator()
- 實作Iterable介面，其目的是為了允許物件可以使用JDK1.5的「**增強型for迴圈(for-each)**」語法
 - Implementing this interface allows an object to be the target of the “foreach” statement”

註：當使用JDK1.5的**for-each**來走訪集合的元素內容時，一切將顯得格外輕鬆！

增強型for迴圈 (for-each)

- 增強型的for迴圈 (Enhanced for Loop) – for each：
 - 在JDK1.5中，針對for迴圈作了一些加強，讓我們無需知道陣列(array) 或集合(collection)的長度，甚至也不用迭代器(iterator)，便可以將其中的元素一一取出
 - **使用for – each來走訪集合的元素會格外輕鬆**
- 語法：
 - **for (資料型態 變數名稱 : 陣列或集合)**
 - **for (Type varName : listName)**
 - 即可將listName裡的元素依順序，由型別為Type的變數varName存取
 - 陣列或集合中元素的型別必須是可以轉型為Type的型別

範例：
[EnhancedForArray.java](#)
[EnhancedForCollection.java](#)
[EnhancedForMap.java](#)

兩種迴圈比較

範例：
[EnhancedForAttention.java](#)

- for-each迴圈之所以被稱為語法蜜糖(sugar)，就是因為對於先前迭代器的操作語法，for-each能達到更簡化的程式碼撰寫，即可走訪集合/陣列裡的所有元素
- 但也因為運作機制為迭代，因此元素取得操作的彈性不大，多用在元素全部取得的情境
- 傳統for迴圈具備迴圈設計要素(初值設定、條件判斷、計次)，除了對存取上更有操作的彈性之外，在執行效能上也會比for-each來得更好

模組18

集合進階操作 – 物件排序

18-1 :
TreeSet與TreeMap

18-2 :
使用Comparable介面

18-3 :
compareTo方法

TreeSet與TreeMap

範例：
`TestTreeSet.java`
`TestTreeMap.java`

- TreeSet實作了SortedSet介面，成為了一個擁有大小排序特性的Set集合
 - 同時也維持了元素不重複的特性
- TreeMap實作了SortedMap介面，成為了一個擁有大小排序特性的Map集合
 - 使用key的資料做為大小排序依據
- 使用有排序特性的集合需保證集合內的元素為同一類型(才有比大小的意義！)
 - 宣告泛型即可保證集合內的元素為同一類別

自訂物件大小排序

- 常用資料的大小排序 (見以下範例)：
 - [TestArraysForArray.java](#)
 - [TestCollectionsForList.java](#)
- 自訂物件若是想藉由集合或陣列擁有大小排序的特性，程式設計師需主動對該類別實作Comparable介面，**實作目的其實就是描述此物件的大小定義與規則**
- 一個有實作Comparable介面的物件實體才會是被Java認同可以進行排序操作的資料

compareTo方法說明

- Comparable介面裡需要實作的抽象方法：
 - int compareTo(T target)
- compareTo方法說明：
 - 回傳一個大於0或是小於0的整數 (通常用 1 跟 -1 做為代表) , 來定義物件的大小
 - 參數型別 T 為泛型機制，在宣告Comparable介面時決定傳入比較物件的型別
 - 在此方法裡決定使用該物件的何種屬性進行大小定義的設計
 - Arrays, Collections類別的sort方法內部即會呼叫該比較物件的compareTo方法

範例：
[Employee.java](#)
[TestArraysForArrayListEmp.java](#)
[TestCollectionsForListEmp.java](#)

compareTo圖解 (以TreeSet為例)

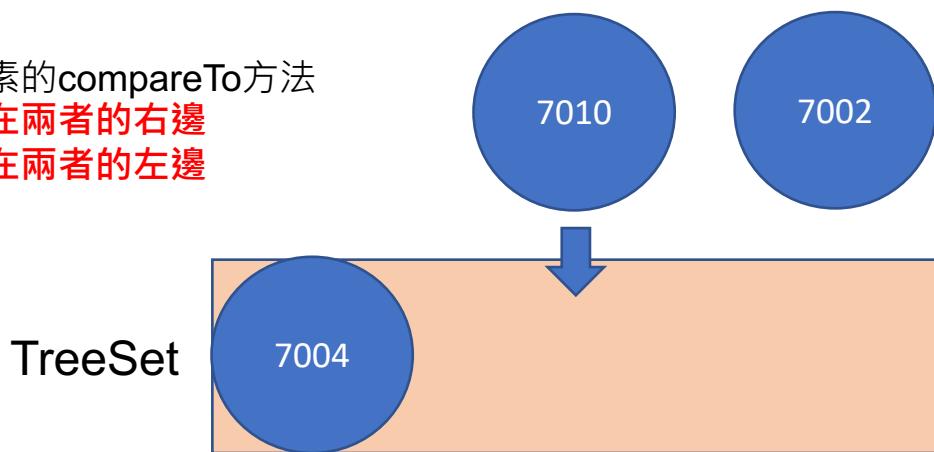
Step 1:

待加入員工



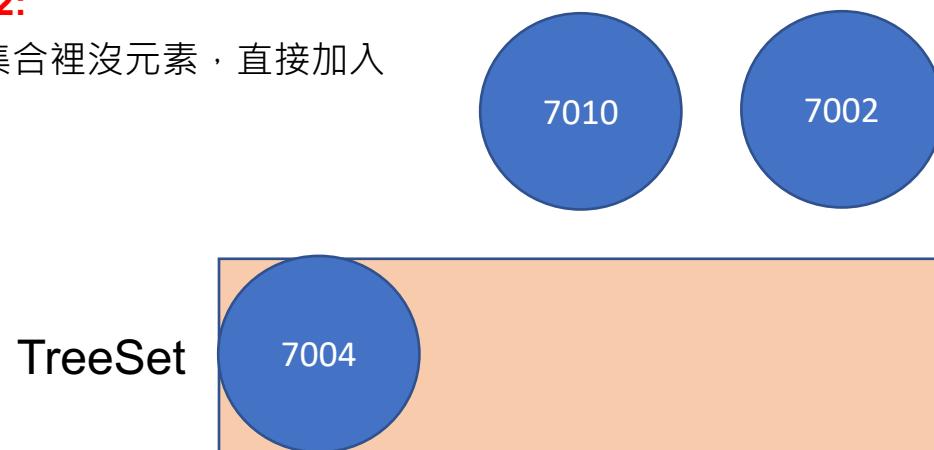
Step 3:

加入時會呼叫集合內元素的compareTo方法
大的回傳正值，就代表在兩者的右邊
小的回傳負值，就代表在兩者的左邊



Step 2:

因為集合裡沒元素，直接加入



Step 4:

再加進來的元素也跟集合內的元素進行比較
直到此元素比較的左邊或右邊已經沒有可比
較的元素為止



模組19

集合進階操作 – 物件唯一性

- 19-1 :
HashSet唯一性操作
- 19-2 :
覆寫hashCode方法
- 19-3 :
TreeSet唯一性操作

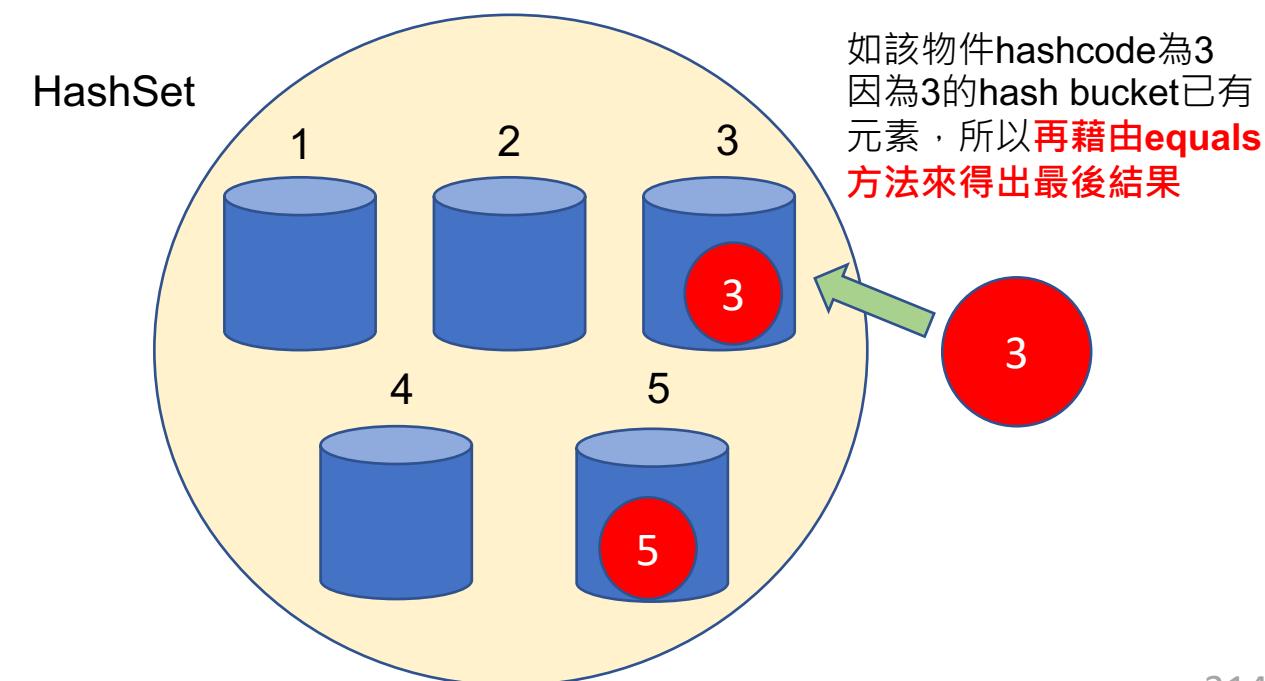
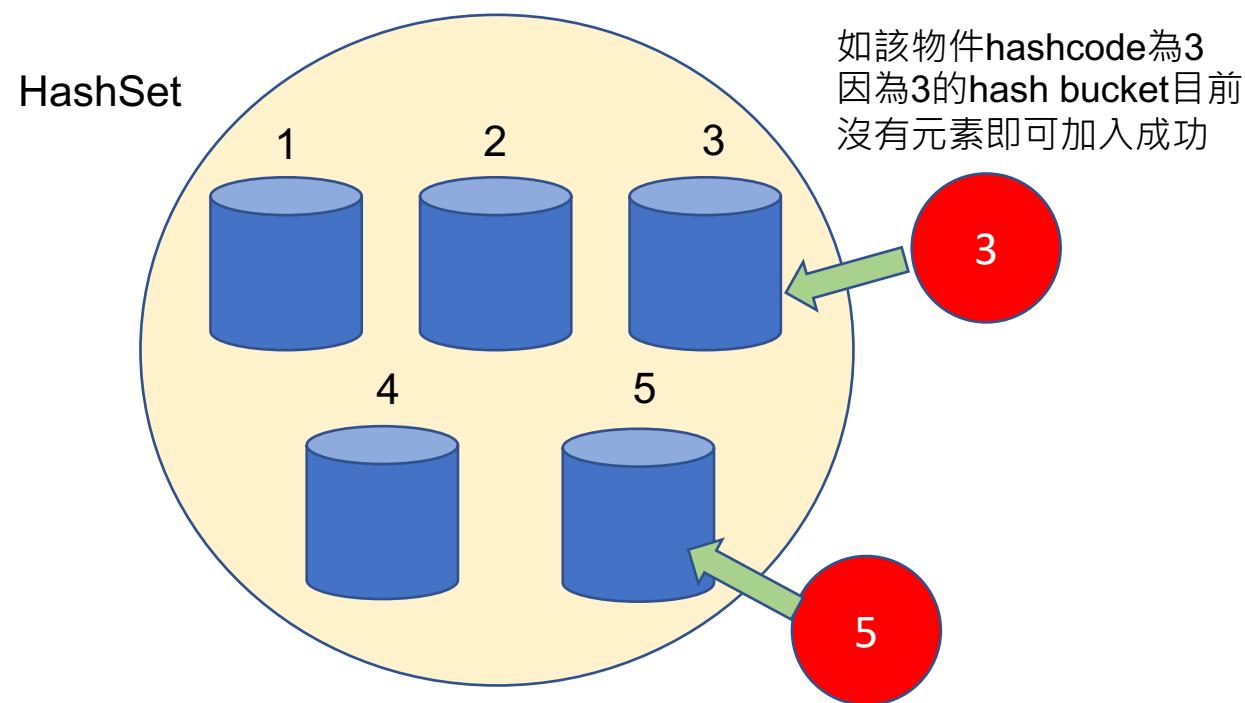
再看HashSet集合

範例：
`Employee.java`
`TestHashSetEmp.java`

- HashSet集合雖沒有排序特性，但保證了元素的唯一性(重複不加入)，在一些功能實現的情境上非常方便即可達成，但使用上需注意：
- **自訂物件若要搭配HashSet達到唯一性，程式設計師需自行主動覆寫從Object類別繼承來的equals與hashCode方法，覆寫目的也是為了定義“什麼叫做一樣的物件”**
- Java API文件中也明確告知，絕大部份情況下，覆寫equals方法最好連帶也一起覆寫hashCode方法
 - equals → 明確比對
 - hashCode → 模糊比對

什麼是hashCode

- 每個Java物件都擁有hashCode方法可以取得一個整數的資料(從Object類別繼承獲得)
- HashSet在加入元素前會先藉由hashCode來區分元素所在的空間(hash bucket)，若該空間裡沒有其它元素即加入，若該空間已有其他元素，再透過equals方法得出最後結果是否為重複



314

覆寫hashCode

- hashCode (雜湊)的演算牽涉到數學領域知識，因此這邊不會做太深入說明，有興趣的學員可以自行補充。基本覆寫的流程如下：
 - (1) 決定好要比較的物件屬性有哪些 (與覆寫equals方法使用的屬性相同)
 - (2) 選擇一個質數，用它與需要運算的屬性做累積相乘 (擴大運算結果避免容易發生碰撞情形)
 - (3) 回傳累積相乘的最後結果
- 結論深入淺出：
滑鼠右鍵 → Source → Generate hashCode() and equals()

有關TreeSet唯一性

範例：
`Employee.java`
`TestTreeSetEmp.java`

- TreeSet除了有元素大小排序的特性之外，也保有Set共同的特性，也就是元素重複不加入，但使用此集合時，必須注意：
 - TreeSet裡面的元素需為”可比較的”，也就是有實作Comparable介面
 - TreeSet不重複的根據是compareTo方法的結果，並不是equals()與hashCode()的結果
 - 在compareTo實作方法裡，對於”一樣大”的元素，需要做 return 0 的設計

模組20 集合與資料結構

20-1：
LinkedList與ArrayList

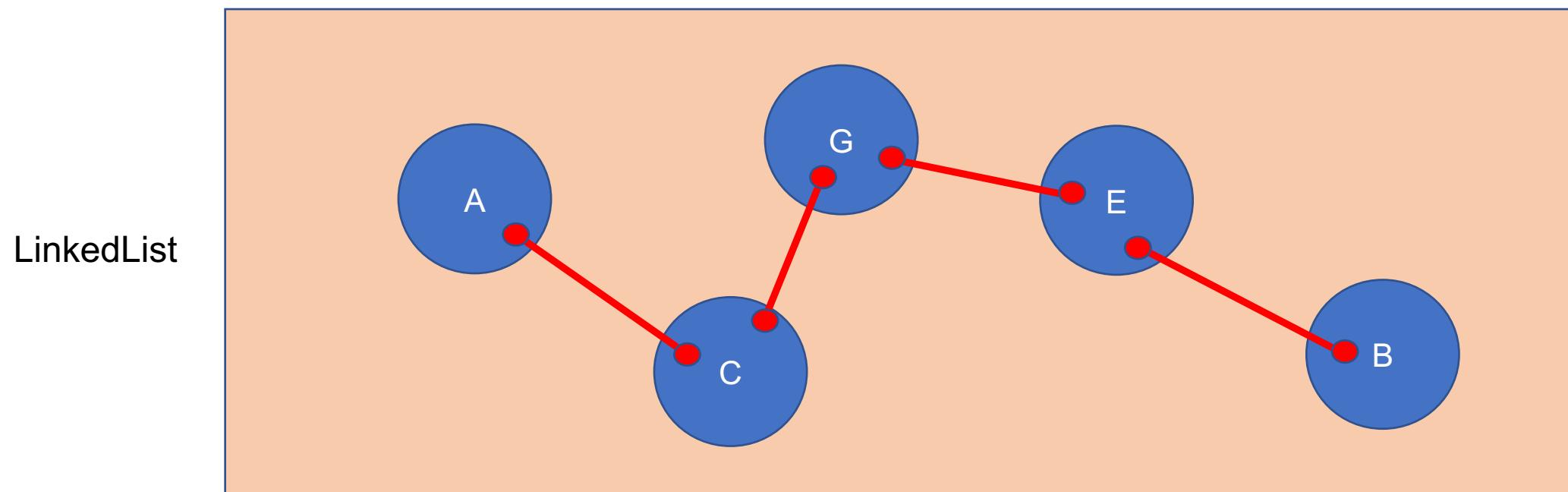
20-2：
佇列(Queue)結構操作

20-3：
FIFO與LIFO結構

Linked(鏈結)List

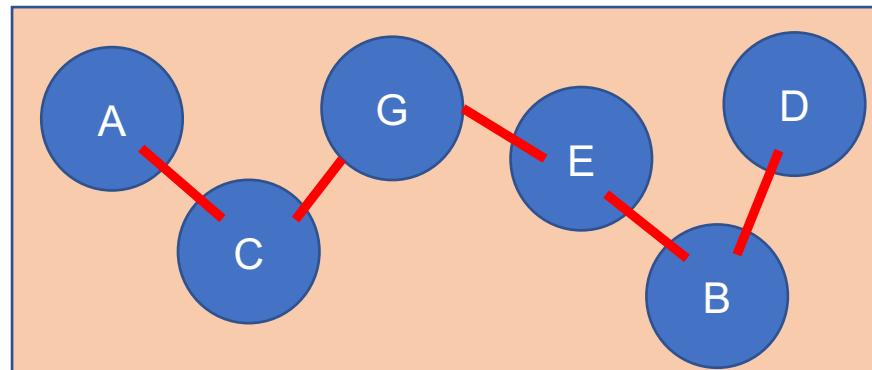
範例：[TestLinkedList.java](#)

- LinkedList實作了List介面，成為了一個有順序性，資料也可重複加入的集合
- 使用方法與ArrayList幾乎相同，但因為是鏈結的資料結構，因此非常適合對頭/尾元素進行處理

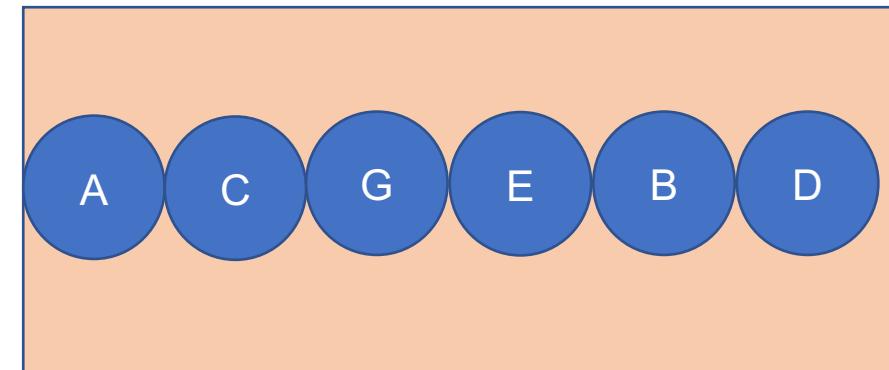


LinkedList與ArrayList比較

LinkedList



ArrayList



優勢：適合對元素常常插入或移除
劣勢：用索引值的存取表現較差

優勢：用索引值的存取表現佳
劣勢：不適合對裡面元素做插入或移除

- 註：LinkedList也同時實作了Queue介面，也可以拿來做為佇列結構使用

佇列(Queue)

範例：[TestQueue.java](#)

- 佇列結構即為日常生活上的隊伍，因此有順序性之外，對於佇列元素也總是只能對”第一個”元素做取出的動作，需注意的是，取出的同時也從此佇列移除了該元素。
- 非常適合使用佇列進行資料”消化”的操作，如：待處理事項



PriorityQueue與Comparator

範例：
`TestPriorityQueue1.java`
`TestPriorityQueue2.java`

- PriorityQueue實作Queue介面而擁有佇列特性之外，也保證了元素會按照大小順序取出，另外也可以提供實作的Comparator的物件，自訂元素大小的排序規則
- Comparator介面使用時機：
 - 想改變排序規則的資料是標準API的項目，如Integer, String...等
 - 想改變排序規則，但沒有原始碼可以調整
- 除了PriorityQueue之外，像是sort()，TreeSet等有大小排序功能的方法或是集合也都可以傳入Comparator實作物件進行排序規則的調整，大幅增加使用上的彈性

FIFO與LIFO

範例：[Test_LIFO_Stack.java](#)

- FIFO (First-In-First-Out) , 也就是口語說的”先進先出”，像剛介紹的佇列結構即是一個標準的先進先出的資料結構，注意這邊說的”出”，是指元素取得並包含移除的動作
- LIFO (Last-In-First-Out) , 也就是口語說的”後進先出”，指的就是堆疊(Stack) , Java集合API就有一個Stack類別可以讓我們實現此種資料結構上的操作
- 需注意的是，使用Stack操作LIFO，並不是用索引值取得元素，而是使用該物件提供的pop()取得

集合API整理

特性	集合類型	List	Set	Queue & Deque(雙向佇列)	Map
無特定順序性			HashSet		HashMap
大小順序	搭配sort方法		TreeSet	PriorityQueue	TreeMap (用Key排序)
加入順序	ArrayList Stack		LinkedHashSet	LinkedList ArrayDeque	LinkedHashMap
唯一性			參考模組19		Key不重複 參考模組19
執行緒安全	Vector				Hashtable ConcurrentHashMap

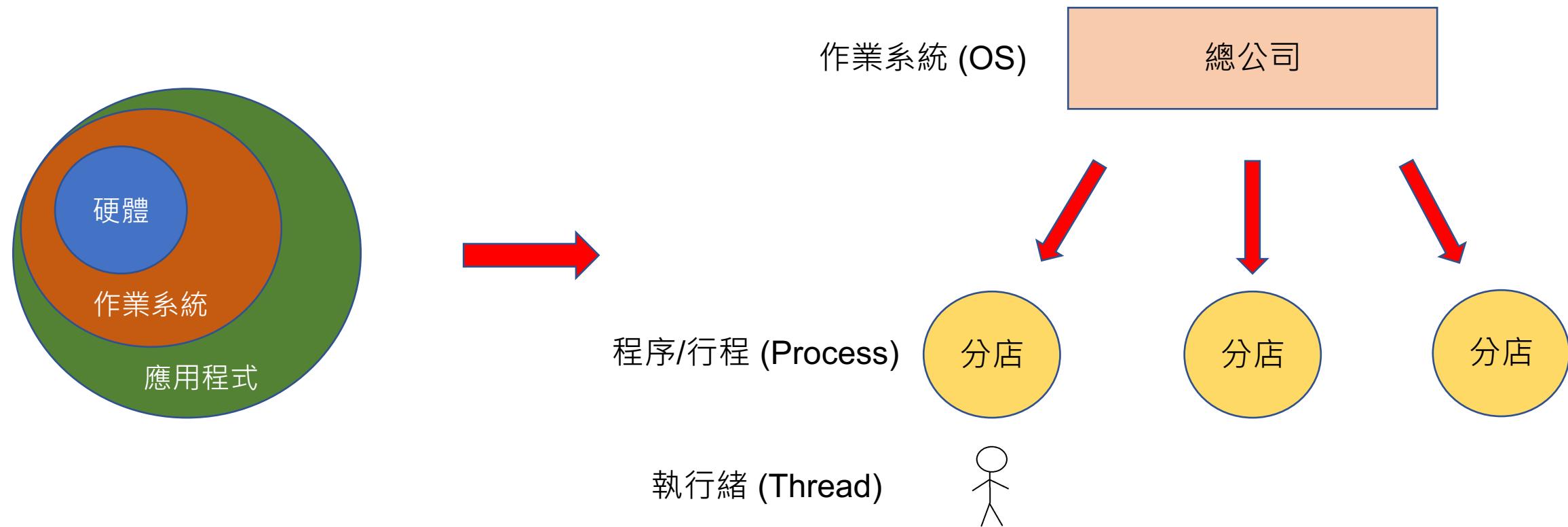
模組21

多執行緒設計 (1)

- 21-1：
區分程序與執行緒
- 21-2：
單執行緒與多執行緒
- 21-3：
使用Thread類別

程序(Process)與執行緒(Thread)

- Process與Thread關係以下圖解：



執行緒 (Thread)

- 什麼是執行緒(Thread)
 - 執行緒是程式中的執行區段，這個區段中的指令能夠不被其它區段影響而獨立執行，並可以在我們需要時啟動，不需要時關閉，以節省系統資源
- Multithreading(多執行緒)
 - Java支援多執行緒：
 - 看似電腦可同時執行許多工作，實質為CPU在各個程式中切換
 - 不是每個程式都必須使用執行緒，但使用多緒多工，可使系統的效率得以充分發揮
 - 已知的執行緒：
 - Java的Garbage Collector即是一個執行緒 (daemon thread)
 - Java程式啟動時，即自動建立一個執行緒，稱為主執行緒 (main thread)

執行緒與多工 (Multitasking)

- Multitasking(多工)指的是單一系統同時執行多個工作，由作業系統角度可以分成以下：
 - 合作型 (Cooperative) 多工
 1. 分享CPU是程式的事情 (容易造成獨佔)
 2. 需使用某些系統程序讓程式執行的控制權轉移
 - 強奪型 (Preemptive) 多工
 1. 由系統分配(排程)CPU的使用 (無法獨佔)
 2. 可由系統中斷工作，切換到另一個工作

執行緒與多工 (Multitasking)

- 由程式角度可以分成以下：
 - Process-based multitasking 多工
 1. 允許電腦同時執行兩個或更多個程式
 2. Program 是分配送遭的最小單位程式碼
 - Thread-based multitasking 多工
 1. 在此環境下，單一程式可以同時執行兩個以上的工作
 2. Thread 是可被分配送遭的最小單位程式碼

使用Thread類別

- 繼承java.lang.**Thread**類別
- 建立**Thread**類別的衍生類別，並覆寫(Override)其**run()**方法
- run()**方法為執行緒執行的地方

範例：
[CounterThread.java](#)
(與[CounterMain.java](#)做比較)

```
1 public class CounterThread extends Thread {  
2     int counter = 10;  
3     public void run() { //執行緒執行的地方  
4         while (counter >0) {  
5             System.out.println(counter);  
6             counter--;  
7  
8             try {  
9                 Thread.sleep(1000); //暫停一秒  
10            } catch (Exception e) {  
11                }  
12            }  
13        }  
14  
15    public static void main(String arg[]) {  
16        CounterThread t1 = new CounterThread(); //產生執行緒物件  
17        t1.start(); //呼叫執行緒物件的start()方法(即啟動執行緒)，隨即執行物件中的run方法  
18    }  
19 }
```

模組22 多執行緒設計 (2)

22-1：
使用Runnable介面
22-2：
執行緒生命週期
22-3：
Thread類別常用方法

使用Runnable介面

範例：
[CounterRunnable.java](#)
(與[CounterThread.java](#)做比較)

- 在不能多重繼承時，可實作java.lang.**Runnable**介面
- 建立實作**Runnable**介面的類別，並實作(implements)其**run()**方法，再將上述衍生類別的參考，透過**Thread**類別的建構子以建立一個**Thread**類別的實體

```
1 public class CounterRunnable implements Runnable {  
2     int counter = 10;  
3     public void run() { //執行緒執行的地方  
4         while (counter >0) {  
5             System.out.println(counter);  
6             counter--;  
7  
8             try {  
9                 Thread.sleep(1000); //暫停一秒  
10            } catch (Exception e) {  
11                }  
12            }  
13        }  
14  
15        public static void main(String arg[]) {  
16            CounterRunnable r1 = new CounterRunnable(); //產生Runnable物件  
17            Thread t1 = new Thread(r1); //再由Runnable物件，產生執行緒Thread物件  
18            t1.start(); //呼叫執行緒物件的start()方法(即啟動執行緒)，隨即執行物件中的run方法  
19        }  
20    }
```

執行緒生命週期 (Life cycle)

- **預備狀態(Ready)**

- 執行start()方法即進入排程器中等候CPU處理

- **執行狀態(Running)**

- run()方法被呼叫時

- **死亡狀態(Dead)**

- run()方法執行完畢時，或stop()方法被呼叫時 (註：stop()不再使用)

- **等待狀態(Waiting)**

- 執行wait()方法即移出執行狀態，透過notify()或notifyAll()方法回到預備狀態

- **睡眠狀態(Sleeping)**

- 停止一段時間後回到預備狀態

- **阻塞狀態(Blocked)**

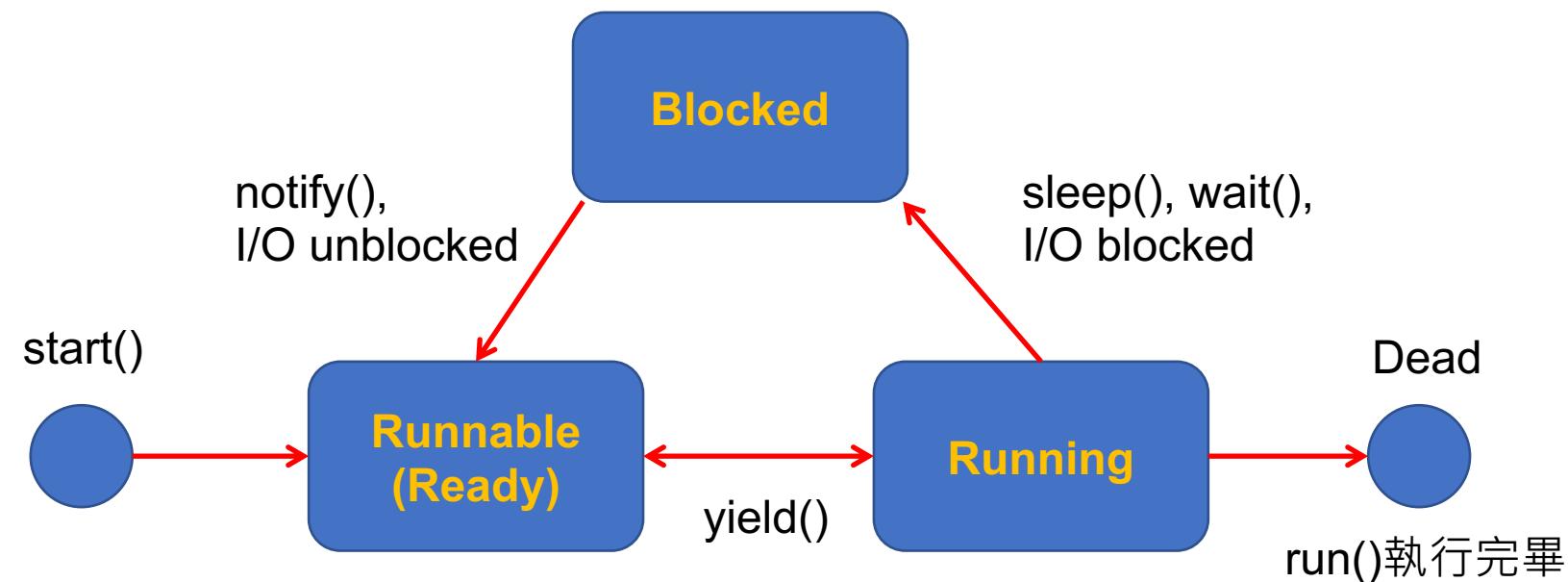
- 需等待一段不確定的I/O時間，移出執行狀態

另有暫停狀態(Suspended)

透過resume()方法回到預備狀態 (註：不再使用)
本課程不會提及

執行緒生命週期 (Life cycle)

執行緒皆可藉由interrupt()方法
從不可執行狀態恢復為預備狀態



Thread類別常用方法整理

範例：
[MyThread.java](#)
[TestMyThread.java](#)

方法	說明
void run()	執行緒執行的地方
void start()	啟動執行緒
void setName(String name)	設定執行緒的名字
String getName()	回傳執行緒的名字
void setPriority(int newPriority)	設定執行緒的優先權為newPriority
int getPriority()	取得執行緒的優先權值
boolean isAlive()	判斷目前執行緒狀態是否為存在
void setDaemon(boolean on)	設定為背景執行緒
boolean isDaemon()	判斷是否為背景執行緒
static Thread currentThread()	回傳目前正在執行的執行緒物件
void join() throws InterruptedException	等待此執行緒執行終止
static void yield()	使目前的執行緒讓出執行權
static void sleep (long milliseconds) throws InterruptedException	使執行緒休眠多少毫秒

模組23

控制執行緒與優先安排

- 23-1：
控制執行緒 – join方法
- 23-2：
執行緒優先權觀念
- 23-3：
執行緒獨佔與禮讓議題

控制執行緒與優先安排

範例：
NewThread.java
TestJoin.java

- 執行緒join其它執行緒：等它所呼叫的執行緒終止後再繼續執行
 - join()
 - join(long milliseconds)
- 優先權(Priority)可確保重要或急迫性執行緒可被立即或經常執行
 - 流程安排(Scheduling)是決定多個執行緒的執行順序
 - 優先權的值為1至10的整數，由Thread類別定義三種常數：

Thread.MIN_PRIORITY : 最小值 = 1

Thread.NORM_PRIORITY : 預設值 = 5

Thread.MAX_PRIORITY : 最大值 = 10

- 可使用setPriority() 和 getPriority()方法重新設定和取得優先權值
- 優先權較高者先執行，但優先權相等時，並非是等待最久者先執行，而是任選其一執行

控制執行緒與優先安排

- 執行緒切換發生於：
 - 有較高優先權的執行緒進入排程時
 - 執行緒被終止執行或 `run() method` 執行完畢
 - Time-Slice系統：系統分配時間用完了
- 註：在一般情況下擁有最高優先權的執行緒先執行，不過有時候會有例外，因有時排程器會挑選優先權較低者來執行，以避免餓死(starvation)的情形，因此優先權的使用只是為了讓排程更有效率而已，勿在程式中完全使用優先權的關係來控制程式的進行
- 時間分割(Time-slicing)：
 - 如windows系統會將CPU的時間分成一段段的時間糟(time slot)，特性為：
 - 具time-slicing特性作業系統將time slot分給「Priority最高且相等的數個執行緒」，直到執行完畢或是被更高優先權的執行緒搶走
 - 無法保證time slot的平均分配，也不保證執行的先後順序

控制執行緒與優先安排

- **自私的執行緒(Selfish thread)：**
 - 自私的執行緒實踐了「socially – impaired」，其特性為：
 - 擁有「密實迴圈(tight loop)」，將一直獨佔CPU執行權
 - 如果系統不支援Time-Slice則易完全獨佔，直到：
 - 該迴圈執行完畢
 - 或被更高Priority之其它執行緒搶走CPU執行權
- **禮讓的執行緒**
 - 用yield()改進，自願移出執行(Running)狀態至預備(Ready)狀態
 - 對Priority相等的數個執行緒有效

範例：
`SelfishRunner.java`
`SelfishTest.java`
`PoliteRunner.java`
`PoliteTest.java`

控制執行緒與優先安排

- **多執行緒程式的特性(結論一)：**
 - 多執行緒是難以預測其行為的
 - 執行緒的執行順序無法完全保證
 - **Task Switches**可能在任何時刻任何位置發生
 - 執行緒對於小改變有高度的敏感性
 - 執行緒並不總是立刻啟動執行(需被排程)
- **多執行緒程式的特性(結論二)：**
 - 在時間分割(Time-Slicing)系統中優先權相同的執行緒會以一種**幾乎**相同機會的循環方式來執行，甚至優先權較低的執行緒也能取得時間糟(time slot)的一小部份，**其比例大約正比於他們的優先權值**，因此在長時間執行中不會有執行緒完全都沒有被顧及到
 - 在非時間分割(Time-Slicing)系統中則易發生**完全**獨佔的情況
 - 對於有大量運算的執行緒應適度的呼叫yield()來讓其它執行緒有執行的機會，尤其可增加圖形使用者介面(GUI)的良好互動

模組24 多執行緒同步

- 24-1：
synchronized關鍵字
- 24-2：
同步鎖定3種操作
- 24-3：
同步執行效率與穩定度

同步 (Synchronization)

範例：[TestSync0.java](#)

- 為什麼使用synchronized關鍵字
 - 原因：在程式的某Critical Section(危險區域)裡，不同執行緒可能同時存取同一份資源因而產生衝突或重複修改的問題
 - 目的：控制每次只能有一個執行緒在使用同一份資源，此時另外的執行緒無法同時使用此同一份資源
 - 舉例：銀行領錢問題
- 使用同步的概念
 - **Monitor**：物件都有一個Monitor，用來當每次只能有一個執行緒進入獨佔的鎖(Lock)(或稱旗標Flag)
 - **進入Monitor**：在Java裡，由呼叫已經synchronized關鍵字修正過的method即進入Monitor並得到鎖(Lock)。此時其它所有嘗試進入Monitor的執行緒將會暫停(Blocked狀態)直到該執行緒離開

同步 (Synchronization)

- 使用同步的方法

- 先找出Critical Section(危險區域)後：

- 1. 在方法宣告中加入synchronized關鍵字

- 2. 或程式區塊以synchronized標示

- 3. 或類別資料以synchronized標示

- 再執行已經用synchronized修正過的方法(method)或程式區塊

範例：
TestSync1.java
TestSync2.java
TestSync3.java

同步與系統效能/穩定度

- 執行效率與程式的穩定度
 - 同步化是一個相當耗時的運算，除非必要否則應減少使用，尤其是常執行的方法(method)或程式區段
 - 然而妥善的運用對程式的穩定度和強健度有極大幫助
- 延伸集合工具：(Vector / Hashtable)與JDK 5 Collections類別新增方法
 - Vector等同於**ArrayList**，但**Vector**提供同步化的優點(與負擔)，對多執行緒的存取是很重要的
 - Hashtable等同於**HashMap**，但**Hashtable**提供同步化的優點(與負擔)，對多執行緒的存取是很重要的
 - 以上兩種集合都是在JDK 1.0時就存在的類別
 - Collections類別於1.5時提供了有同步處理機制集合的各種方法，詳見API文件

模組25

多執行緒溝通

25-1：
執行緒之間互動

25-2：
wait方法

25-3：
notify與notifyAll方法

執行緒通訊

- **執行緒間的通訊(interthread communication)**
 - 目的：讓執行緒之間可互相交談，彼此等待
 - 方式：
 1. 可透過共同使用的資料交談
 2. 或使用執行緒控制的方法(method)，如join()
 3. 或於**synchronized**的方法內使用**wait()**, **notify()**, **notifyAll()**等更細微溝通機制，彼此等待，以避免「**生產過剩、不足**」或是「**消費過剩、不足**」的問題，並使CPU使用更有效率

執行緒之間等待與通知

範例：[TestWaitNotify.java](#)

- **wait(), notify(), notifyAll()**這些方法在Object類別裡被實作成final的方法，所有Java類別都可以使用
 - **wait()**：當一執行緒呼叫wait()方法時，會放棄monitor，將lock釋放出給另一個正等待進入monitor的執行緒，並且進入等待執行緒群(pool)開始等待，直到等待時間終了，或是被另一個進入相同monitor的執行緒呼叫notify()或notifyAll()方法所叫醒，而再進入ready狀態
 - **notify()**：隨機叫醒在相同物件上某一個正在waiting的執行緒
 - **notifyAll()**：叫醒在相同物件上所有正在waiting的執行緒，priority最高者將第一個執行

模組26 死結問題

- 26-1：
死結發生原因
- 26-2：
哲學家用餐與死結問題
- 26-3：
stop方法與API
Deprecated方法

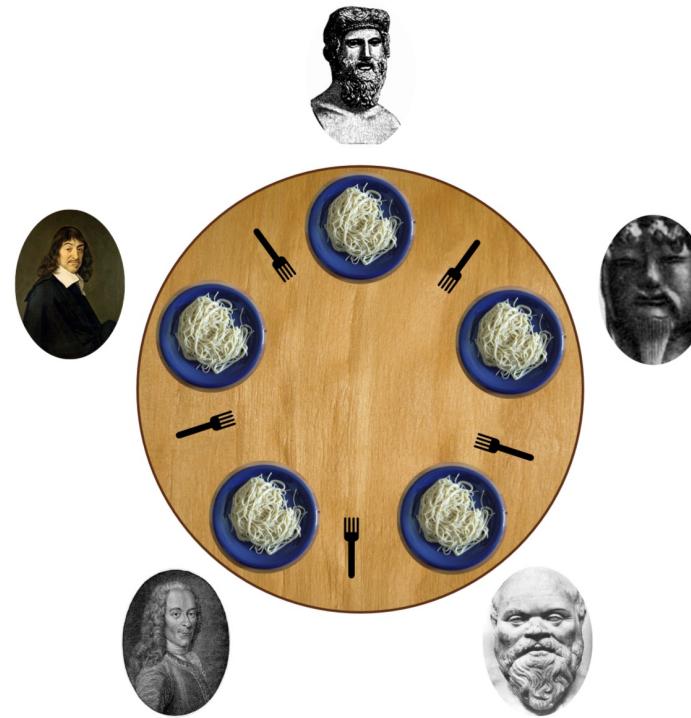
什麼是死結(DeadLock)

範例：
`DeadLock.java`
`NoDeadLock.java`

- 死結(DeadLock)
 - 原因：多執行緒的同步化鎖定(synchronized)可能造成執行緒間相互等待的死結。
而Java無法偵測或預防死結的發生，須由程式設計師自行控制與掌握
 - 避免死結：Java編譯器無法在編譯時期得知是否會有死結發生的可能，再加上多執行緒行為難以預測
最好方法是預防而非偵測它
 - 預防死結：最簡單方式是將一群物件的鎖定動作，按照相同的順序進行處理

哲學家用餐與死結

- 哲學家用餐問題
<http://zh.wikipedia.org/wiki/%E5%93%B2%E5%AD%A6%E5%AE%B6%E5%B0%B1%E9%A4%90%E9%97%AE%E9%A2%98>
- 此議題描述了在多執行緒同步環境下，會造成死結狀況的說明，也是在電腦科學領域裡的經典問題



圖片來源：維基百科(連結同上) 349

過時API (Deprecation)

- 隨著Java版本的更新，標準類別函式庫 (class library)除了會加入新的類別/介面跟其相關的屬性、方法與建構子等等，也會有取代既有的內容
- 在原始碼進行編譯時，若是出現了“xxx.java uses or overrides a deprecated API”時，就表示我們使用到了過時的內容
- 在編譯時，可以透過指令 `-deprecation` 來檢視程式碼裡哪些敘述用到了過時內容
如：`javac -deprecation HelloWorld.java`
- 像`stop()`、`resume()`方法已被列為Deprecated，就是因為容易發生死結問題，所以不建議我們使用

模組27

數字與文字資料

27-1：
Math類別介紹

27-2：
StringBuffer類別

27-3：
StringBuilder類別

工具類別 - Math

- `java.lang.Math`類別提供許多數學上實用的方法如亂數、絕對值、平方根、立方根與三角函數等，讓程式設計師在設計時，省去許多數學運算程式碼的撰寫
- 因為`java.lang.Math`類別所提供的屬性與方法都是**類別等級**(都是`static`修飾子)，因此我們只要透過`Math`類別名稱，即可呼叫所需的屬性或方法，非常方便
- 常用方法，見下一頁：

Math類別常用方法

範例：
[TestMath.java](#)
[TestNumber.java](#)

方法	說明
double abs(double a) float abs(float a) int abs(int a) long abs(long a)	回傳a的絕對值
double max(double a, double b) float max(float a, float b) int max(int a, int b) long max(long a, long b)	比較a, b大小後，回傳較大者
double min(double a, double b) float min(float a, float b) int min(int a, int b) long min(long a, long b)	比較a, b大小後，回傳較小者
double pow(double a, double b)	回傳a的b次方運算結果
double random()	回傳一個double類型的亂數，介於0.0(含)~1.0(不含)
double sqrt(double a)	回傳a的正平方根
double cbrt(double a)	回傳a的立方根

StringBuffer類別

範例：[TestStringBuffer.java](#)

- String類別不可在原字串所在記憶體位置改變字串內容，StringBuffer類別則在原字串所在記憶體位置改變字串內容 (append, insert, delete, replace)
- 使用StringBuffer類別中的任何方法時，回傳的字串會使用原有的記憶體空間
 - 創建者設計模式 (Builder Pattern)
- StringBuffer字串與String字串不可以比較 (沒有意義)
如：

```
String s1 = new String("test");
StringBuffer s2 = new StringBuffer("test");
if (s1 == s2) {...}      //false
if (s1.equals(s2)) {...} //false
```

StringBuilder類別

- `StringBuilder`類別是JDK 5的新類別，其用法與`StringBuffer`類別完全一樣(`append`, `insert`, `delete`, `replace`)
- 舊類別`StringBuffer`是thread-safe，新類別`StringBuilder`則是not thread-safe
- 使用上，如果不考慮多執行緒的問題就可以使用`StringBuilder`來提升執行的效率
- 常見於結合`BufferedReader`的`readLine()`方法進行文字串接，對執行資源損耗減輕不小

模組28

Regular

Expression

- 28-1：
正則表達式(Regex)介紹
- 28-2：
正則表達式驗證資料
- 28-3：
正則表達式切割字串

正則表達式 (Regular Expression)

- 正規表示法(Regular Expression)就是由許多樣式的符號組成的樣式句，主要功能就是用來比對文字是否符合該規則的要求
- 正規表示法並非Java語法，但為了通過編譯，都是以字串型式存在，等到要執行時，再由特定的編譯器進行處理
- 正規表示法在本課程會簡略說明，有興趣的同學可以在網路上搜尋到更多規則與說明

正則表達式常見用法

符號	說明
[ABC]	A、B、C任一個字元都符合要求 例：[ABC]ook，可以是Aook, Book或Cook
[^ABC]	不可以含有A、B、C任一個字元 例：[^ABC]ook，就不可以是Aook, Book或Cook
[A-C]	可以是A到C連續字元的任何一個 例：[A-C]ook，可以是Aook, Book或Cook
[^A-C]	不可以含有A到C連續字元的任何一個 例：[^A-C]ook，就不可以是Aook, Book或Cook
{n,m}	代表指定字元出現次數最少n次，最多m次 逗號之間不得有空白，n與m都要是大於等於0的整數 n <= m 例：Book{1,2}代表k最少要出現1次，最多2次，即Book或Bookk
{n}	代表指定字元正好出現n次 例：Book{1}代表k要出現正好1次，所以只能是Book
{n,}	代表指定字元至少出現n次 例：Book{1,}代表k要出現至少1次以上，可以是Book, Bookk...

正則表達式常見用法

符號	說明
\d	可以是0~9任何一個數字，相當於[0-9]
\D	不可以是0~9任何一個數字，相當於[^0-9]
\s	可以是空白的字元
\S	不可以是空白的字元
\w	可以是一個英文字母或數字
\W	不可以是任何英文字母或數字
?	指定字元最多出現1次，也可以不出現，相當於{0,1} 例：S?PP就可以是PP或SPP。?在S後面，則是S受到?限制
+	指定字元至少要出現1次以上，相當於{1,}
*	指定字元出現0次以上，相當於{0,}
.	任一字元

正則表達式常見用法

符號	說明
()	括弧內，代表同一個群組 例：(SPP){2}代表SPP是一體，必須同時出現2次：SPPSPP
\	取消原運算符號的功能，使其成為單純文字 例：2014\06\13代表取消「-」符號功能，這與Java的跳脫符號「\」相同，所以結果一定要是2014-06-13
^	做為一行正則表達式的開始標記
\$	做為一行正則表達式的結束標記

- 可搭配String類別提供的boolean matches(String regex)方法，回傳結果即為是否符合正則表達式的文字格式

字串切割

範例：
[TestSplit1.java](#)
[TestSplit2.java](#)

- 文字切割是程式設計師在處理文字資料時常見的操作，Java也提供了相關的類別與方法，而切割操作也可以結合正則表達式規則使用
- String類別在JDK 1.4時加入了 `String[] split(String regex)`方法，符合運算式的部份就會被當成分隔符號移除掉，剩下部份就會置入於最後回傳的字串陣列裡
- StringTokenizer類別也可以將一個字串分成數個字串處理

模組29

日期時間資料 (1)

- 29-1 :
Calendar類別
- 29-2 :
GregorianCalendar類別
- 29-3 :
Calendar設計探討

Calendar類別與相關常數

範例：[TestCalendar.java](#)

- 取得今天的日期與現在時間
 - `Calendar rightNow = Calendar.getInstance(); //Calendar為抽象類別`

常數名稱	值
<code>Calendar.YEAR</code>	年
<code>Calendar.MONTH</code>	(+1) 月
<code>Calendar.DATE</code>	日
<code>Calendar.HOUR_OF_DAY</code>	24時制的時
<code>Calendar.MINUTE</code>	分
<code>Calendar.SECOND</code>	秒
<code>Calendar.DAY_OF_WEEK</code>	(-1) 星期幾

GregorianCalendar類別

範例：
[TestGregorianCalendar.java](#)

- GregorianCalendar 為Calendar的子類別，適合用來設定某一**特定的日期時間**
- 呼叫建構子即可傳入需要的年、月、日甚至時、分、秒來指定想要的特定時間點
- GregorianCalendar中文可稱為格里曆，是目前世界上各國常用的標準日曆系統，其實也就是我們所謂的”國曆”或”陽曆”的制度
- 像是isLeapYear()方法，就是只有GregorianCalendar類別才有定義的方法，使用此方法就不可以用多型的宣告方式，如：Calendar cal = new GregorianCalendar();

Calendar抽象類別

- Calendar類別為抽象類別，所以不能藉由new關鍵字產生實體
- 但可使用該類別的方法public static Calendar getInstance()來獲得一個“Calendar類型”的物件
- 將於課堂上搭配Calendar原始碼說明該設計目的為何

模組30 日期時間資料 (2)

30-1：
java.util.Date類別
30-2：
java.sql.Date類別
30-3：
認識系統時間

父子同名的Date類別

- 取得今天的日期與現在時間
 - `java.util.Date rightNow = new java.util.Date(); //java.util.Date表示某一時間點`
- 在Java 1.0.2版中`java.util.Date`有數種功能，但在Java 1.1開始，其中大部份的方法都已被淘汰(Deprecated)，所以`java.util.Date`目前功能就是**表示某一時間點**
- 利用`Calendar`物件的`getTime()`方法，可產生`java.util.Date`物件
- 小心！在做`import`(引入套件)設定，不要選錯套件了

父子同名的Date類別 (續)

範例：[TestDate.java](#)

- `java.sql.Date`為`java.util.Date`的子類別，用在**資料庫的日期欄位資料的對應與格式**
- 若是使用建構子產生`sql.Date`物件的話，參數一定要提供一個為`long`型別的時間資料
- 此類別只有對應到年、月、日的格式，不包含時、分、秒
- 顯示格式為 `yyyy-mm-dd`

作業系統時間

- 我們使用日期/時間相關API取得的資料，事實上是向執行環境的作業系統取得的系統時間。若是系統時間本身是有問題的，當然就會連帶影響程式執行的結果
- 利用`java.util.Date`物件的`getTime()`方法，可得到自**1970年1月1日0時0分0秒起的毫秒**
- 該時間也被稱為**系統起始時間**，普遍的說法是為了紀念**UNIX**作業系統的誕生，所以制定此時間點為開始時間
- 程式裡只要遇到`long`型別的時間資料，就是代表起始時間開始的總毫秒數

模組31

日期時間資料 (3)

- 31-1 :
DateFormat類別
- 31-2 :
SimpleDateFormat類別
- 31-3 :
printf方法介紹

java.text.DateFormat類別

範例：[TestDateFormat.java](#)

- 用來格式化 `java.util.Date`，可設定國別格式與時區，用在**國際化**
- `DateFormat`類別與`Calendar`類別設計相似，也是一個抽象類別，但可以透過`getInstance()`來取得一個實體進行格式化的動作
- 結合`TimeZone`與`Locale`類別，可輕鬆完成日期時間的格式轉換
- 註：`TimeZone`與`Locale`均為`java.util`套件裡的類別

java.text.SimpleDateFormat類別

範例：[TestFormatter.java](#)

- 用來格式化 `java.util.Date`，可以用更簡便的做法完成日期格式化
- 使用上，利用建構子的呼叫，傳入想要的格式定義與代號完成初始化後，即可用 `format()` 得到結果（注意 `format()` 回傳為字串的結果）
- 代號的定義請查閱 Java API 文件！
- 額外補充：`DecimalFormat` 類別操作使用

printf方法

範例：[TestFormatter.java](#)

- Simple Formatter Output(簡易的格式化輸出：printf)

- JDK 5新增System.out.printf()方法

- printf()方法源自於java.util.Formatter類別

- 比如只需寫出：

```
java.util.Date d = new java.util.Date();
System.out.printf("%tY/%<tm/%<td %<tT%n", d);
```

即可輸出：2014/06/13 17:38:20之結果

- 相關代碼定義，也可以從Java API文件裡取得資訊

模組32

System類別

- 32-1：
System類別介紹
- 32-2：
系統屬性操作
- 32-3：
gc方法與垃圾回收機制

java.lang.System類別

範例：[Elapsed.java](#)

- System類別為Java 1.0即存在，用來對應到系統環境相關的資訊取得或操作，像是取得系統時間與系統屬性等
- 需注意的是，System類別沒有建構子的宣告，這也代表我們無法產生System類別的物件實體
- 因此，System類別裡面的所有屬性與方法全部為static等級
 - 如：System.out、System.gc()等...

系統屬性 (System Property)

範例：[TestProperties.java](#)

- 系統屬性(System Properties)可以顯示系統的環境資訊
- public static Properties getProperties() :
 - 取得所有的系統屬性
 - 回傳Properties類別的物件
- public static String getProperty(String key) :
 - 回傳特定系統屬性名稱的值
- public static String setProperty(String key, String value) :
 - 設定特定的系統屬性
 - 要設定系統屬性，也可在程式執行時設定
 - `java -DmyProperty=myValue HelloWorld`

-D與屬性之間不得有任何空白字元

垃圾回收機制(Garbage Collector)

範例：[TestGC.java](#)

- 通常Java會在記憶體不足時，自動執行垃圾收集的動作
- 如果想要自己強制Java進行垃圾收集時，可透過使用System.gc();的方法
- JVM將記憶體空間最佳化後，就會將控制權還給原來進行中的程式
- 嚴格來講System.gc();只是建議系統應啟動GC，它不一定會完全執行，我們也不知道GC的正確啟動時間

模組33

Runtime類別

- 33-1：
Runtime類別介紹
- 33-2：
Runtime與Singleton
- 33-3：
Runtime類別常用方法

Runtime類別與Singleton模式

範例：[TestRuntime.java](#)

- Runtime類別作為代表執行環境，此類別提供的方法讓我們可以取得執行時的資訊
- Runtime實體的取得方式並非是透過 new 關鍵字，而是呼叫該類別提供的一個 static 方法：`getRuntime()`
- 此種設計為一個最具Singleton模式的代表作！
- **Singleton模式**：此類別在執行期間，只會有一個物件實體存在，因此也常被稱之為“單例模式”

模組34 列舉類型 Enum

- 34-1：
列舉(enum)類型介紹
- 34-2：
列舉進階設計
- 34-3：
列舉使用情境

列舉(enum)類型 JDK 5

範例：
[EnumeratedTypes1.java](#)

- **列舉類型(enum)**

- 列舉enum適合使用在某些狀況的表現(如：一年有四季、一週有七天)
- enum除了常數設置功能外，還給了您許多編譯時期的檢查功能

- **enum的特性**

- enum本質上還是一個類別，編譯器會將enum轉成類別，其內部除可定義本身的enum types(列舉子)外，仍可以有fields、methods跟constructors，但與一般類別不同的是：
 - enum本身不具備類別的某些功能，如繼承
 - constructors不能為public和protected，因enum型別不能產生物件
- 必需使用關鍵字 enum 定義列舉型態
 - enum型態預設繼承自java.lang.Enum類別
 - enum types(列舉子)預設為public static final，而列舉子的值其實是它本身的名稱
- 它可以和泛型以及增強型for迴圈(for-each)很好地搭配
- 它也可以使用在switch控制中

列舉(enum)進階使用

- 更複雜的enum列舉定義：
 - 它可以擁有自訂建構子(constructors)，欄位(fields)以及方法 (methods)
 - 詳見範例內說明：

[EnumeratedType2_1.java](#)

[EnumeratedType2_2.java](#)

列舉(enum)應用

- 實際應用例參考：
 - 詳見範例額外補充的演進流程，共3組
 - PrivateShirt.java + PrivateShirtTest.java
 - ColorCode.java + PrivateShirt2.java + PrivateShirtTest2.java
 - ColorCode2.java + PrivateShirt3.java + PrivateShirtTest3.java

模組35 內部類別 (1)

35-1：
內部類別目的與好處
35-2：
static內部類別特性
35-3：
成員內部類別特性

內部類別 (Inner Class)

- Java可以將一個類別變成另一個類別的成員，如下：

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

- Java 1.1之後，除了先前所探討的一般類別與介面之外，額外定義了四種內部類別(inner class)
 - 靜態內部類別 (static inner class)
 - 成員內部類別 (member inner class)
 - 區域內部類別 (local inner class)
 - 匿名內部類別 (anonymous inner class)

使用內部類別 (Inner Class) 目的

範例：
OuterDemo1.java
OuterDemo2.java

- 使用內部類別的好處
 - 可以直接存取外部類別的私有(private)成員
 - 例如在視窗程式中，可以使用內部類別來實作一個事件傾聽者 類別(Listener)，這個視窗傾聽者類別可以直接存取視窗元件，而不用透過參數傳遞
 - 另一個好處，當某個slave類別只完全服務於一個master類別時，就可以將之設定為內部類別，如此使用master類別的人就不用知道slave的存在

static內部類別特性

範例：StaticInner.java

- 內部類別在宣告時可根據功能上的需要加上static修飾字，即成為了一個靜態內部類別
- 需注意的是，因為載入先後順序的關係，所以：
 1. static內部類別無法存取外部類別的實體成員，會編譯失敗
 2. static內部類別可以不藉由外部類別即可直接存取並實體化物件
(需注意存取修飾關係)

成員內部類別特性

範例：[MemberInner.java](#)

- 內部類別在宣告時，若不是在方法裡，也沒有**static**關鍵字，即為成員內部類別
- 需注意的是：
 1. 因為成員內部類別依附在外部類別的物件實體上，所以必須先產生外部類別的物件實體才能再產生內部類別的物件實體
 2. 成員內部類別除了可以存取外部類別的實體成員外，也同樣可以存取**static**屬性與方法

模組36 內部類別 (2)

- 36-1：
區域內部類別特性
- 36-2：
匿名內部類別特性
- 36-3：
實作匿名內部類別

區域內部類別特性

範例：[LocalInner.java](#)

- 宣告在方法裡，即為區域內部類別，跟區域變數一樣，不能宣告存取修飾字
- 需注意的是：
 1. 區域內部類別會隨著方法執行完畢後跟著被釋放，所以需注意宣告位置與執行順序的關係
 2. 區域內部類別若是使用了所屬方法的參數或變數時，被使用到的參數或變數都得宣告為final，也就是不得修改 (原因於課堂說明)

匿名內部類別特性

範例：[Anonymous.java](#)

- 匿名類別多見於對介面的實作，算是一種在撰寫程式碼時，達到簡化語法的設計
- 參考範例說明與實作
- 額外補充：內部類別/匿名類別與.class檔的關係

模組37

Java 8

Lambda語法

37-1：
匿名類別與Lambda

37-2：
Lambda語法結構

37-3：
SAM介面

為何需要Lambda

- 為何一定要加入Lambda呢？
- (1) 時代在改變
 - 1995年那時，主要的程式語言都沒有支援closures(閉包)，例如Fortran, C, Pascal
- (2) Java是最後一道防線
 - C++加進去啦
 - C#在3.0時也加進去啦
 - 現今新起的語言通通都加啦
- (3) JSR 335講好要加入Lambda了

為何需要Lambda

- 讓我們先從內部類別開始說起(參看下圖)：

```
public class OuterClass {  
    private String message = "Hi there";  
  
    public class InnerClass {  
        public void print(){  
            System.out.println("Message from inner: " + message);  
        }  
    }  
  
    public void print(){  
        System.out.println("Message from outer: " + message);  
        InnerClass ic = new InnerClass();  
        ic.print();  
    }  
  
    public static void main(String[] args) {  
        OuterClass oc = new OuterClass();  
        oc.print();  
    }  
}
```

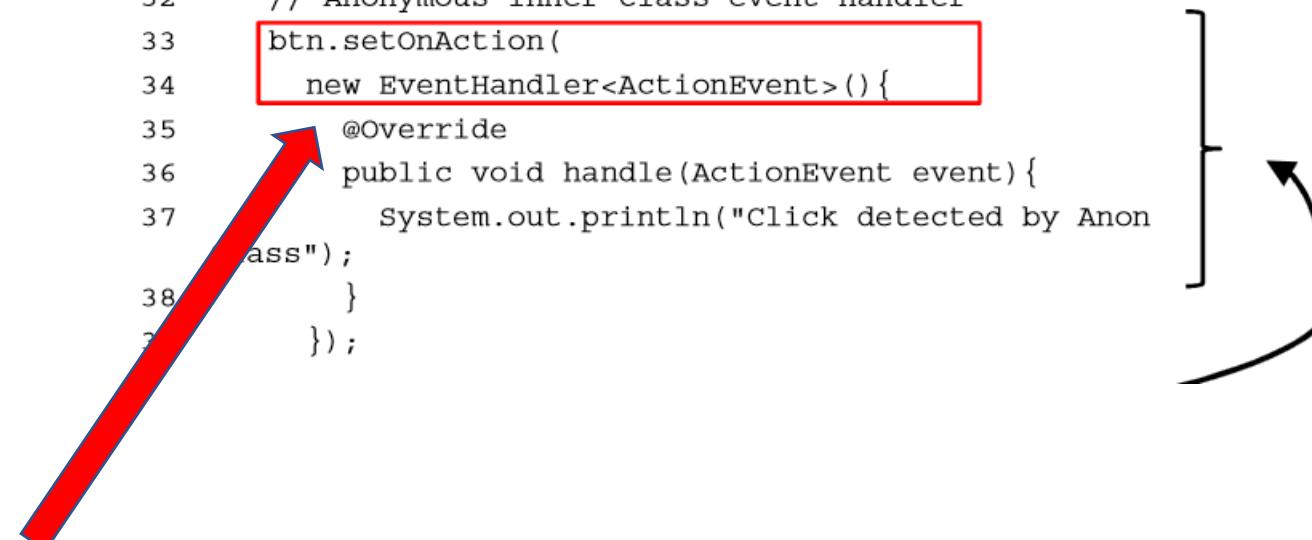
為何需要Lambda

- 為什麼要設計成內部類別？
- (1) 讓彼此有邏輯關聯性的類別們能在同一個地方使用
 - 設計為內部類別即可**直接使用外部類別的private成員**，無須再透過參數傳遞，增加了使用上的便利性
- (2) 增加封裝設計的安全性
 - 因為**內部類別可以宣告為private**，這樣外界就不會知道內部類別的存在，達到隱藏細節的設計
- (3) 增加程式閱讀性與維護性
 - 因為程式碼都在同一隻Java類別裡，所以在調整與閱讀理解上方便

為何需要Lambda

- 再看看匿名內部類別

```
32      // Anonymous inner class event handler
33      btn.setOnAction(
34          new EventHandler<ActionEvent>() {
35              @Override
36              public void handle(ActionEvent event) {
37                  System.out.println("Click detected by Anon
38                  ass");
39              }
40          });
41      }
```



- 垂直問題
- 不好處理的語法
- 無法重複使用
- 會產生額外的.class檔

為何需要Lambda

- 改成Lambda表示式

```
43 // Lambda expression
44 btn2.setOnAction(
45     (ActionEvent e) -> {System.out.println("Test 2 button
        clicked");}
46 );
```

- 垂直問題解決 (一行完成)
- 乾淨的語法
- 可重複使用
- 不再產生額外的.class檔**

Lambda Expression

- 基本的Lambda表示式
 - `(int x, int y) -> x + y`
 - `(x, y) -> x + y`
 - `(x, y) -> { System.out.println(x + y); }`
 - 兩個以上參數或無參數需要小括號，一個參數可省略小括號
- 註：區塊 (Block) 裡是可以多行敘述的 (但 lambda 風格不建議這麼做)
 - 實作方法有回傳值的話：單行敘述不必 `return` 關鍵字，多行敘述一定要有 `return`

Parameter List	Arrow token	Body
<code>(int x, int y)</code>	<code>-></code>	<code>x + y</code>

Method Reference

- Lambda 語法裡對方法呼叫的簡化，主要有四種用法：
 - 類別方法 (**類別名::方法名**)
(args) -> ClassName.method(args) 可簡化為 ClassName::method
如：(a, b) -> Math.pow(a, b) 就可以變成 Math::pow
 - 成員方法 / 一般方法 (**物件型別::方法名**)
(obj, args) -> obj.method(args) 可簡化為 ClassName::method (ClassName是指obj的型別)
如：(s1, s2) -> s1.endsWith(s2) 就可以變成 String::endsWith

Method Reference

- Lambda 語法裡對方法呼叫的簡化，主要有四種用法(續)：

- 成員方法 / 一般方法 (**物件參考變數::方法名**)

(args) -> obj.method(args) 可簡化為 obj::method

(obj並不是由參數傳入，而是前面程式碼已有的參考變數)

如：Employee emp = new Employee();

(s) -> emp.setEname(s) 可簡化為 emp::setEname

- 建構子 (**類別名::new**)

(args) -> new ClassName(args) 可簡化為 ClassName::new

如：() -> Employee::new

(i1, s1) -> Employee::new (Employee需要有對應參數的建構子才可以)

Single Abstract Method介面

- 剛剛是怎麼辦到的!?

```
package javafx.event;

interface EventHandler<T> {
    public void handle(T event);
}
```

- 因為該介面只擁有一個抽象方法 (**Single Abstract Method**)
- 所以它就被稱為SAM (Java 8裡又稱為**Functional Interface**(函式介面))
- 例如Runnable, GUI Listeners, Comparator都是



您好，
我是Sam ^^ !!



好巧喔！
我也叫Sam

模組38

Java 8

函式介面

- 38-1：
java.util.function套件
- 38-2：
四大函式介面介紹
- 38-3：
基本型別/特殊函式介面

java.util.function套件

- Java 8的標準API新增了一個套件為java.util.function，請注意查閱時，API文件的版本
- 此套件裡全為函式介面(Single Abstract Method)，共有43個(幫你算好了！)
- 掌握接下來要說明的主要函式介面特性，即可延伸出其它函式介面的功能
- 因為Lambda語法只能搭配函式介面使用，所以跟此套件的關係密不可分

java.util.function的四大介面

- 最重要的四個介面：

範例：
[TestPredicate.java](#)
[TestConsumer.java](#)
[TestSupplier.java](#)
[TestFunction.java](#)

名稱	回傳值	方法簽章 (T, R為搭配泛型)
Predicate<T>	boolean	public boolean test(T t);
Consumer<T>	void	public void accept(T t);
Supplier<T>	T	public T get();
Function<T, R>	R	public R apply(T t);

基本型別函式介面

範例：
`TestToPrimitiveFunction.java`
`TestPrimitiveFunction.java`

- 因為泛型需指定的都是類別型態的資料，所以對於`Integer`, `Double`, `Long`這些包裝類別的資料都要再透過auto boxing/unboxing方式來取得基本型別資料，進而增加了執行上效率的損耗
- 可利用`java.util.function`的primitive interface來進行基本型別資料的處理，避免auto-boxing/unboxing造成過多的資源浪費
- 例如：
 - 回傳一個基本型別資料：
 - ToDoubleFunction,ToIntFunction, ToLongFunction**
 - 傳入一個基本型別資料
 - DoubleFunction, IntFunction, LongFunction**

特殊函式介面

範例：
[TestBinary.java](#)
[TestUnaryOperator.java](#)

- **Binary Types**

- 可以傳入兩種類型參數，達到需要實現的邏輯
- 如：`BiPredicate<T, U>`, `BiConsumer<T, U>`, `BiFunction<T, U, R>`

- **UnaryOperator**

- 傳入與回傳的資料型別都是相同的情況下，即可以使用 `UnaryOperator<T>` 介面實現

模組39

Java 8

Stream API (1)

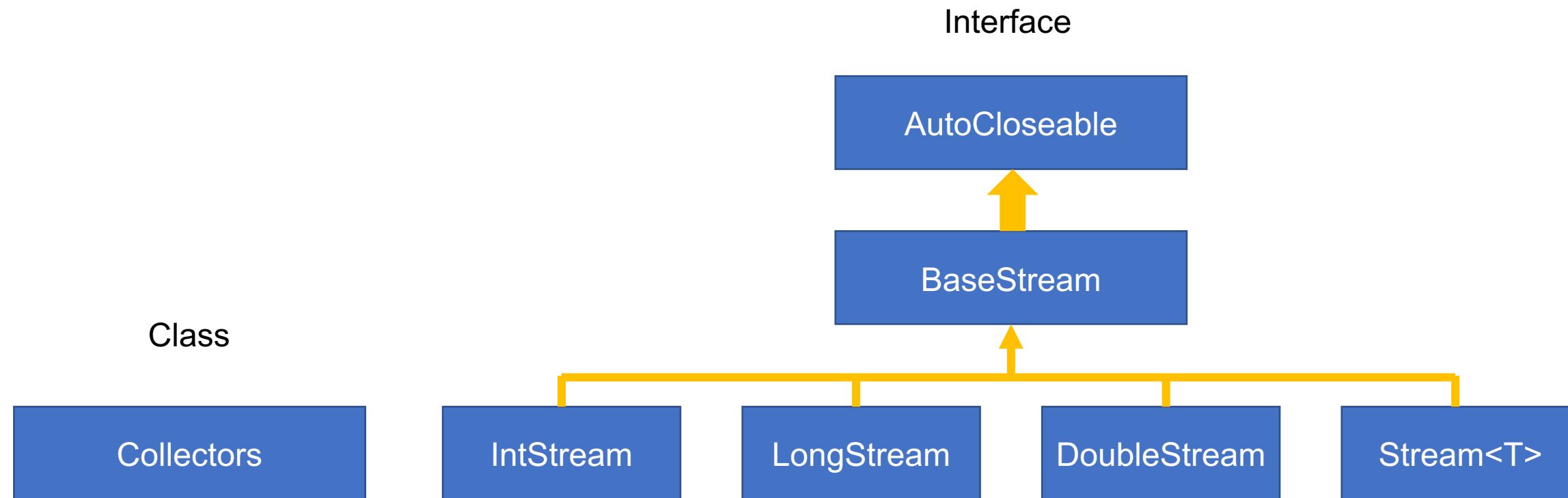
39-1：
Stream API架構介紹

39-2：
方法鍊與Builder模式

39-3：
filter與map中繼方法

Stream API

- **java.util.stream**套件



Stream API特性

- 串流 (Stream)
 - java.util.stream
 - 對於一組元素的處理可用各種方法串接起來使用
- 方法鍊 (Method chaining)
 - 可以把對集合操作的方法們組合成一個敘述
- 串流特性
 - 不可變的 (immutable)
 - 只要該元素使用過，就無法再從此串流裡再次取用
 - 一次串流操作視為一次性的使用 (如同java資料流)
 - 可以是序列串流(預設)，或平行串流

Stream API特性(續)

- Stream可以將一個集合轉化成一條管線 (pipeline)
 - 管線使用期間，是不可以對裡面的資料進行變動的 (immutable)
 - 管線為一次性使用，用完即扔
- 一個管線組成會有以下內容：
 - **1個源頭(source) (例如一個集合，或檔案，甚至是stream都可以)**
 - **0 ~ n個中繼操作 (intermediate operation)**
 - **1個結束操作 (terminal operation)**
- 如：
 - one collection
 - filter or map (這裡的map是指Stream API的map方法，不是map集合)
 - forEach

創建者模式(Builder Pattern)

- **Builder Design Pattern (創建者模式)**

- 程式碼更簡單更好閱讀
- 讓物件的創建變得更為彈性
- 方法本身回傳自己 (this)
- 寫程式時的流暢度 (方法一個接著一個呼叫)

```
penList.add(  
    new Pen.Builder()  
        .setBrand("SKB")  
        .setPrice(10)  
        .setStock(50)  
        .setColor(Color.BLUE)  
        .setCanErase(false)  
        .build()  
);
```

方法鍊 (Method chaining)

- 管線設計允許我們可以進行方法鍊設計 (就像創建者模式一樣)
- 其中方法包含filter跟其他串流方法

- 如：

```
penList.stream()
    .filter(p -> p.getStock() >= 50 && p.getStock() <= 100 && p.getPrice() <= 1000)
    .forEach(Pen::printDetails);
```

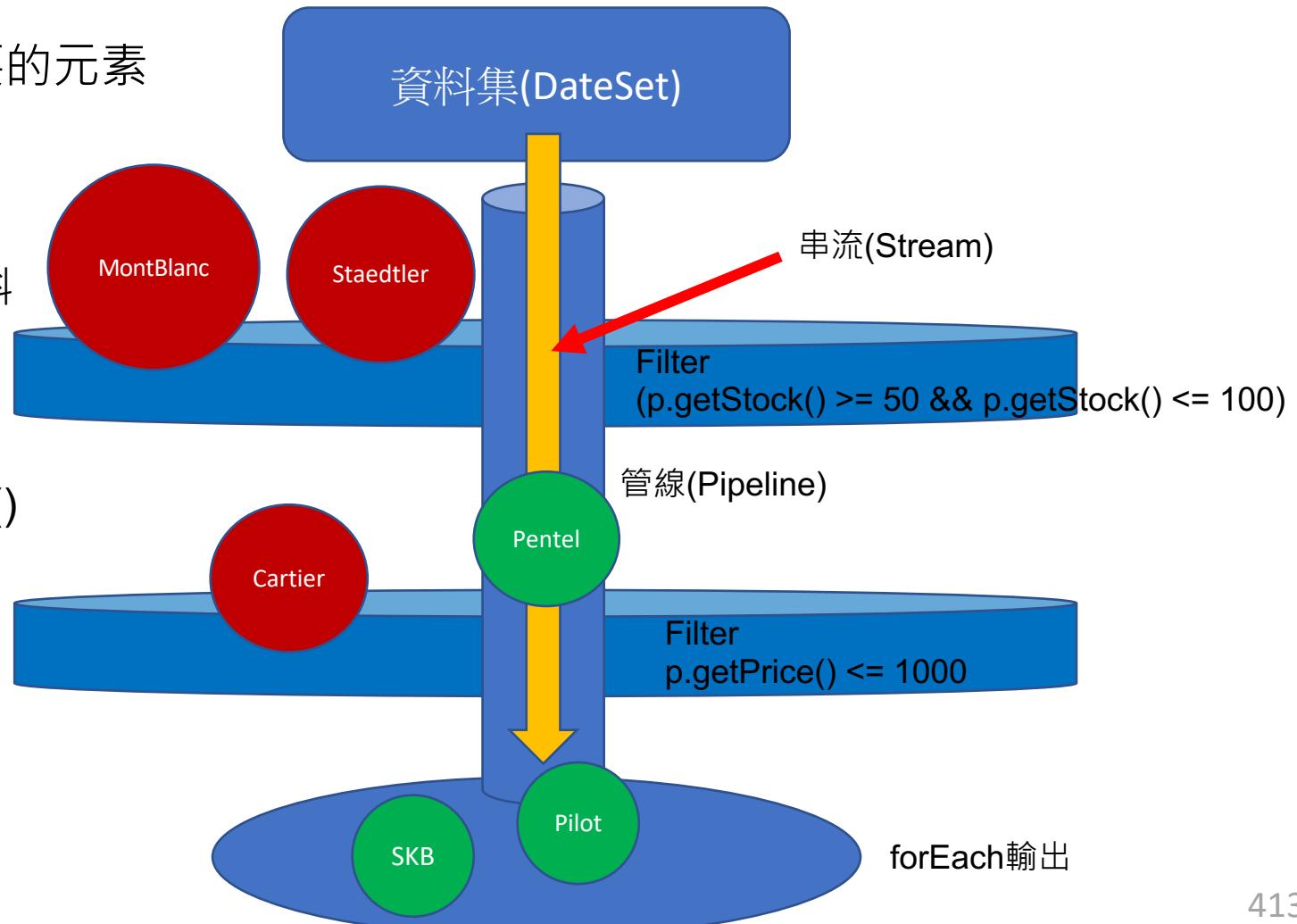
```
// 可考慮將複雜的邏輯轉換成以下method chaining方式，增加閱讀性
penList.stream()
    .filter(p -> p.getStock() >= 50 && p.getStock() <= 100)
    .filter(p -> p.getPrice() <= 1000)
    .forEach(Pen::printDetails);
```

- 可以進行邏輯合併增加閱讀性，由你決定！

串流中繼操作 (filter, map與peek)

範例：idv.david.stream套件

- Filter方法使用了Predicate lambda來濾出需要的元素
- map(Function<? super T, ? extends R> mapper)
 - 使用map方法取得經過運算或是操作後的轉換資料
 - 需傳入一個Function類型參數
 - 搭配基本型別的方法：
 - mapToInt(), mapToLong(), mapToDouble()
- peek(Consumer<? super T> action)
 - 可以在中間操作過程中，從串流裡取得資料
 - 最常拿來在操作串流過程中做為列印資料的處理



模組40

Java 8

Stream API (2)

40-1：
元素搜尋操作

40-2：
認識惰性求值

40-3：
Optional類別與null

搜尋結束方法

範例：[idv.david.stream套件](#)

- `findFirst()`
 - 回傳第一個符合條件的元素
 - 多搭配`filter`方法過濾出符合條件的元素
- `allMatch(Predicate<? super T> predicate)`
 - 回傳布林值，當所有的元素都符合條件時即為true，反之false
- `noneMatch(Predicate<? super T> predicate)`
 - 回傳布林值，當所有的元素都不符合條件時即為true，反之false
- 以上方法皆具有”短路運算”邏輯

搜尋結束方法 (續)

範例：[idv.david.stream套件](#)

- 不確定性(Nondeterministic)的搜尋操作
 - 對於不確定性搜尋操作，平行串流處理會更加有效率
 - 但結果可能會有很大的差異展現
- `findAny()`
 - 回傳第一個符合條件的元素
 - 在平行串流處理上，結果差異性可能會很明顯
- `anyMatch(Predicate<? super T> predicate)`
 - 回傳布林值，當任何一個元素有符合條件即回傳true，反之false
 - 在平行串流處理上，結果差異性可能會很明顯

惰性求值 (Lazy Evaluation)

- 惰性求值為程式語言理論中的一個概念，其最主要的目的就是要讓電腦的執行工作“最小化”
- 可分為以下兩種含意：
 - 延遲求值
 - 最小化求值
- 這邊我們主要探討的是最小化求值，因為延遲求值特性會是常見在函式程式語言(Functional Programming)裡，而Java語言並不是
- 說穿了，所謂最小化求值就是“短路運算”(Short-circuit)邏輯

Optional類別

範例：[TestOptional.java](#)

- **java.util.Optional<T>**
- Java Collection API與JSR166的參與者之一 Doug Lea說過：“Null sucks!”
(延伸: Tony Hoare的The Billion Dollar Mistake!)
- 我們吃 NullPointerException 吃到都要吐了 !!! 而搭配Optional使用可以避免NullPointerException再出現在你眼前囉 ^^
- 使用isPresent()檢查是否有取到值，有值的情況下會回傳true，再搭配get()方法取得
- 也可以使用orElse()，在取不到指定值的情況下，能有一個替代值

模組41

Java 8

Stream API (3)

41-1：
元素運算操作

41-2：
元素排序操作

41-3：
元素收集與整理操作

串流資料運算

範例：[idv.david.stream套件](#)

- `count()`
 - 回傳該串流的總元素數量
- `max(Comparator<? super T> comparator)`
 - 根據傳入的Comparator比較器的規則，回傳串流裡最大的元素
- `min(Comparator<? super T> comparator)`
 - 根據傳入的Comparator比較器的規則，回傳串流裡最小的元素

串流資料運算(續)

範例：[idv.david.stream](#)套件

- **average()**
 - 根據串流的資料進行平均運算，回傳一個Optional
 - 若是串流裡沒有資料，則回傳的Optional為empty
 - 從Optional裡取出的值為基本資料型別
- **sum()**
 - 根據串流的資料進行總合運算
 - 在IntStream, LongStream, DoubleStream裡都有定義sum()方法

串流資料排序

範例：[idv.david.stream套件](#)

- `sorted()`
 - 對還在串流裡的資料進行自然排序(natural ordering)
- `sorted(Comparator<? super T> comparator)`
 - 根據傳入的比較器內容進行排序
- `comparing(Function<? super T, ? extends U> keyExtractor)`
 - 可以搭配方法參考或是lambda對指定的欄位做排序
 - 支援基本型別Function使用
- `thenComparing(Comparator<? super T> comparator)`
 - 可再指定額外的欄位做排序
- `reverse()`
 - 排序結果為反轉

串流資料收集

範例：[idv.david.stream套件](#)

- `java.util.stream.Collectors`類別提供了對串流資料的收集操作，使用起來非常方便
- `collect(Collector<? super T, A, R> collector)`
 - 可以把串流處理後的結果再存到另一個新的資料結構裡
 - 需依靠`Collectors`類別完成操作
 - 例如：
 - `stream().collect(Collectors.toList());`
 - `stream().collect(Collectors.toMap());`

模組42

Java 8

新日期/時間API

- 42-1：
比較新舊API差異
- 42-2：
**LocalDate, LocalTime
類別介紹**
- 42-3：
**Instant, Duration,
Period類別介紹**

Date/Time API (JDK 8)

- 以往Java處理日期與時間的API：
 - 有sql.Date又有util.Date易混淆，而且util.Date竟然還包含了時間@#\$%^&!
 - Calendar的月份是0 – 11，你還得手動 + 1
 - 以上都是可變物件，non thread-safe
- JDK 8新增了**java.time**相關套件
 - 讓類別與方法的使用更加直覺
 - 讓新的API使用上能更加流暢使用 (就像Builder模式一樣)
 - **物件實體內容為不可變(Immutable)，對Lambda執行來說非常重要**
 - 因為實體為不可變，即為**thread-safe**
 - 使用了ISO標準定義了Date / Time
 - `toString`方法可以顯示讓人好閱讀理解的格式

Date/Time API (JDK 8)

- java.time套件裡的重要兩個類別 (**無關時區**)
- **LocalDate**
 - 不包含時間
 - 以年-月-日表示
 - `toString` (採ISO 8601格式 YYYY-MM-DD)
- **LocalTime**
 - 不包含日期
 - 保存了時、分、秒與奈秒
 - `toString` (HH:MM:SS)
- 也有**LocalDateTime**類別可以使用

範例：
`TestLocalDate.java`
`TestLocalTime.java`
`TestLocalDateTime.java`

Date/Time API常用方法

名稱	用法	用途
now	today = LocalDate.now()	根據系統時間建立一個LocalDate
of	meet = LocalTime.of(13, 30)	根據提供的參數建立一個LocalTime
get	today.get(DAY_OF_WEEK)	回傳參數指定的資訊
with	meet.withHour(12)	改變該物件的某個時間資料並回傳一個新的物件
plus, minus	nextWeek.plusDays(7) sooner.minusMinutes(30)	回傳一個經過增加或是減少後的LocalDate/Time物件
to	meet.toSecondOfDay()	轉換成另一型別資料回傳， 例如toSecondOfDay()即回傳一個int的資料
at	today.atTime(13, 30)	將LocalDate加上時間後，回傳一個LocalDateTime
until	today.until	計算兩個時間點相隔
isBefore, isAfter	today.isBefore(lastWeek)	比較兩個物件時間線為之前或之後
isLeapYear	today.isLeapYear()	檢查是否為閏年

Date/Time API

範例：
[TestInstantDurationPeriod.java](#)

- **Instant – 用來表示一個瞬間的時間點**
 - 可拿來當作時間戳記 (time stamp) 如使用者登入系統的時間
 - 常用來做時間的比較，如之後或是之後
- **Period – 用來表示一個日期週期**
 - 年、月、日的資料是根據ISO 8601曆法 (Gregorian)
 - 適合拿來進行日期的加減計算
- **Duration – 用來表示一個時間週期**
 - 最小單位可以紀錄到奈秒(nano seconds)
 - 秒數資料可用long資料型別做儲存
- **TemporalUnit – 用來表示各種時間單位**
 - 由ChronoUnit列舉(enum)所定義

Date/Time API

範例：
[TestDateTimeFormatter.java](#)

- `java.time.format`套件提供了更為方便的日期/時間格式化
- `DateTimeFormatter`類別裡面定義了許多格式化的常數，讓我們在進行格式化時，就不用像以往使用`SimpleDateFormat`的做法，要自行指定格式項目
- `ofPattern`方法如同`SimpleDateFormat`，也能讓開發者自行決定格式
- 另也有`FormatStyle`列舉，定義了`SHORT`, `MEDIUM`, `LONG`, `FULL`
- **API文件是您的好朋友 ☺**

額外補充附件

1. 名詞解釋
2. I/O相關
3. 集合相關
4. 執行緒相關
5. 標註(Annotation)
6. Javadoc說明文件
7. JAR檔介紹

名詞解釋

- 方法(Method)與函式(Function)：

- 宣告在類別裡面為方法，代表物件的行為/功能
- 函式則宣告在類別之外
- Java語言只有方法

- 參數(Parameter)與引數(Argument)：

- 宣告方法定義在小括號裡的項目為參數，代表此方法呼叫時所需要的資料
- 呼叫時傳入的資料即為引數

```
public void myMethod(int data){  
    .....  
}  
  
        ↓  
Parameter
```

```
xxx.myMethod(10);  
  
        ↓  
Argument
```

名詞解釋

- Field (值域) :

在類別裡宣告的資料成員都稱為field

- Attribute (屬性) :

Field的一種，可直接存取的資料

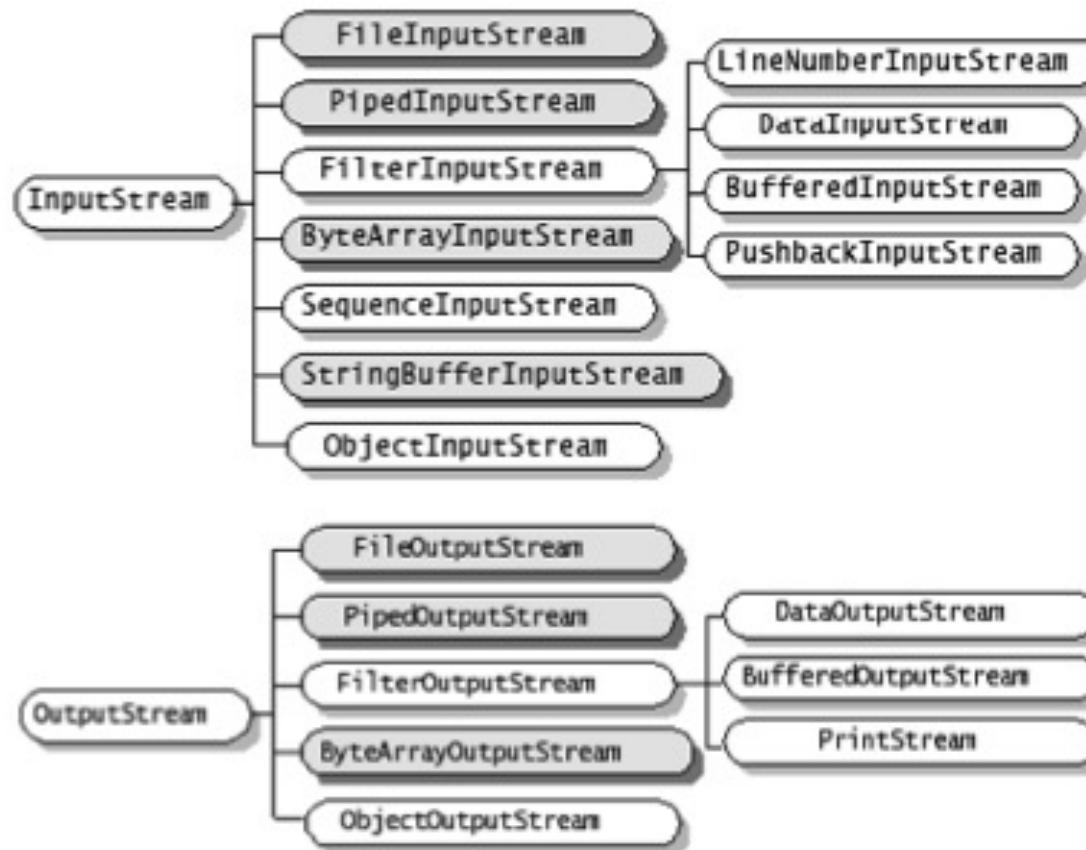
- Property (屬性) :

Field的一種，無法直接存取，但可透過getter/setter存取

```
public class MyData {  
    // Constant (You call call it field or not)  
    public static final double PI = 3.14;  
  
    // Attribute (You can access directly)  
    public String name;  
  
    // Property (You can access by getter/setter)  
    private int age;  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        this.age = age;  
    }  
  
    // all above are fields  
}
```

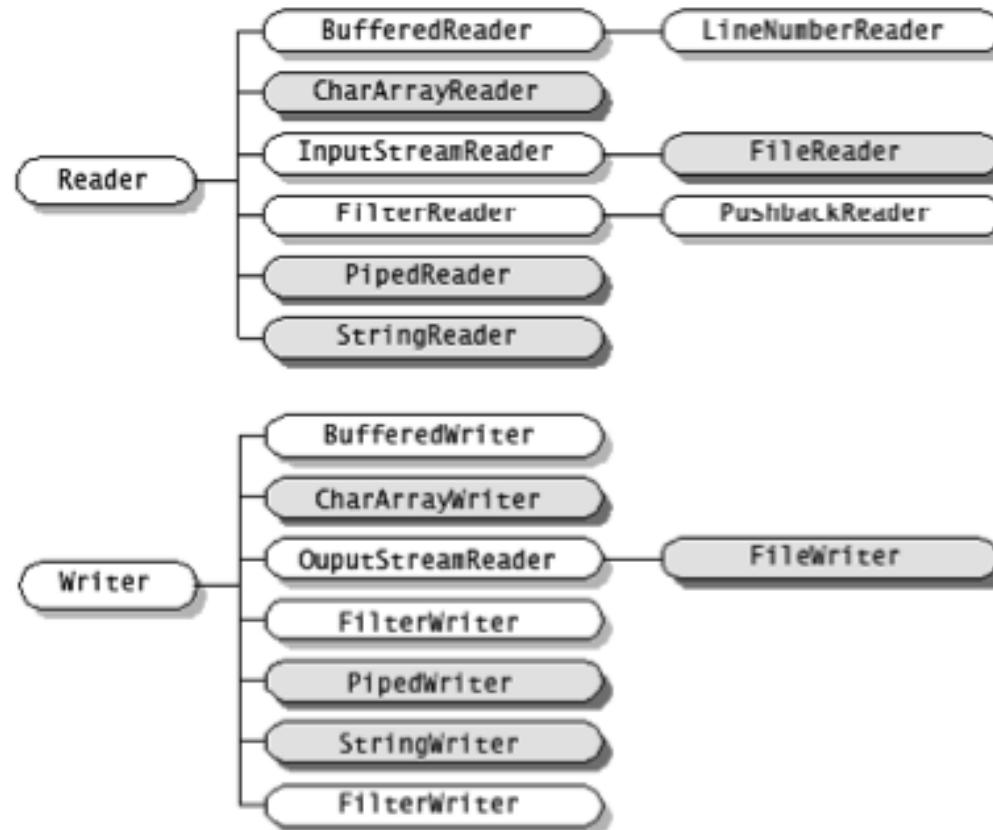
Java I/O API

- 負責位元資料的輸入 / 輸出工作
- 灰底者為 Data Sink



Java I/O API

- 負責字元資料的輸入 / 輸出工作
- 灰底者為Data Sink



Java節點資料流

Sink類型	字元資料流 Character Streams	位元資料流 Byte Streams	
記憶體 Memory	陣列 Array	CharArrayReader, CharArrayWriter	ByteArrayInputStream, ByteArrayOutputStream
	字串 String	StringReader, StringWriter	N/A
管線 Pipe	PipedReader, PipedWriter	PipedInputStream, PipedOutputStream	
檔案 File	FileReader, FileWriter	FileInputStream, FileOutputStream	

Java節點資料流

- 用來讀取與寫出記憶體內char陣列資料 (後者為byte陣列資料)
 - CharArrayReader與CharArrayWriter
 - ByteArrayInputStream與ByteArrayOutputStream
- 用來讀取與寫出記憶體內String物件內的字元
 - StringReader與StringWriter
- 管線資料流：提供有關執行緒的通訊功能
 - PipedReader與PipedWriter
 - PipedInputStream與PipedOutputStream
- 檔案資料流：用來存取檔案系統內容
 - FileReader與FileWriter
 - FileInputStream與FileOutputStream

Java處理資料流

處理動作 Process	字元資料流 Character Streams	位元資料流 Byte Streams
緩衝 Buffering	BufferedReader, BufferedWriter	BufferedInputStream, BufferedOutputStream
過濾 Filtering	FilterReader, FilterWriter	FilterInputStream, FilterOutputStream
Converting between Bytes and Characters	InputStreamReader, OutputStreamWriter	
串接 Concatenation		SequenceInputStream
物件序列化 Object Serialization		ObjectInputStream, ObjectOutputStream
資料轉換 Data Conversion		DataInputStream, DataOutputStream
計數 Counting	LineNumberReader	LineNumberInputStream
向前窺視 Peeking Ahead	PushbackReader	PushbackInputStream
列印 Printing	PrintWriter	PrintStream

Vector與Hashtable類別

- **Vector**為一可置入「任意物件」的「動態陣列」(可隨內含物自動增減長度)
- **Hashtable**內部是一個「類似表格」的資料結構來儲存資料
- Vector從JDK1.0時代就已存在，在JDK1.2時，新的List介面出現後，Vector才加入**List**家族，並實作新的方法
- Hashtable從JDK1.0時代就已存在，在JDK1.2時，新的Map介面出現後，Hashtable才加入**Map**家族，並實作新的方法
- Vector類別等同**ArrayList**類別，但**Vector**提供同步化的優點(與負擔)，這對多執行緒存取是很重要的
- Hashtable類別等同**HashMap**類別，但**Hashtable**提供同步化的優點(與負擔)，這對多執行緒存取是很重要的

範例：
`TestVector1.java`
`TestVector2.java`
`TestHashtable.java`

Enumeration介面

範例：
[TestVector_Enumeration.java](#)
[TestHashtable_Enumeration.java](#)

- JDK1.0的Enumeration介面類似JDK1.2的迭代器Iterator介面
- 介面方法
 - public boolean hasMoreElements()
 - public Object nextElement()
- 物件實作此介面目的是產生一序列的元素，透過呼叫nextElement()方法來取得一個一個接續的元素
- nextElement()所回傳的資料是Object的型態，使用者需要做Casting
- Hashtable與Vector都有提供方法將其資料置於Enumeration物件中，並透過Enumeration物件來存取所有的資料

背景執行緒 (Daemon)

- **Daemon執行緒**
 - Daemon執行緒的工作是等待別人要求服務，其run方法通常是一個無窮迴圈
 - 當其它所有執行緒都結束執行，只剩Daemon執行緒時，JVM便會結束Daemon執行緒的執行
 - 通常為系統程式，而非Daemon執行緒通常為應用程式，建議前者的優先權應比後者為低
 - Java的Garbage collector即是一種Daemon執行緒

執行緒池 (Thread Pool)

- 提供程式設計師更加彈性的產生與管理執行緒任務，其執行緒池功能讓我們能重複利用已產生的執行緒
- 任務可以是以下兩種：
 - `java.lang.Runnable`
 - `java.util.concurrent.Callable`
- 透過`Executors`取得`ExecutorService`的實體：
 - `Executors.newCachedThreadPool()`
 - 有需要就新增執行緒、可重複使用、閒置60秒後自動終止移出
 - 適用短暫的非同步任務
 - `Executors.newFixedThreadPool(int amount)`
 - 不能任意新增執行緒、可重複使用、沒有閒置機制
 - 適用於伺服器常駐任務

Callable / Future介面

範例：[TestCallable.java](#)

- 實作Callable介面如同Runnable一樣，差別在於：
 - 需改寫call()方法，而非run()方法
 - call()方法執行完畢後有回傳值(泛型)，run()方法為void
 - 可拋出受檢例外(Checked Exception)
- Future介面為用來取得Callable介面實作的 V call()方法回傳值，常被拿來跟Callable搭配使用
- 因為使用future.get()或future.get(long timeout, TimeUnit unit)取得回傳值可能會有阻斷的情形發生，所以使用後者時，傳入的指定時間還沒有結果產生，會丟java.util.concurrent.TimeoutException例外出來；也可以用isDone()來確認是否已產生結果

標註 (Annotation) JDK 5

- Metadata與Annotation
 - metadata簡單的說法就是資料的資料(Data about data)，JDK1.5中對metadata的支援就是Annotation，Annotation字面上的意思就是「標註」，目的在對程式碼作出說明以利分析工具使用
 - Annotation對程式運行沒有影響，它的目的在對編譯器或分析工具說明程式的某些資訊，我們可以在 package、class、method、field等處加上Annotation
- Annotation入門例：
 - 限定Override父類別方法 → @Override
 - 標示方法為Deprecated → @Deprecated
 - 隱匿編譯器的警告訊息 → @SuppressWarnings
 - 自訂Annotation型態

製作Java說明文件 - javadoc

- 自動產生HTML的說明文件
- 在類別名稱、實體變數、建構子與成員方法前加上 */**...*/* 的註解
- 在命令列透過 Javadoc XXX.java 的指令即可產生說明文件 (如同Java API文件的格式)
- Java即會自動產生XXX.html的檔案與其相關的html檔
- 只有**非private的成員**才會在文件上顯示資訊

Java壓縮檔 - JAR

- JAR (Java ARchive) 檔案為zip檔案格式
- 一個JAR檔可包含類別檔、聲音檔、影像檔...等需要下載到用戶端的各種資源
- JAR檔提供的相關好處：
 - 數位簽章與驗證機制增強安全性
 - 減少下載時間
 - 壓縮
 - 可以攜帶

JAR檔操作

- 產生JAR檔基本指令：**jar cf jar-file input-file(s)**
 - c : 要產生一個JAR檔
 - f : 將結果輸出至檔案而非螢幕
 - jar-file : JAR檔的名稱
 - input-file(s) : 說明要加入到JAR檔裡的各式檔案，可用空白區隔或 * 符號，能包含目錄、目錄的內容會以遞迴方式加入至JAR檔裡
- 其它額外選項：
 - v : 將加入JAR檔的過程顯示在螢幕上
 - 0(zero) : 不要壓縮

JAR 檔操作 (續)

- 檢視 JAR 檔基本指令：**jar tf jar-file**
 - t : 要檢視一個 JAR 檔
 - f : 將結果輸出至螢幕
 - jar-file : 想要檢視 JAR 檔的名稱
- 解開 JAR 檔基本指令：**jar xf jar-file [archived-file(s)]**
 - x : 要解開一個 JAR 檔
 - f : 將解開結果輸出至檔案
 - jar-file : 想要解開的 JAR 檔名
 - archived-file(s) : 說明要解開的檔名，可用空白隔開，若不提供代表解開全部檔案

附件

Eclipse工具操作

Eclipse簡介

- Eclipse簡述
 - Eclipse最早是由Object Technologies International這家公司開發，在1996年被IBM併購後，IBM將Eclipse公開捐出，成為一般所謂的Open Source Software，並成立了Eclipse.org，負責相關的開發工作
 - Eclipse設計美妙之處在於所有東西都是外掛，除了底層核心外，這樣的設計讓Eclipse具備強大擴充性
- Eclipse版本
 - Eclipse除了可開發Java之外，另有支援許多程式開發，如C/C++的CDT，PHP的PDT，當然也有Android的ADT可使用
 - 建議下載Eclipse EE版本，功能齊全

Eclipse簡介

- Eclipse下載
 - 官方下載網址：<http://www.eclipse.org/downloads/>
- Eclipse安裝
 - 系統環境變數設定JAVA_HOME，如C:\jdk-17.0.1
 - 將eclipse-jee-luna-R-win32(-x86_64).zip解壓縮到硬碟根目錄即可。解壓縮後會有一個eclipse目錄，裡面的eclipse.exe就是主執行檔
- Workspace目錄設定
 - 第一次執行前，應先在任意位置建立一個資料夾，所有的工作都會存在此目錄，若要備份工作目錄，只要備份此資料夾即可，若要升級新版Eclipse，只要將此目錄拷貝過去即可

P.S. **Eclipse位元版本需配合JDK位元版本下載**

如：使用JDK 64位元，則Eclipse也需使用64位元，否則會造成啟動失敗

Eclipse簡介

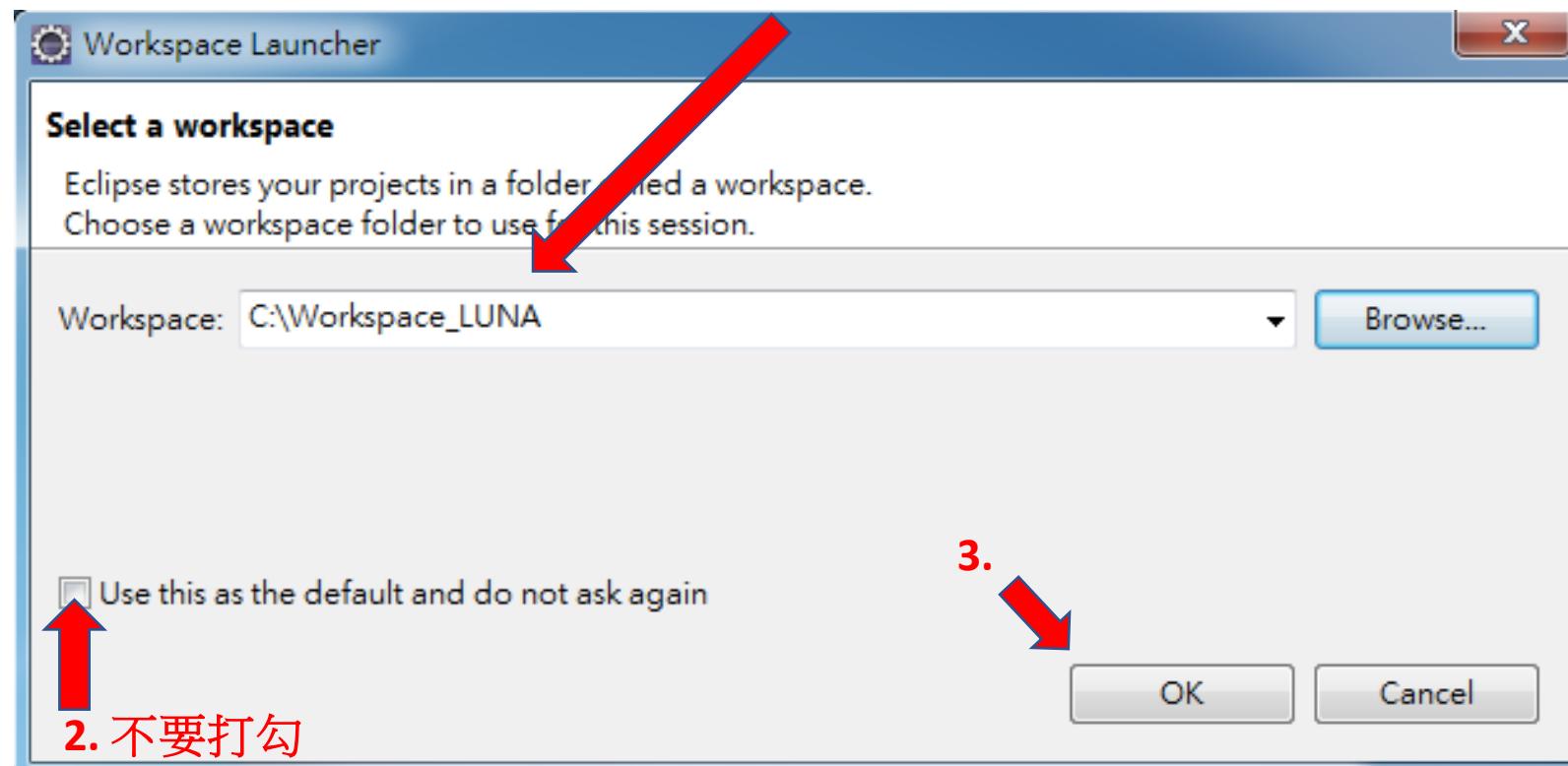
- Eclipse的起始畫面 (以LUNA版為例)



Eclipse簡介

- 選擇預先建立好的Workspace資料夾

1. 選擇預先建立好的資料夾

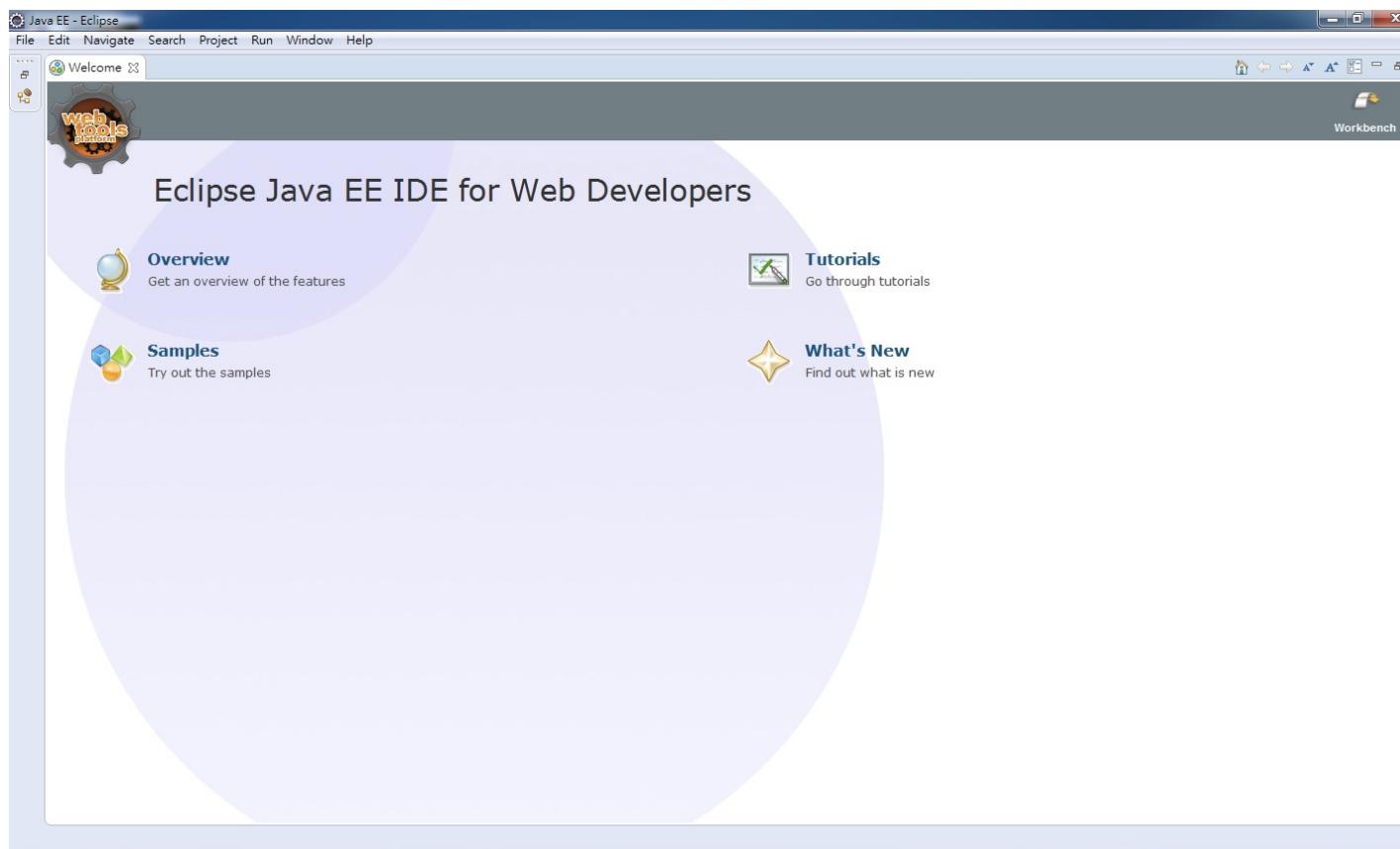


2. 不要打勾

3.

Eclipse簡介

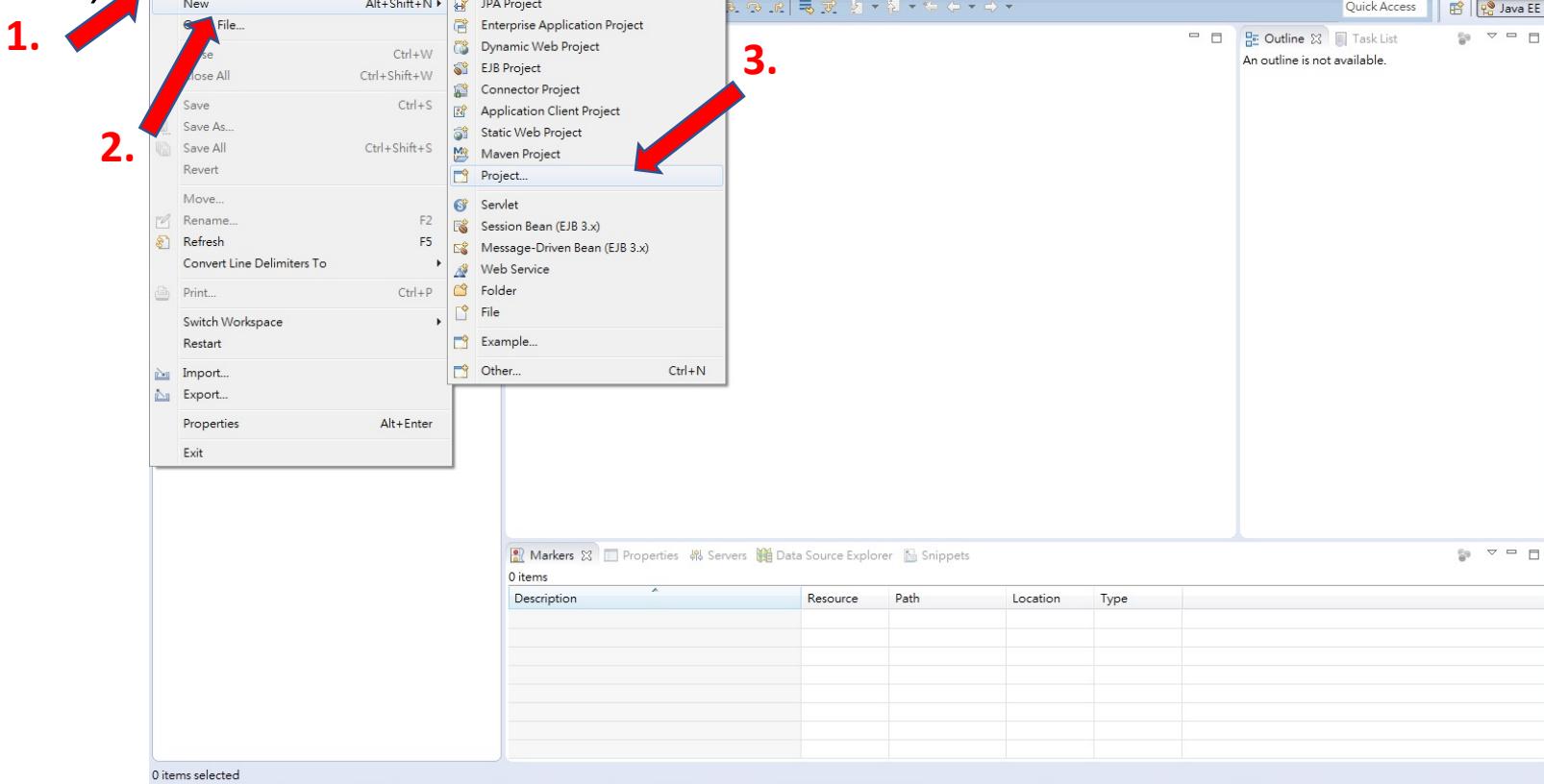
- Eclipse的歡迎畫面



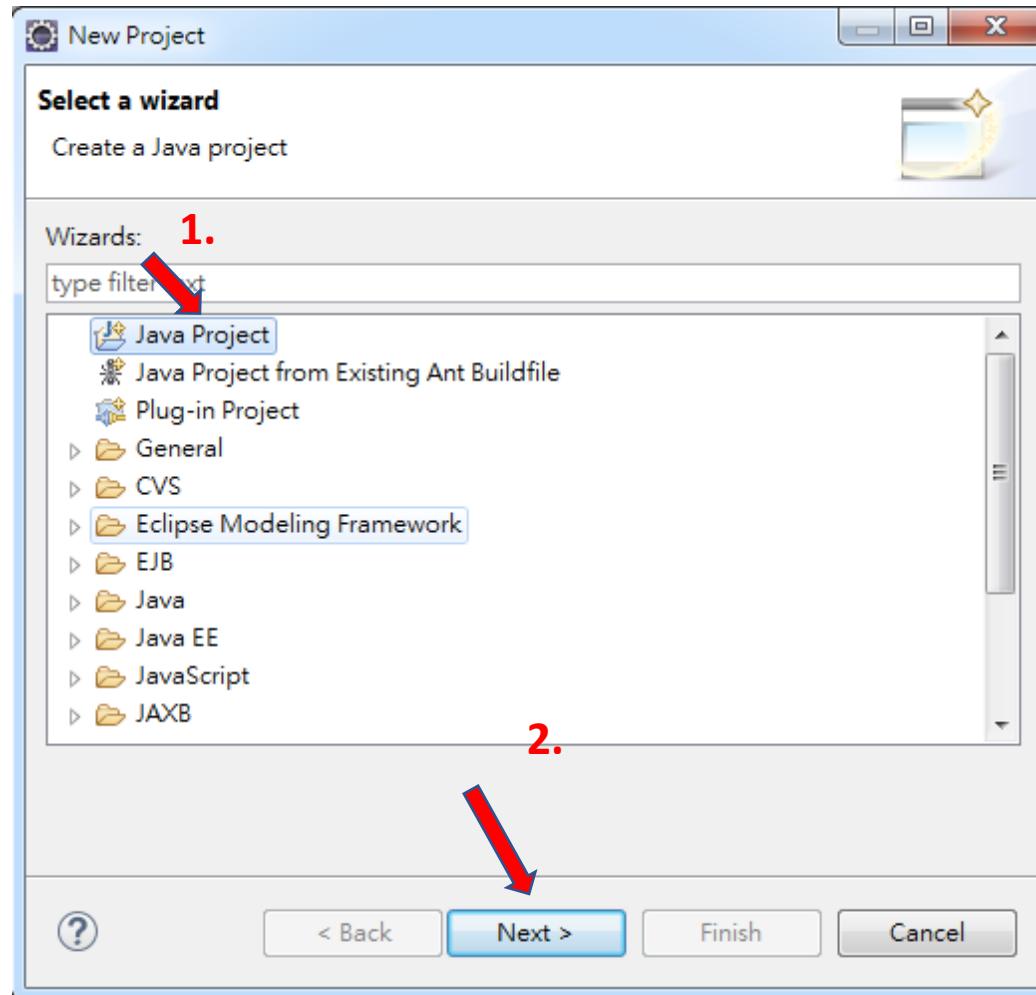
建立Java專案

- 每個以Eclipse開發的檔案，都必須隸屬於某個專案之中，因此得先建立一個Java專案

(File → New → Project...)

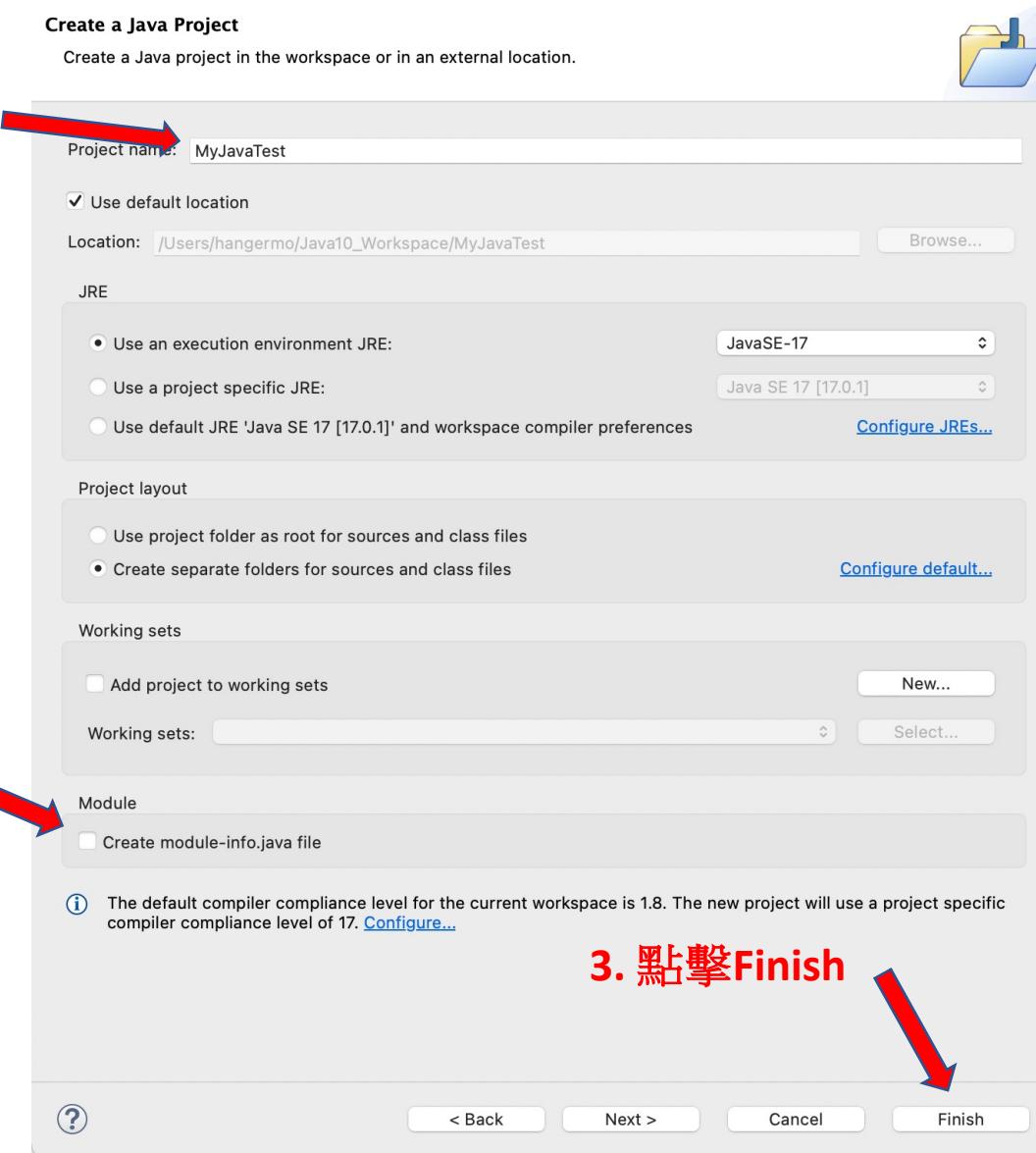


建立Java專案



建立Java專案

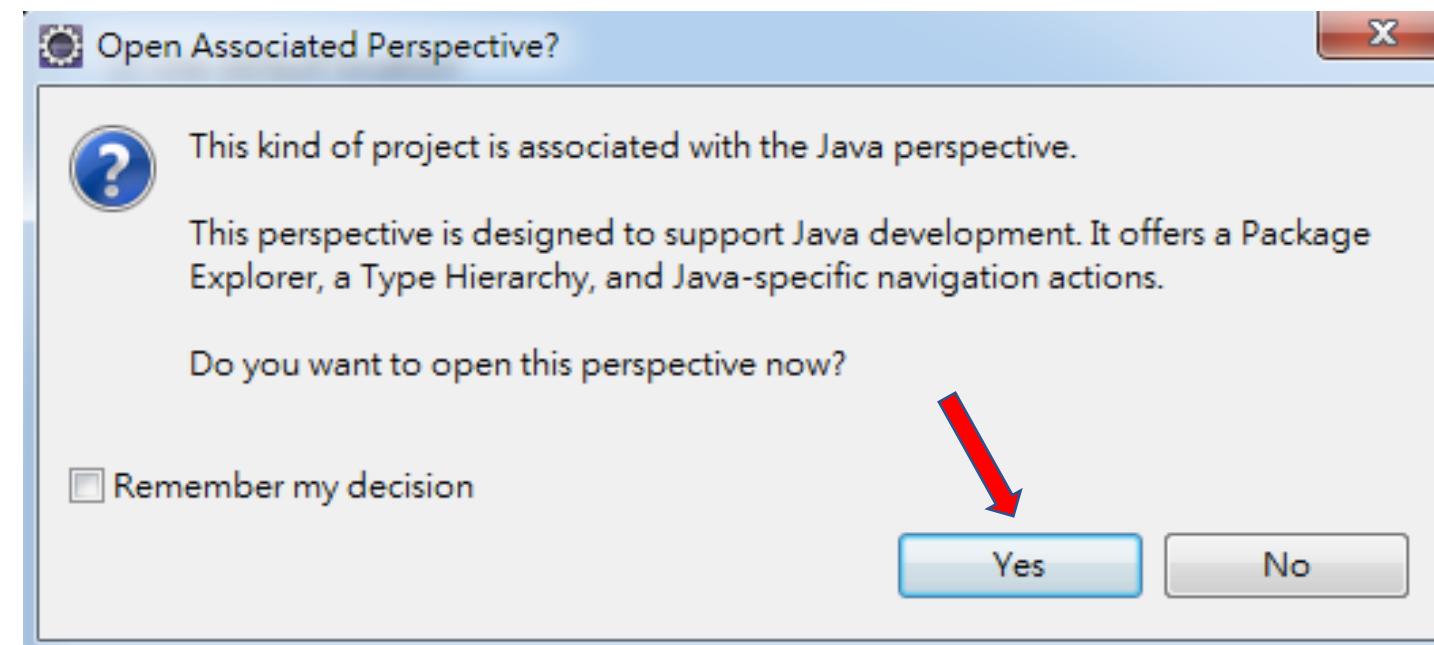
1. 輸入專案名稱



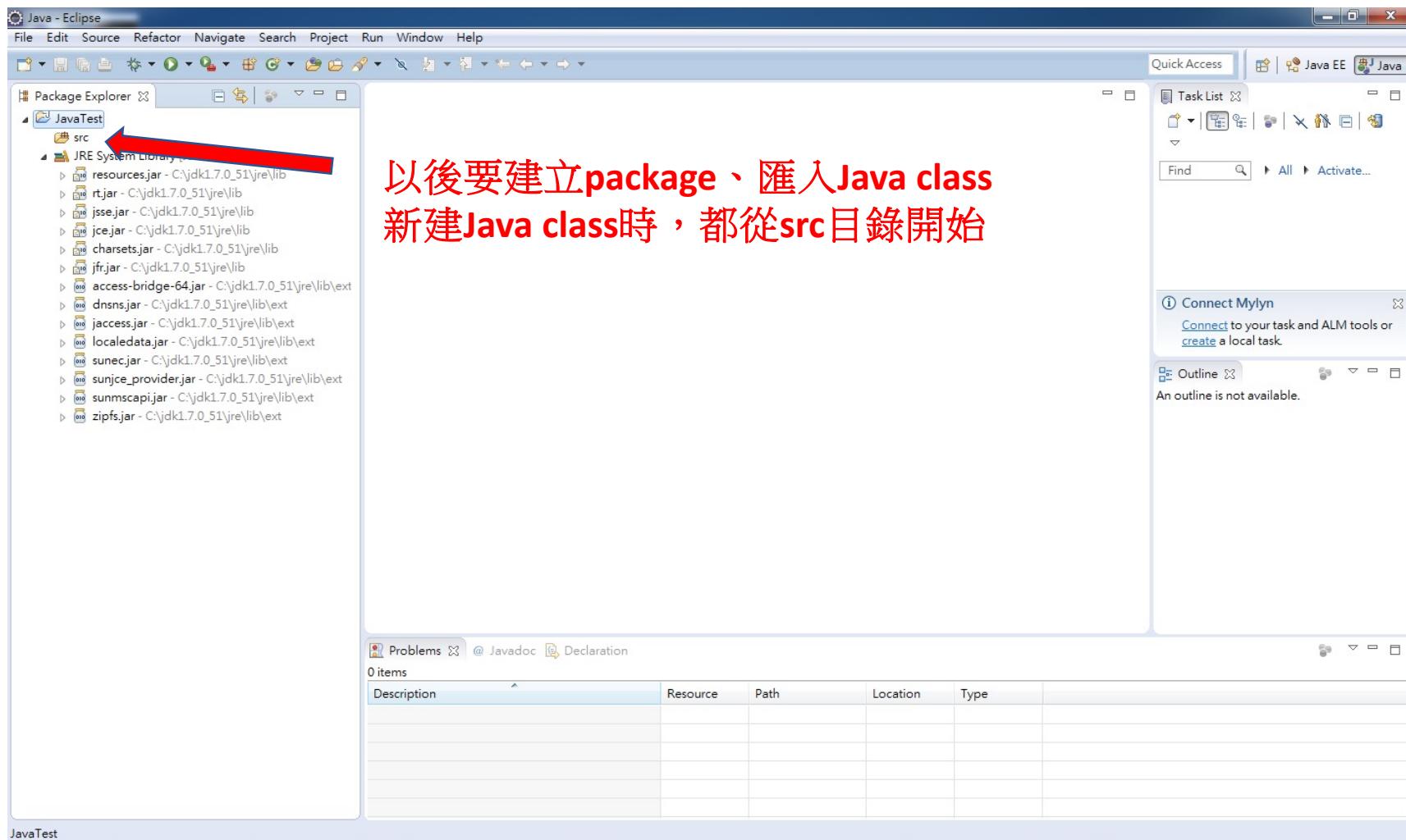
2. 取消勾選 (沒用到)

3. 點擊Finish

建立Java專案

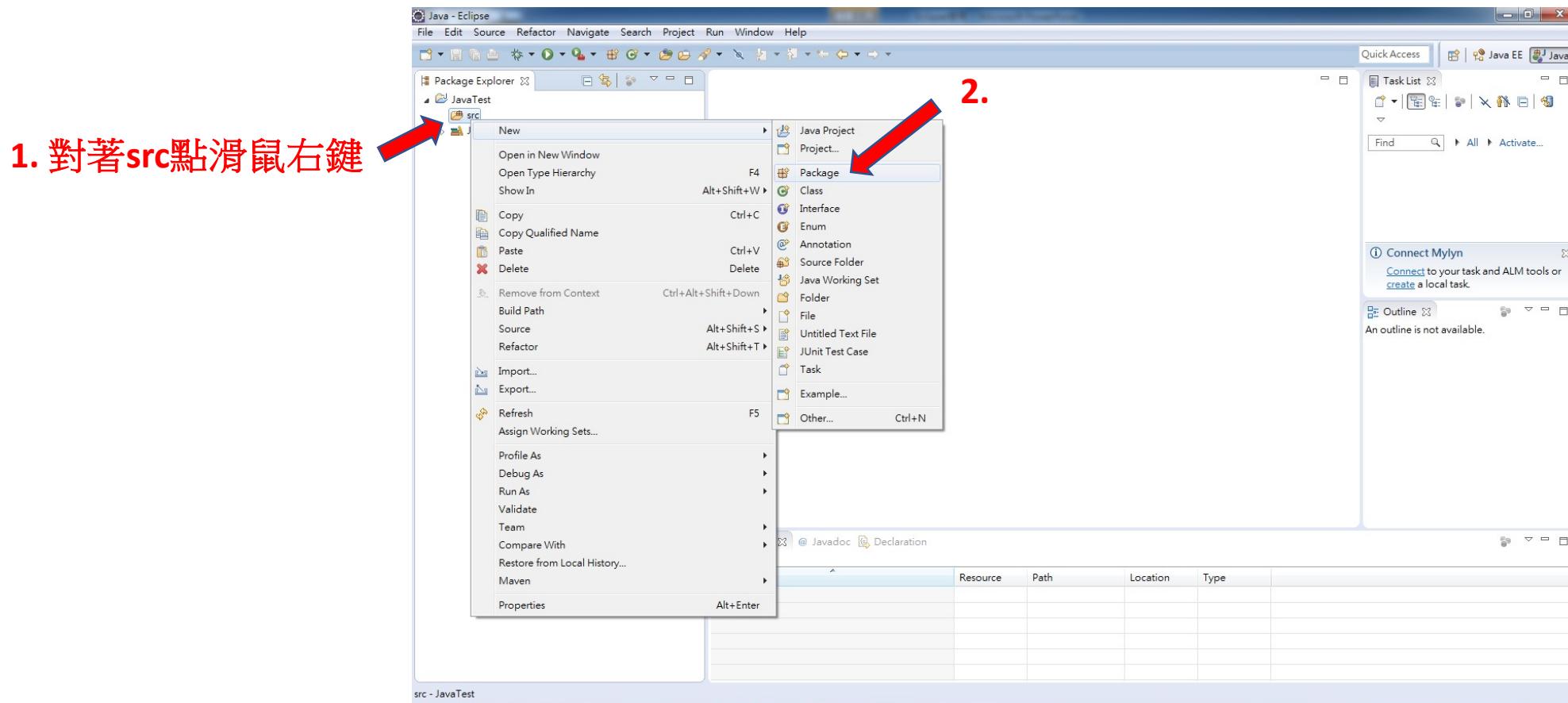


建立Java專案

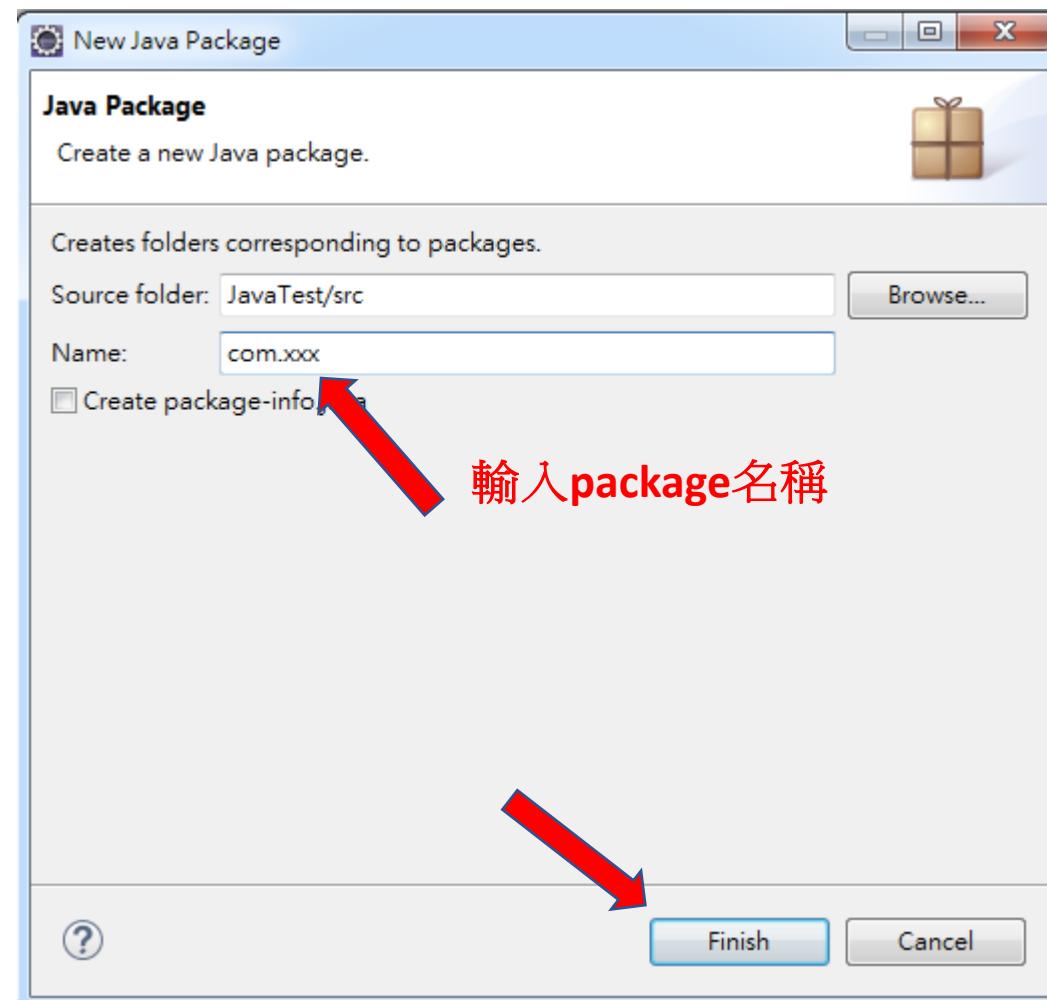


建立package / Java class

- 建立package (New → Package)

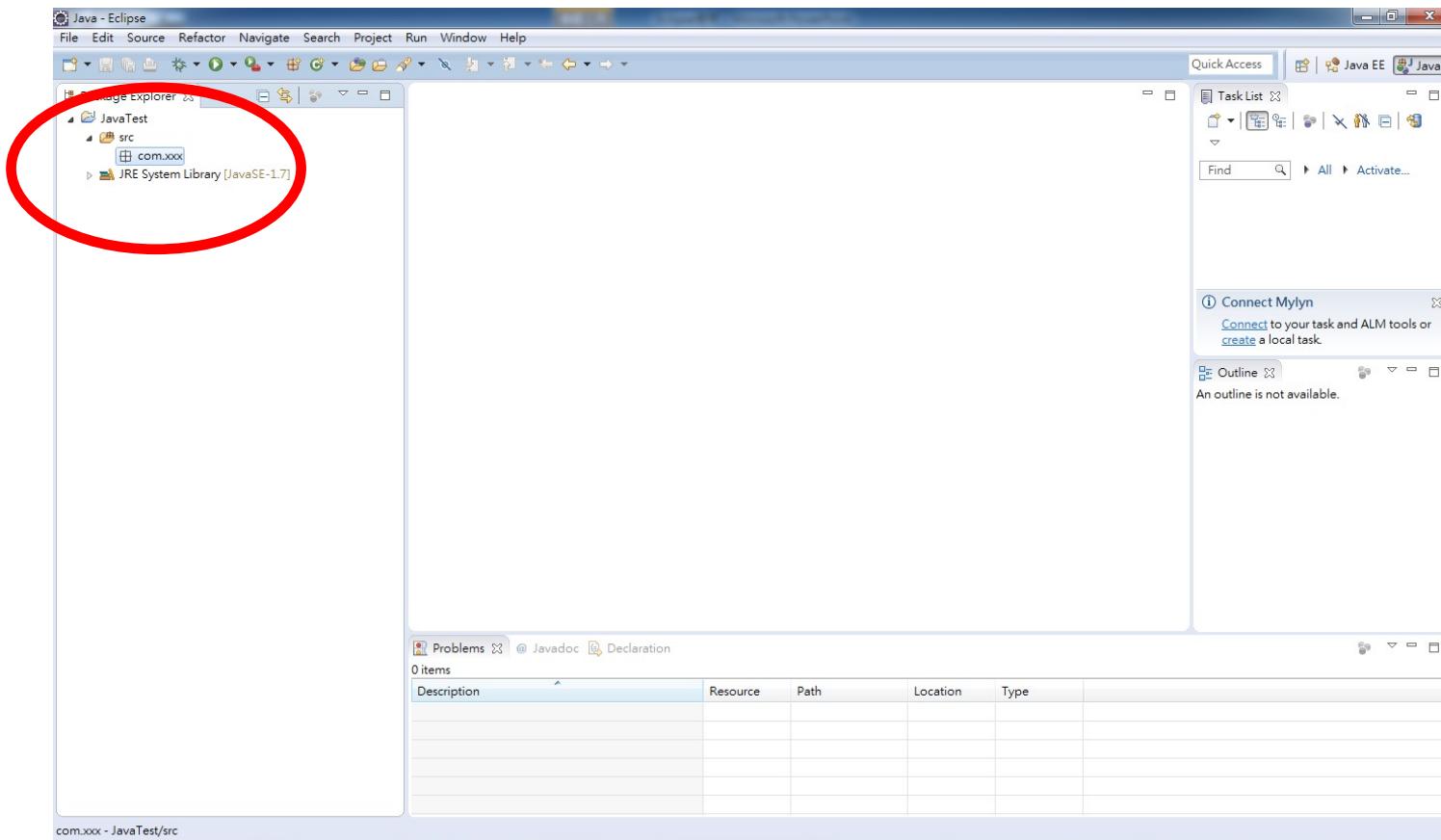


建立package / Java class



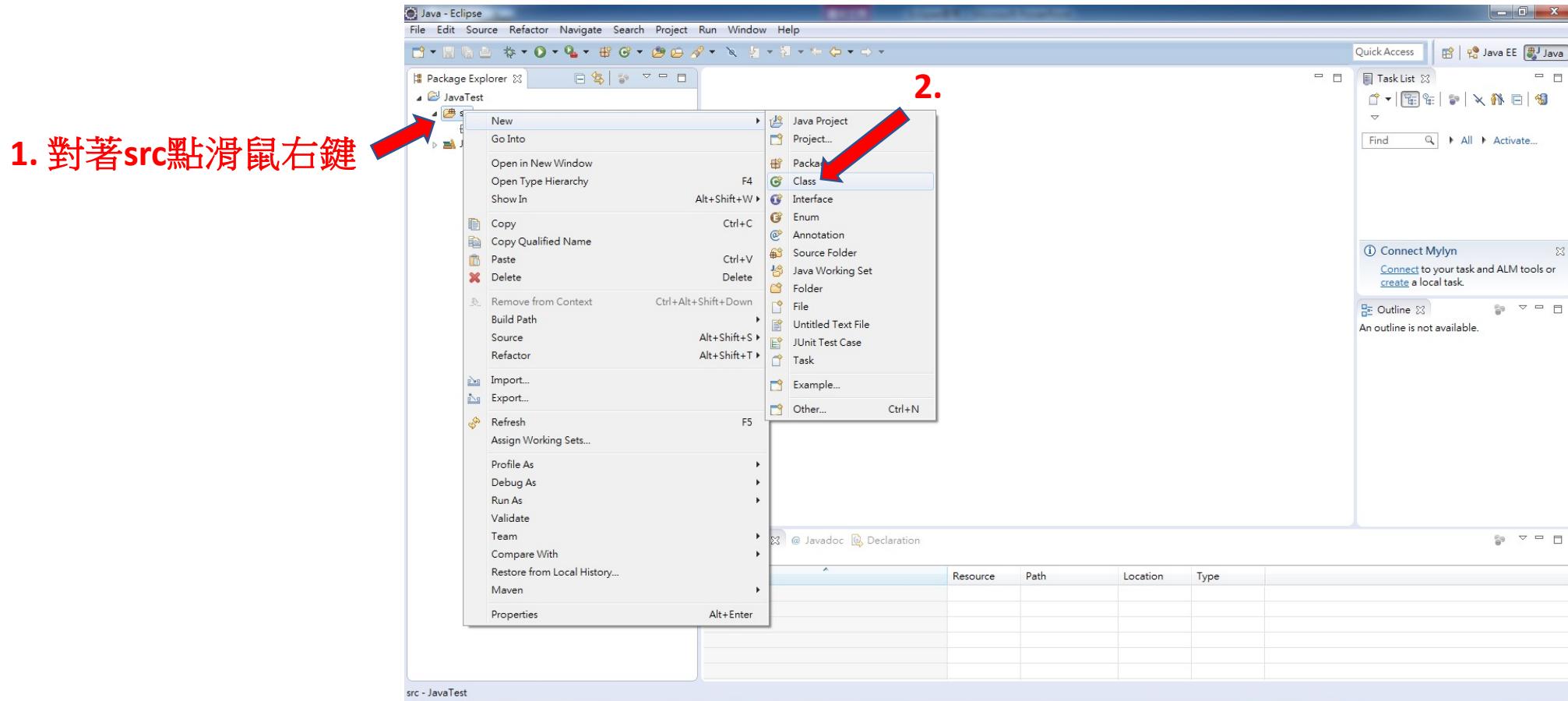
建立package / Java class

- 建立package完成畫面

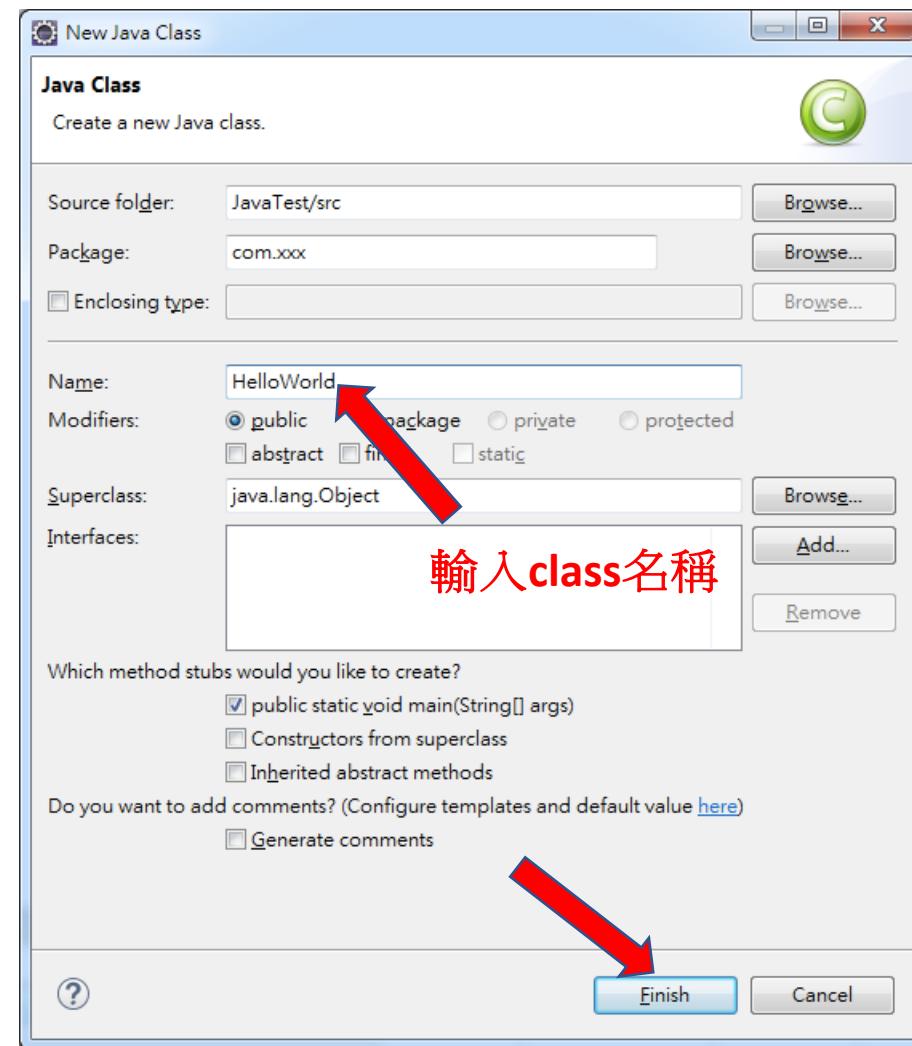


建立package / Java class

- 建立Java class (New → Class)

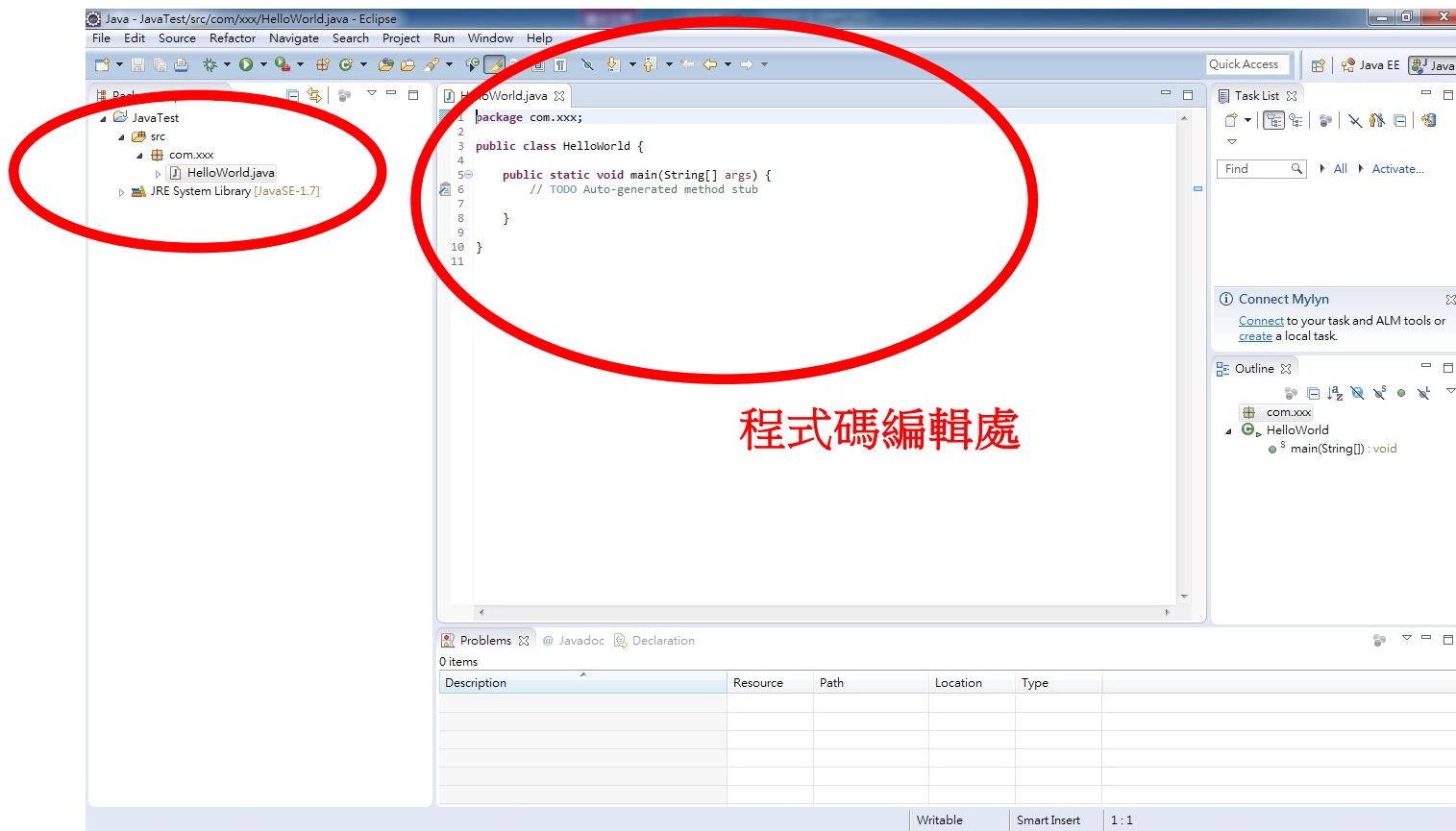


建立package / Java class



建立package / Java class

- 建立Java class完成畫面

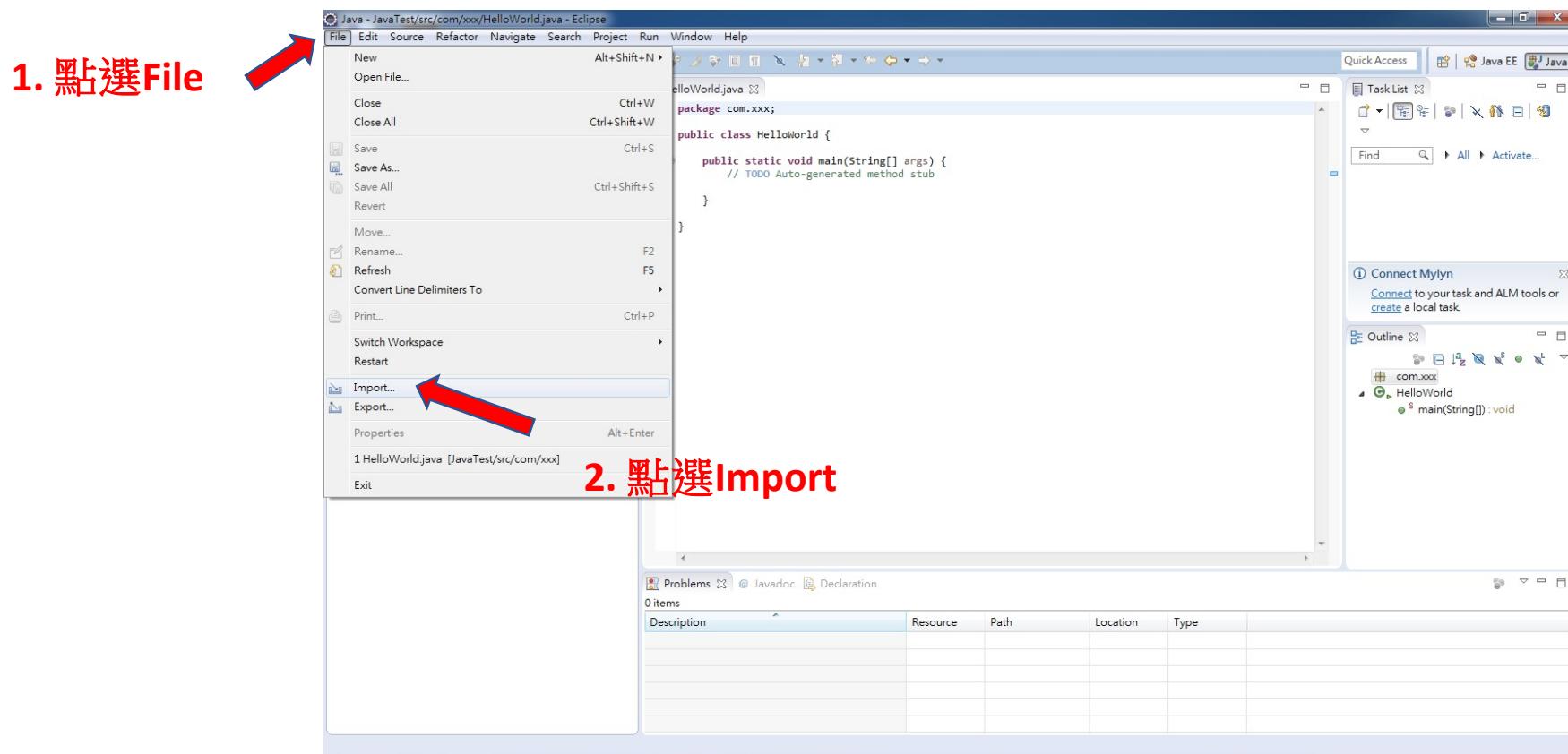


匯入Java class / Java專案

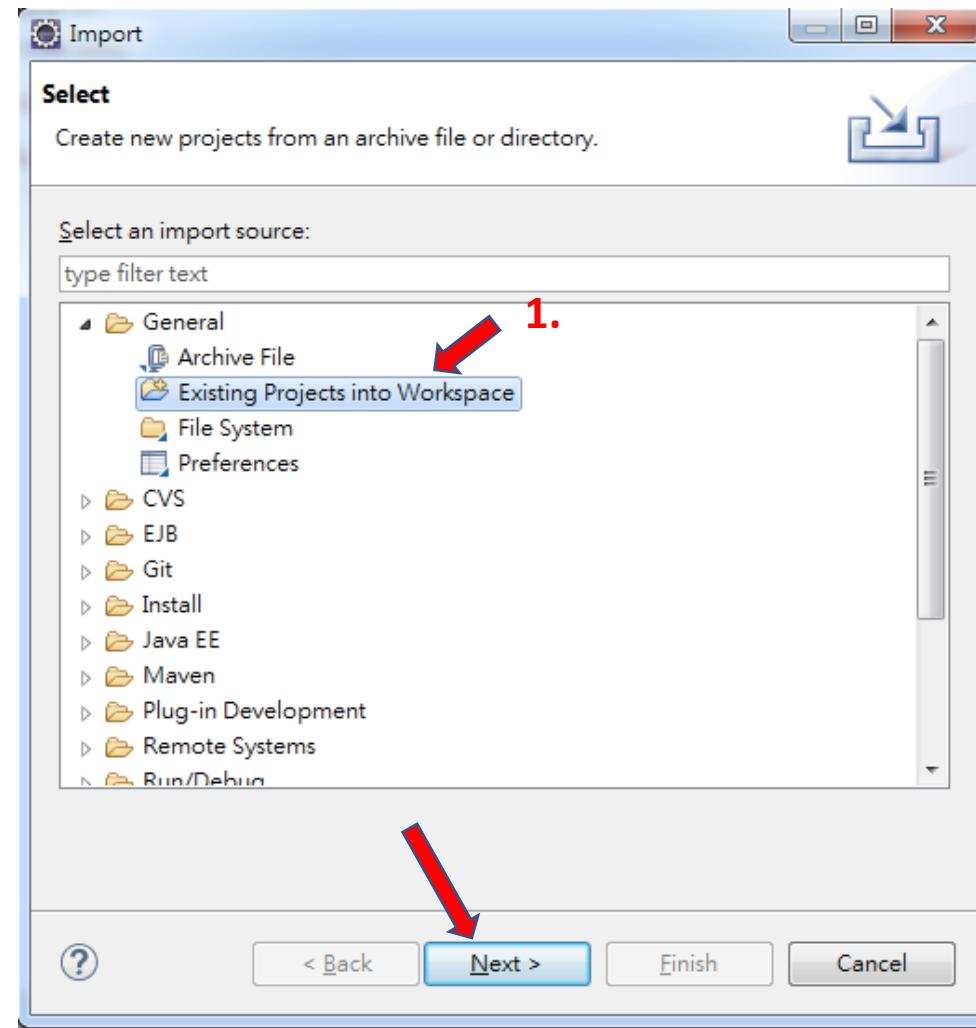
- 要匯入現有的Java class只要直接使用拖曳(或複製 – 貼上)的方式即可
 - 要匯入的Java class如果沒有套件：
 - 直接用拖曳(或複製 – 貼上)的方式放到src目錄，在src底下會自動出現default-package
 - 要匯入的Java class如果有套件：
 - 直接用拖曳(或複製 – 貼上)的方式放到預先建立好的套件目錄(如com.xxx)即可

匯入Java class / Java專案

- 匯入Java專案 (以本課程使用範例JavaEx_Part1.zip進行匯入操作說明)

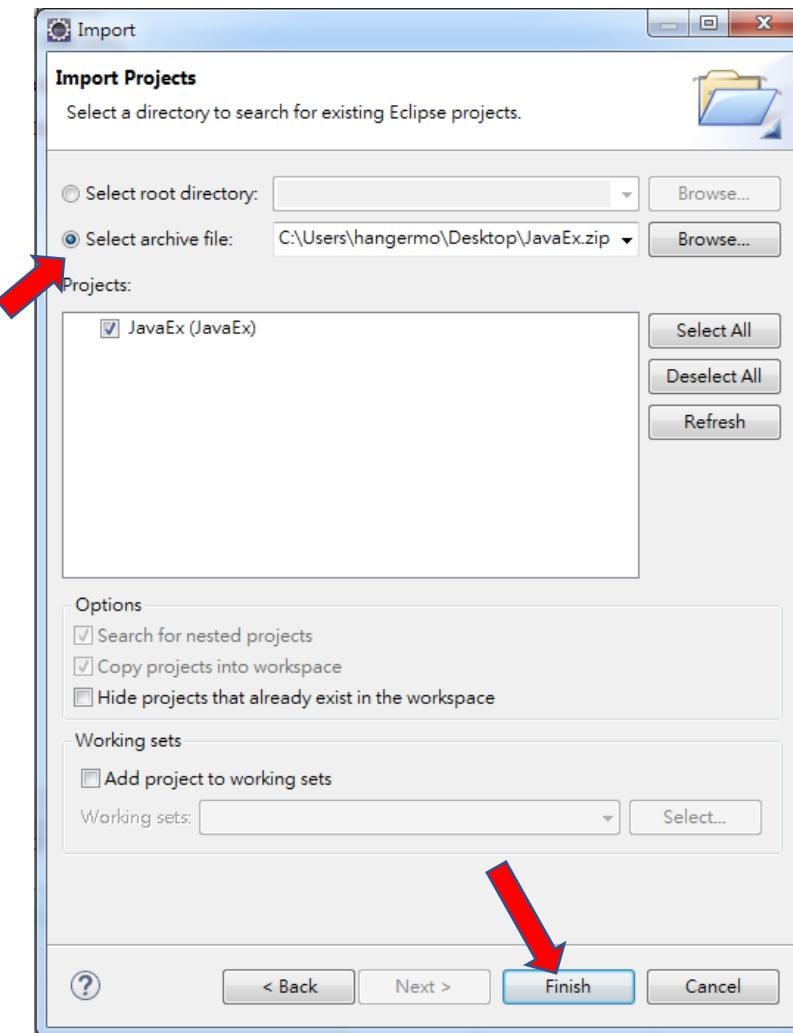


匯入Java class / Java專案



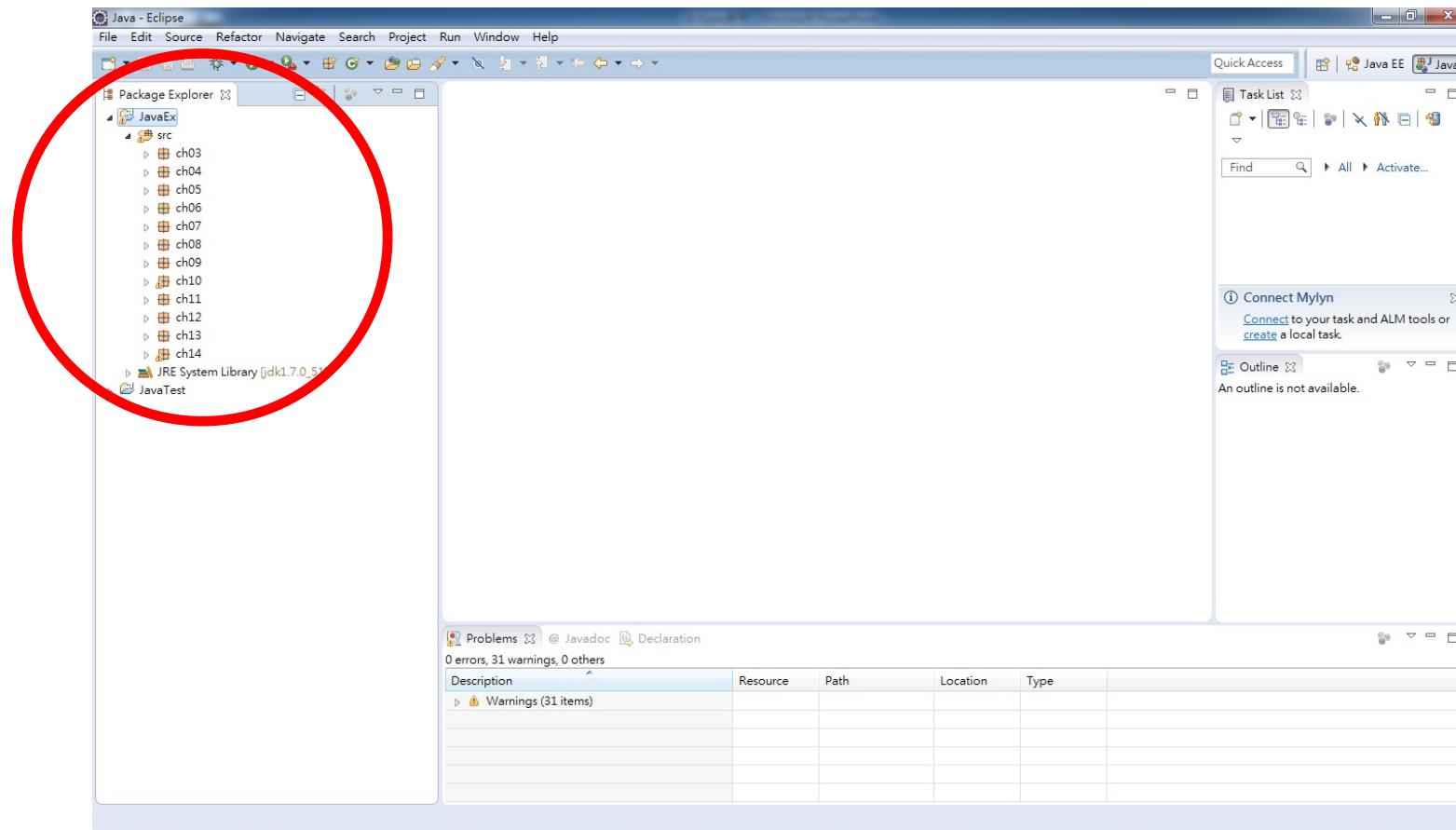
匯入Java class / Java專案

Browser找到要匯入的壓縮檔

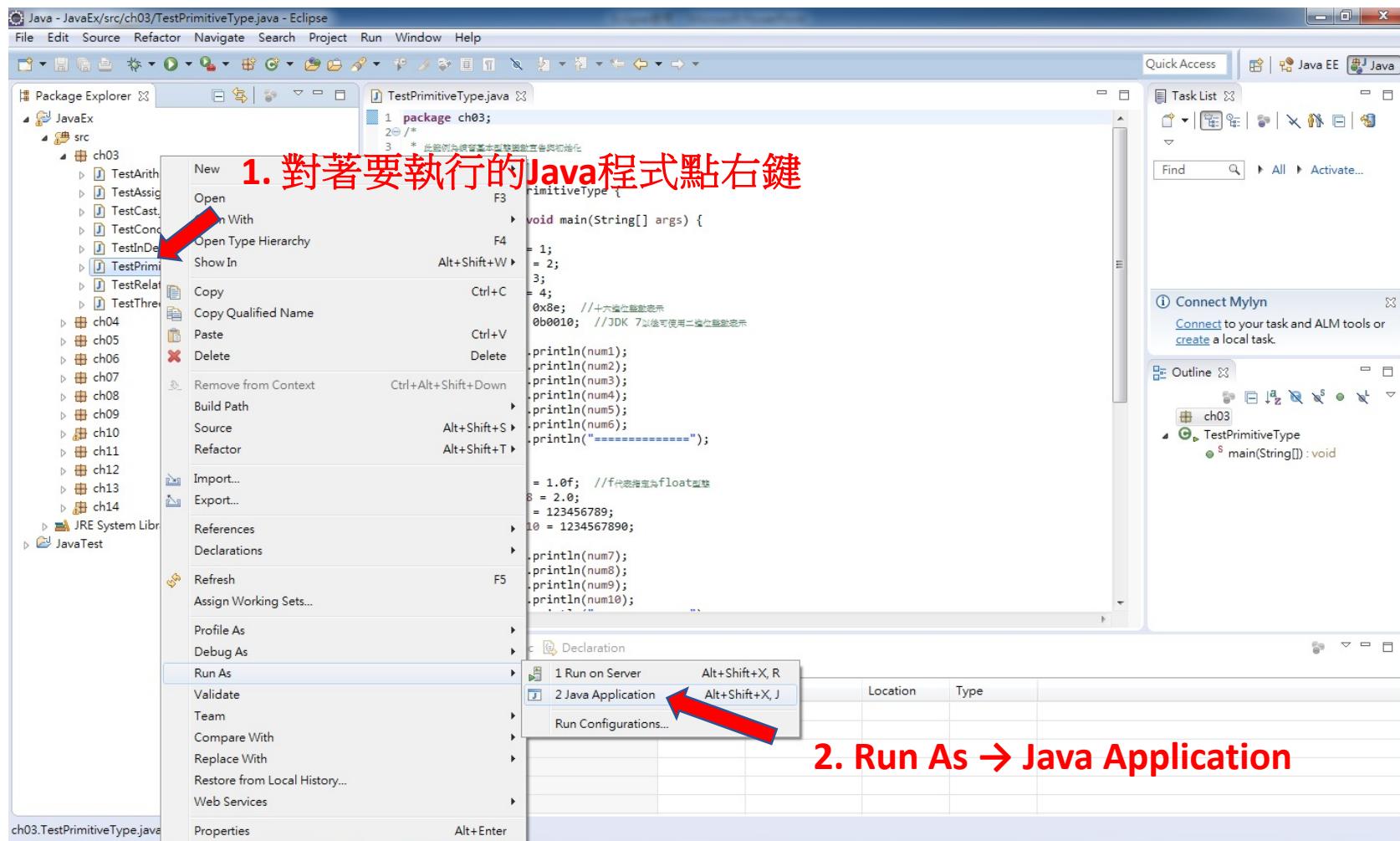


匯入Java class / Java專案

- 匯入Java專案完成畫面



執行Java程式



執行Java程式

The screenshot shows the Eclipse IDE interface with the following details:

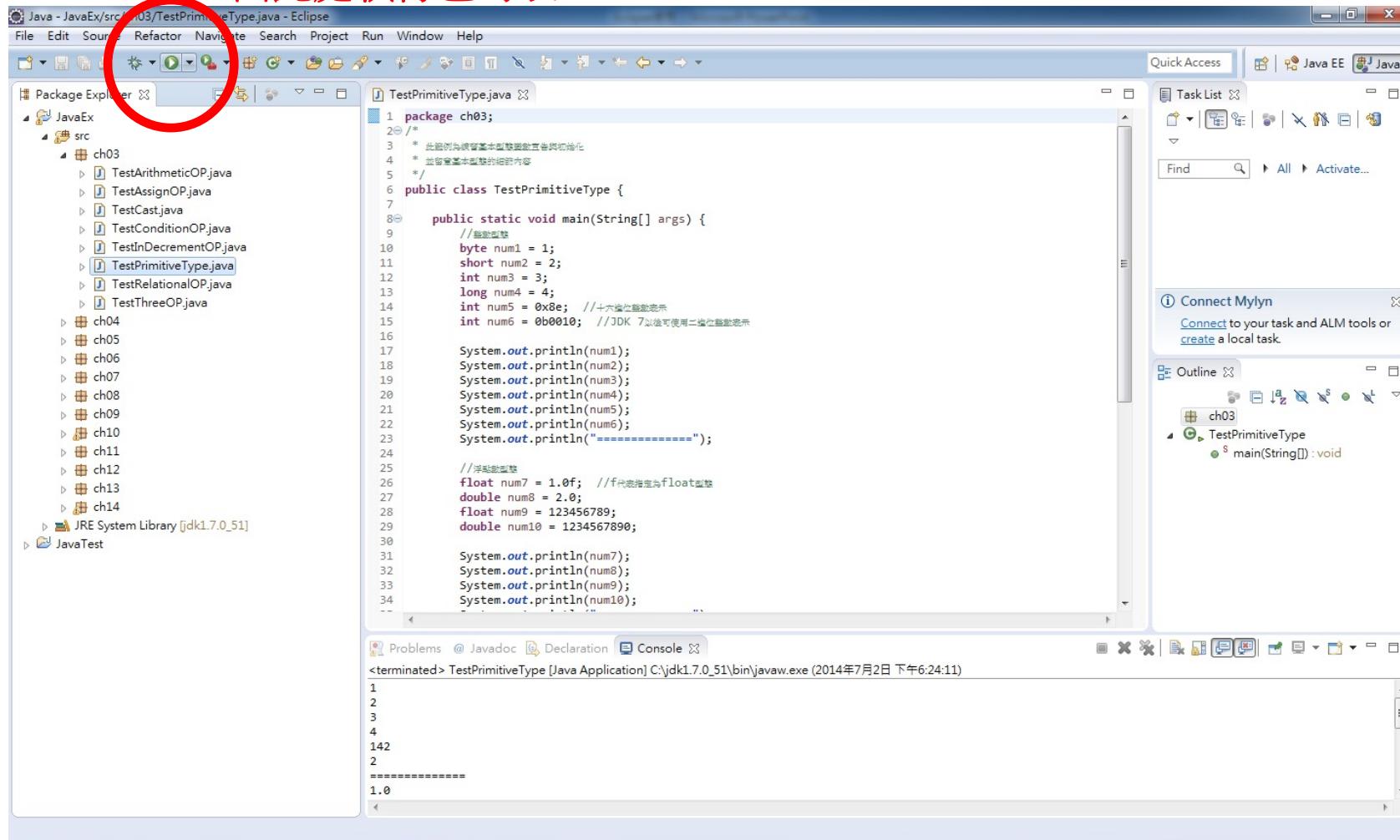
- Project Explorer:** Shows the project structure under "JavaEx". The file "TestPrimitiveType.java" is selected.
- Code Editor:** Displays the code for "TestPrimitiveType.java". The code prints various primitive data types to the console.
- Console View:** Shows the execution results:

```
<terminated> TestPrimitiveType [Java Application] C:\jdk1.7.0_51\bin\javaw.exe (2014年7月2日 下午6:24:11)
1
2
3
4
142
2
=====
1.0
```
- Red Circle:** A red circle highlights the output in the Console view, specifically the numbers 1, 2, 3, 4, 142, 2, and the decimal value 1.0.
- Annotations:** The text "執行結果" (Execution Result) is overlaid in red at the bottom right of the console output.

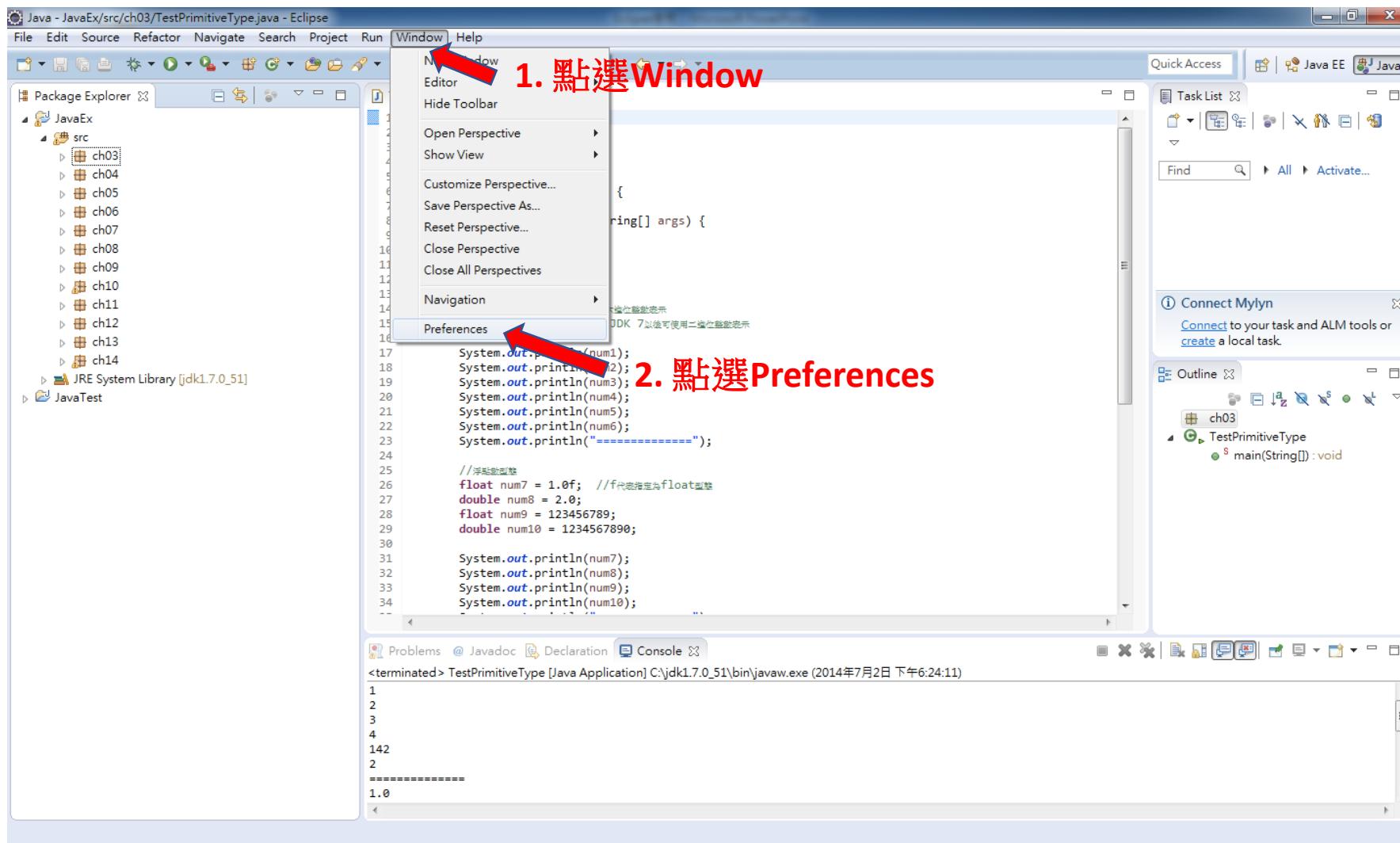
執行結果

執行Java程式

由此處執行也可以

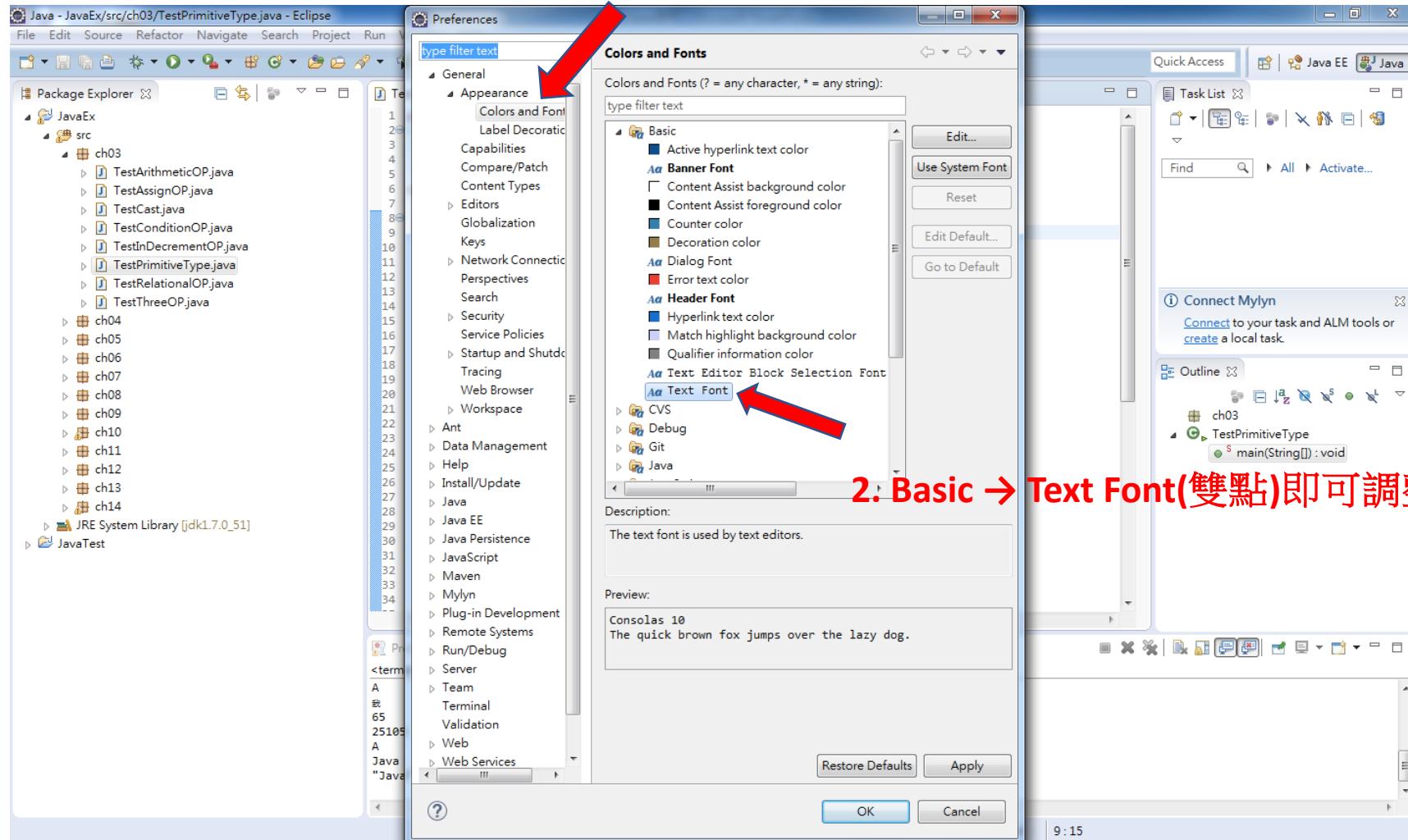


偏好設定 – 字型大小



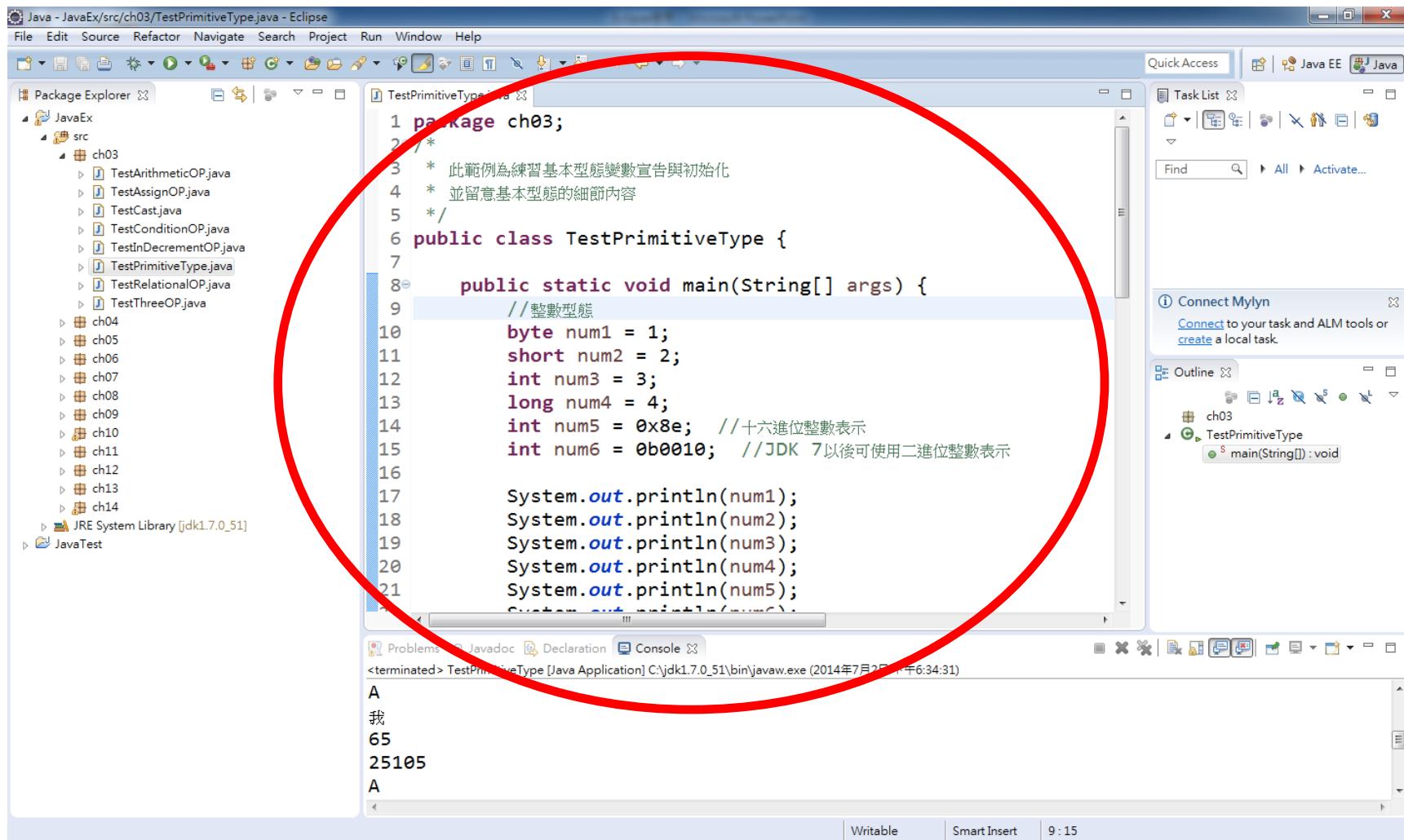
偏好設定 – 字型大小

1. General → Appearance → Colors and Fonts

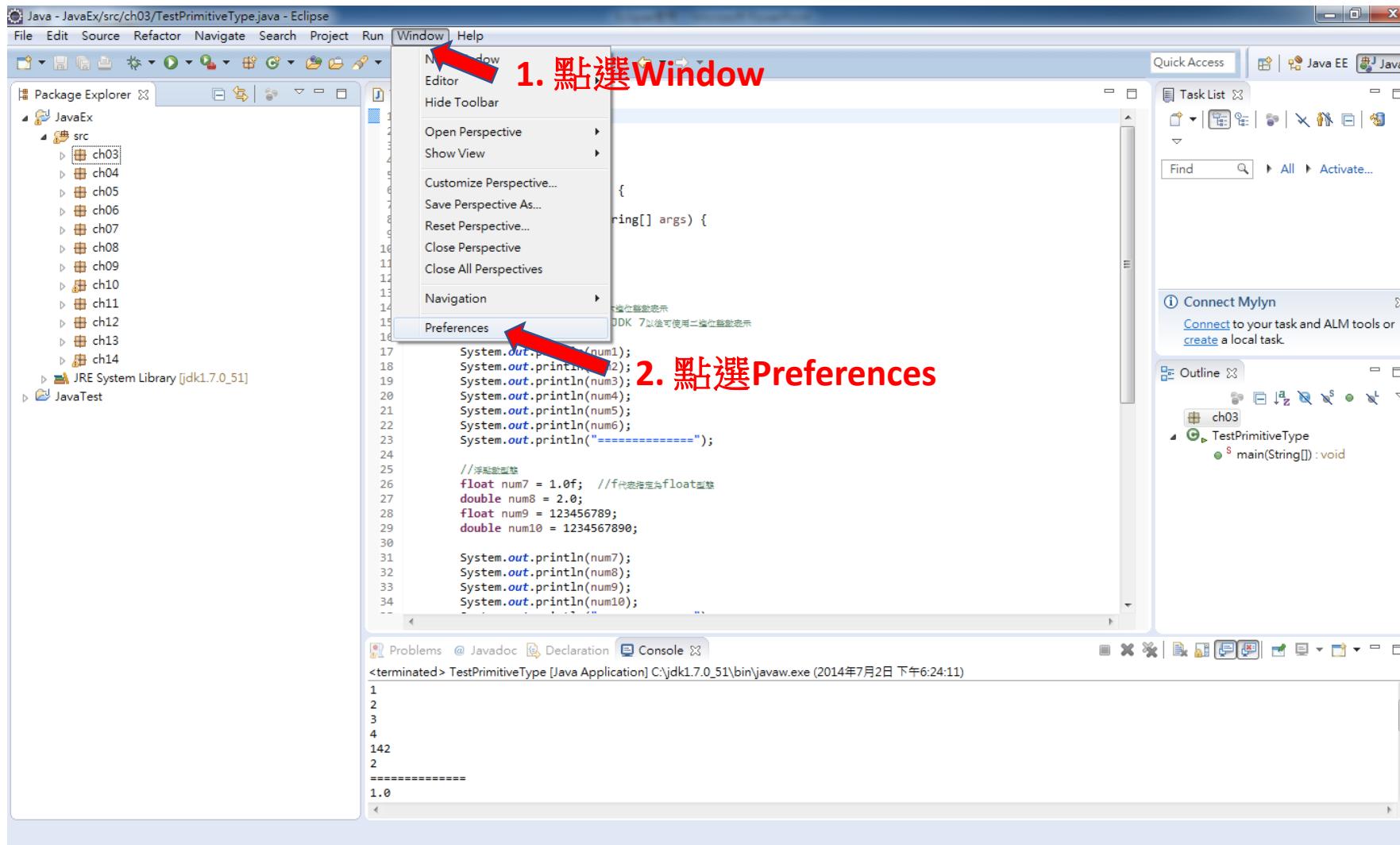


2. Basic → Text Font(雙點)即可調整

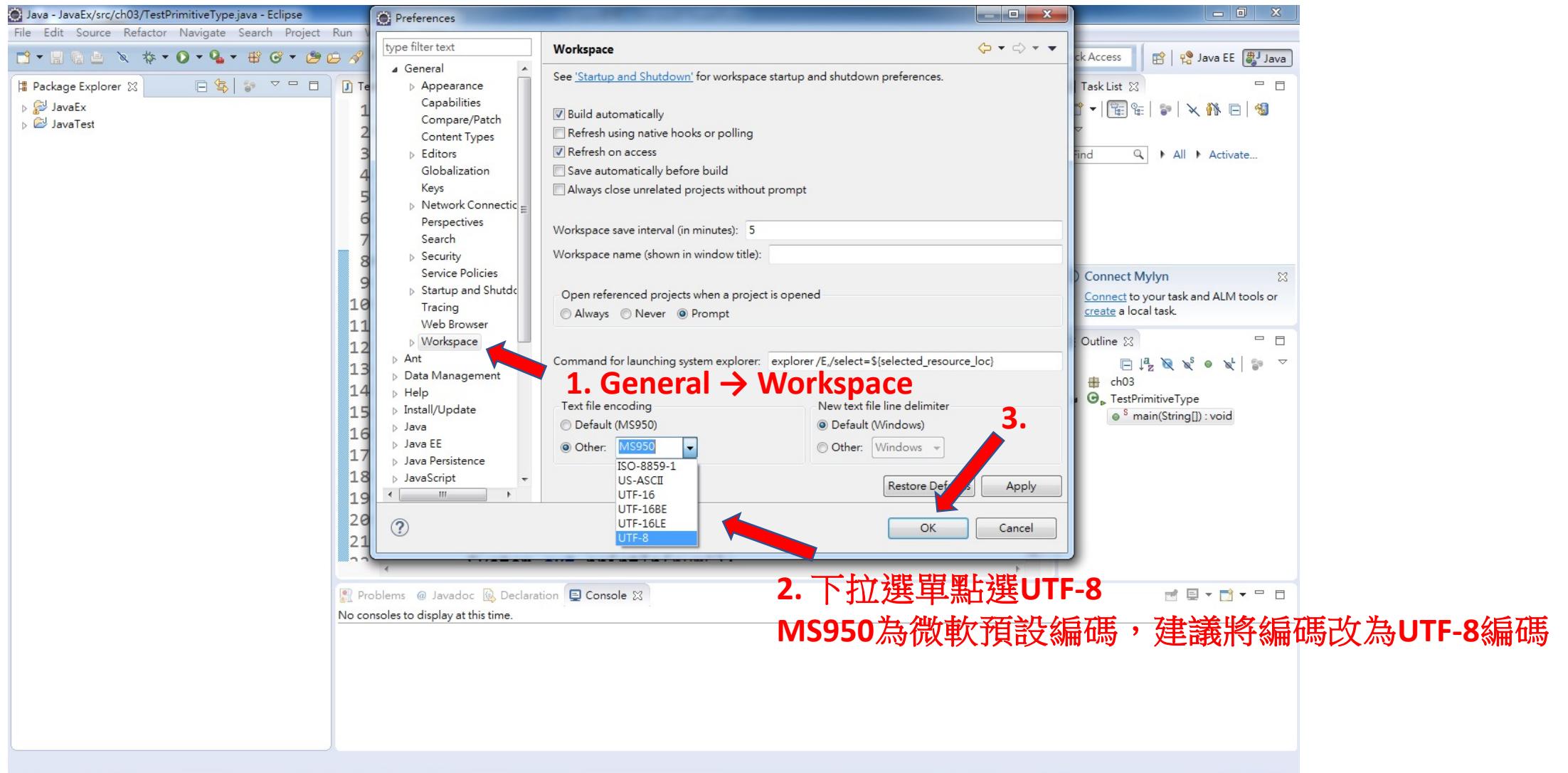
偏好設定 – 字型大小



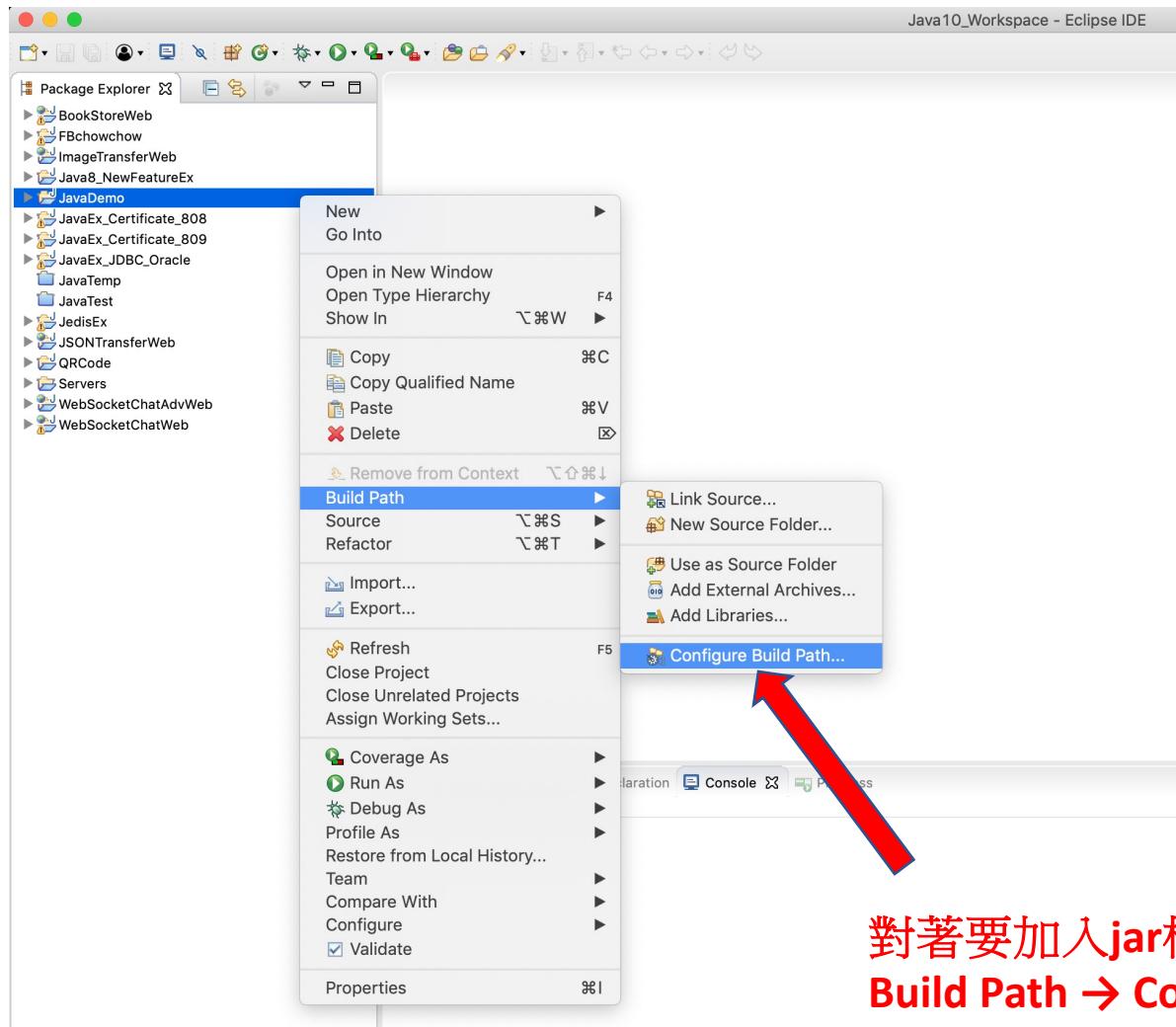
偏好設定 – 專案編碼



偏好設定 - 專案編碼

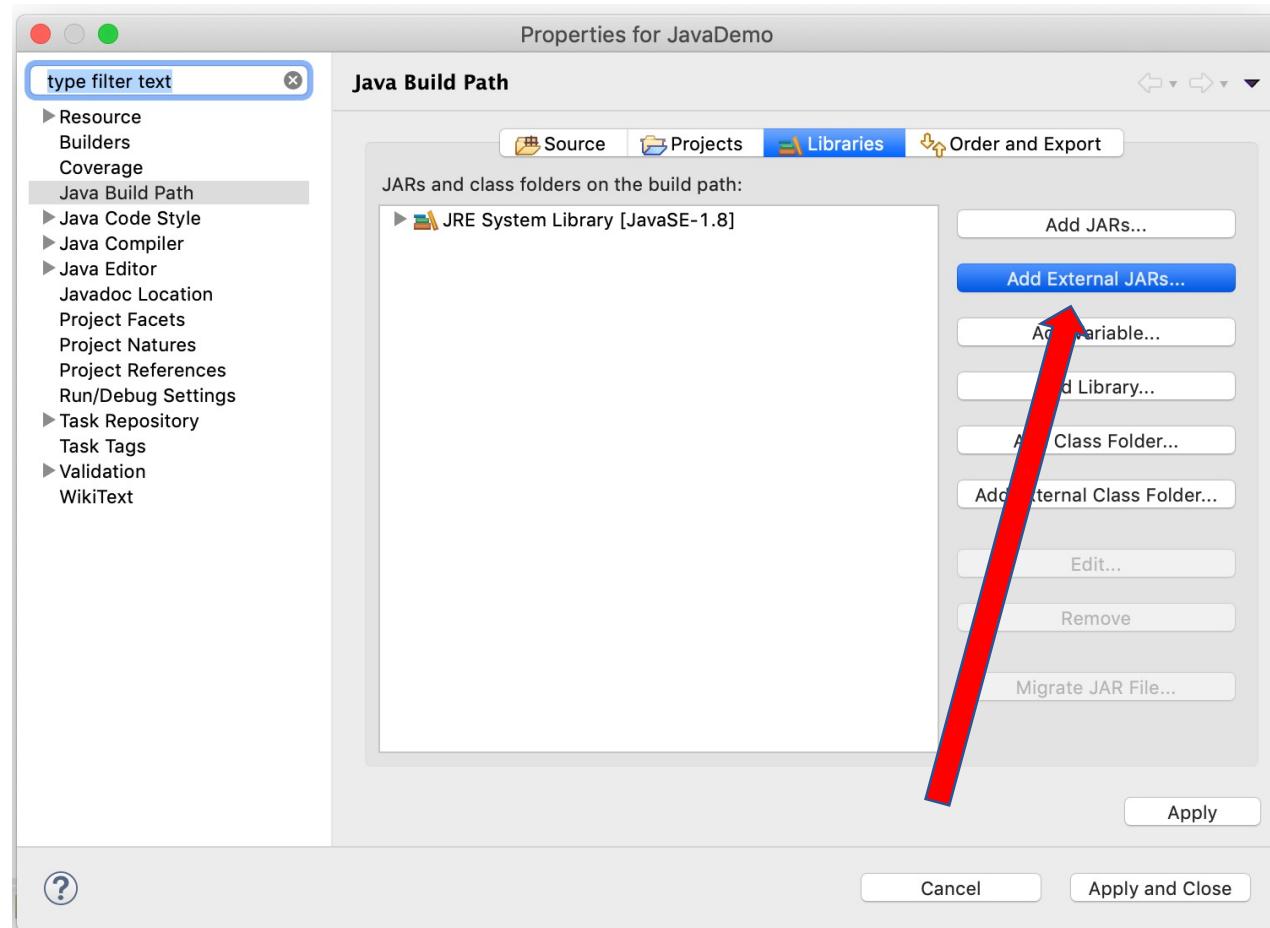


Java Project引入外部jar檔



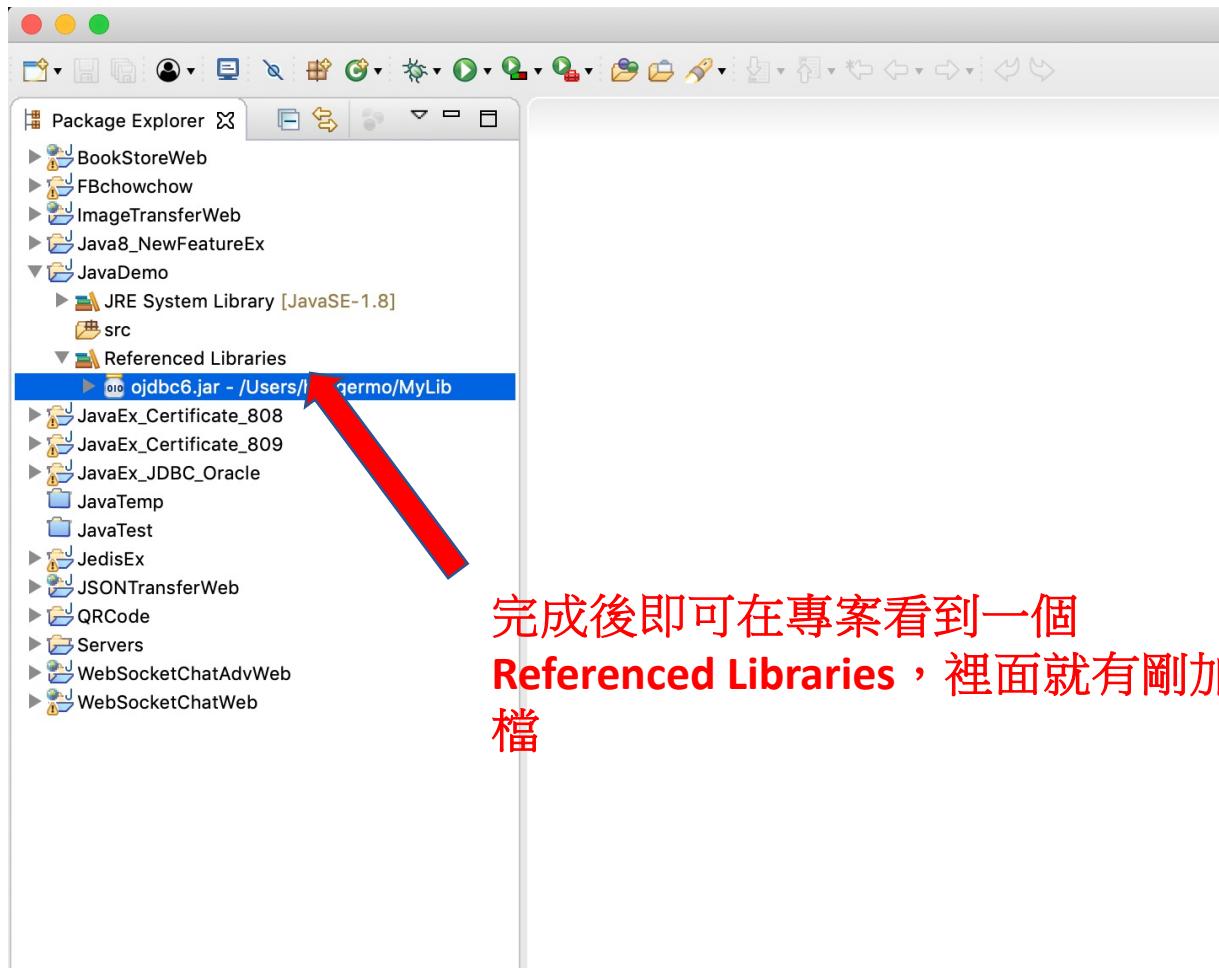
對著要加入jar檔的專案點選滑鼠右鍵
Build Path → Configure Build Path...

Java Project引入外部jar檔



點選**Add External JARs**，並選擇要加入的jar
檔

Java Project引入外部jar檔



完成後即可在專案看到一個
Referenced Libraries，裡面就有剛加入的jar
檔

Eclipse常用快速鍵

- 輔助完成程式碼 : Alt + /
- 程式碼格式化排列整齊(程式碼不可以有錯誤發生) : Ctrl + Shift + F
- 自動import (1/2) : Ctrl + Shift + M 【Add import】
- 自動import (2/2) : Ctrl + Shift + O 【Organize imports】
- 刪除單行 : Ctrl + D
- 搜尋關鍵字 : Ctrl + F
- 註解單行(游標移至該行後) : Ctrl + /
- 註解多行(將欲註解的程式碼反白後) : Ctrl + /
- 執行Java程式 : Ctrl + F11