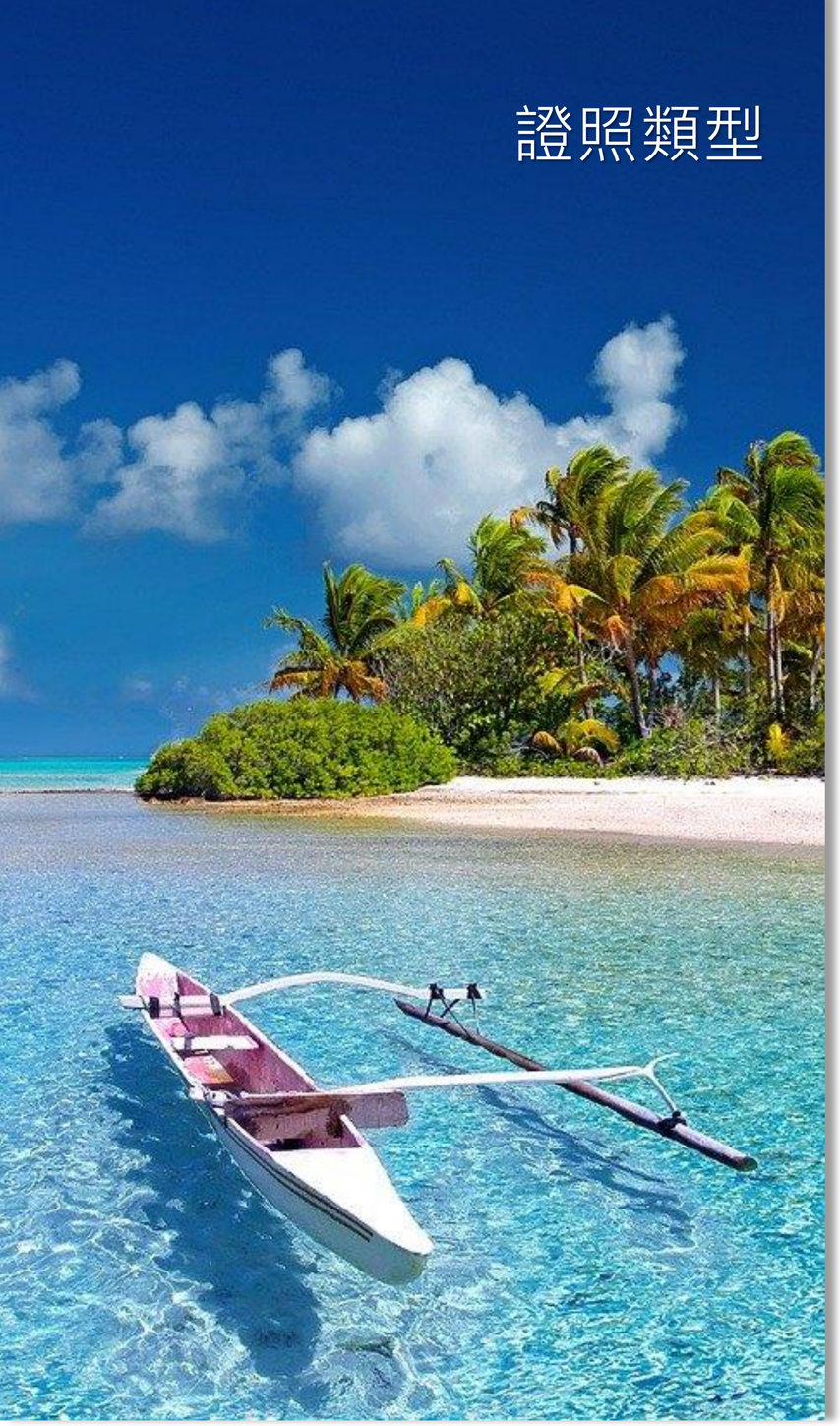
ORACLE
Certified
Professional
Java SE 11 Developer
Java
25 Years

JAVA OCP

同學，歡迎你參加本課程

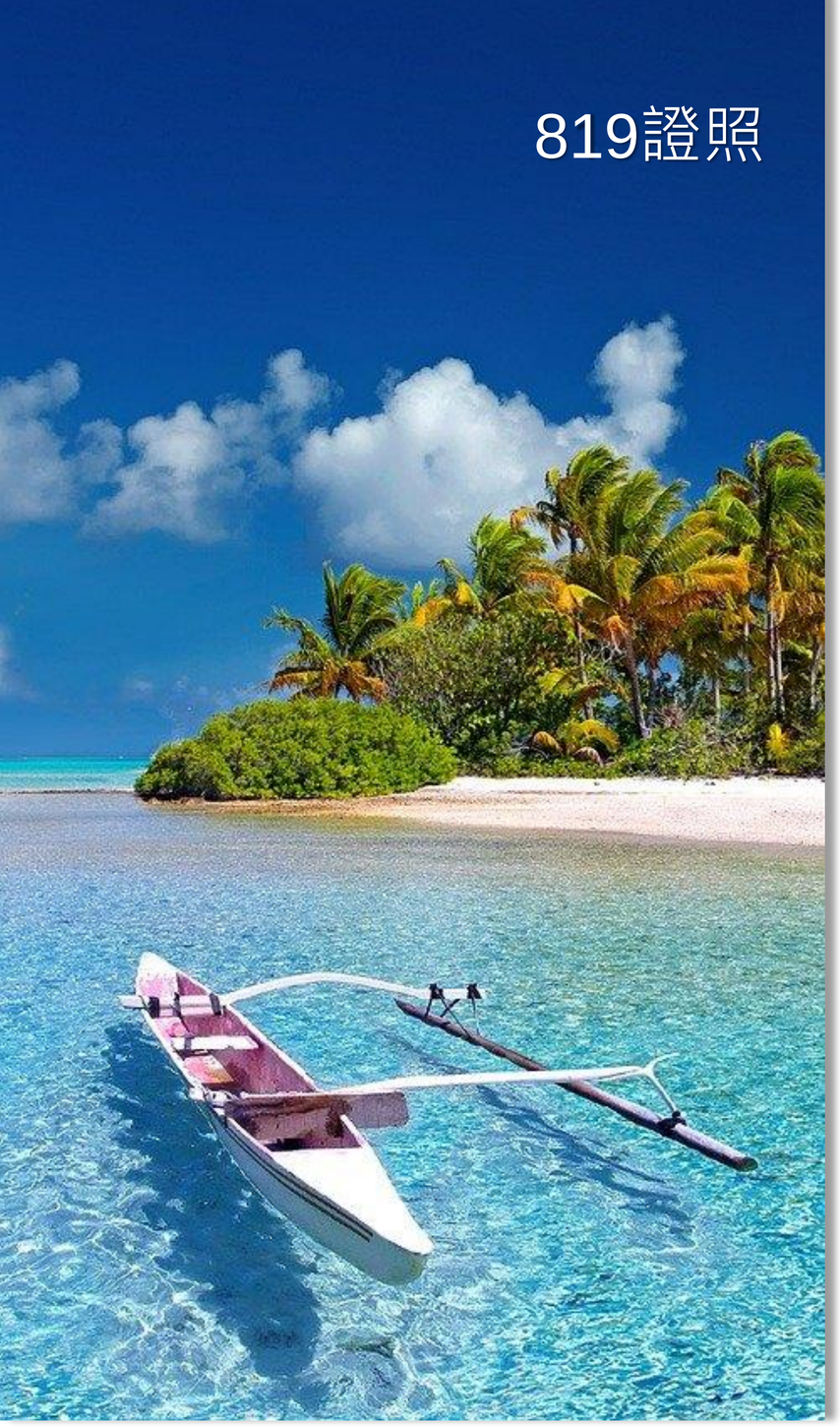
- 請關閉你的FB 、Line等溝通工具，以免影響你上課。
- 考量頻寬、雜音，請預設關閉攝影機、麥克風，若有需要再打開。
- 隨時準備好，老師會呼叫你的名字進行互動，鼓勵用麥克風提問。
- 如果有緊急事情，你必需離開線上教室，請用聊天室私訊給老師，以免老師癡癡呼喚你的名字。
- 軟體安裝請在上課前安裝完成，未完成的同學，請盡快進行安裝。

證照類型



- 1Z0-819
  - OCP(Oracle Certified Professional)
  - JAVA SE 11 Developer
  - 考試時間 90分鐘
  - 題數 50題
  - 滿分1000分 及格分數 680分
  - 題型：單選　複選
  - 考試費用　US$245元約新台幣7350元
  - 考試範圍
    - Java程式語言的基本開發方式，像是定義變數、類別、方法等，還要知道迴圈和例外。
    - 物件導向觀念
    - 字串與集合相關類別(如：StringBuilder、ArrayList)，
    - Lambda表示式和時間與日期套件的用法類別、方法的設計
    - 泛型、集合的觀念和應用
    - 串流、例外、Assertion、JDBC、檔案和I/O相關類別的使用方式
    - 日期與時間套件的用法，多執行緒程式的理解，本地化的基本概念
    - 模組應用

819證照



ORACLE University

Oracle Certified Professional

Certificate of Recognition

HSI YI PAN

Oracle Certified Professional: Java SE 11 Developer

This certifies that the above named is recognized by Oracle Corporation as Oracle Certified.

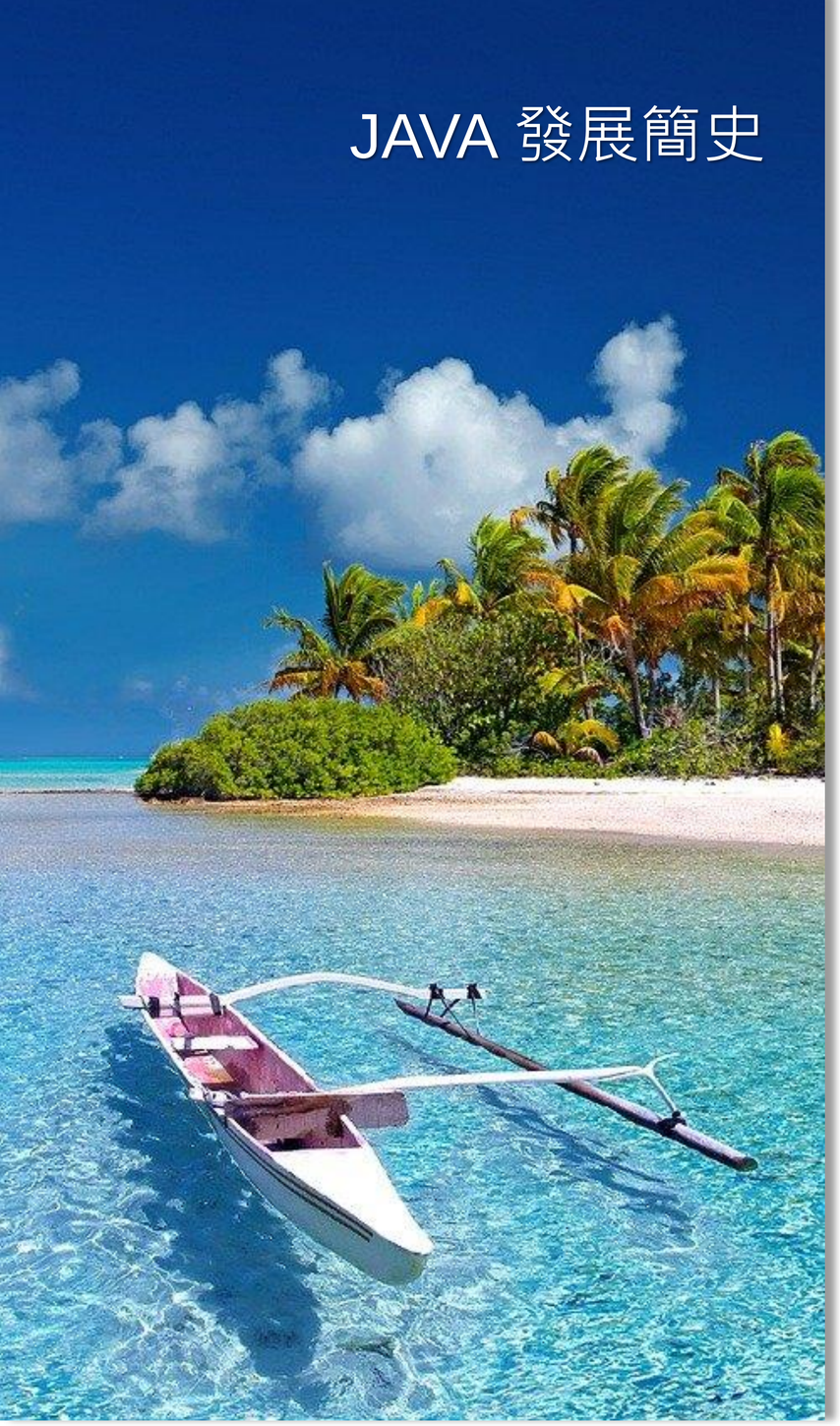
February 25, 2022

Date

Damien Carey
Senior Vice President, Oracle University

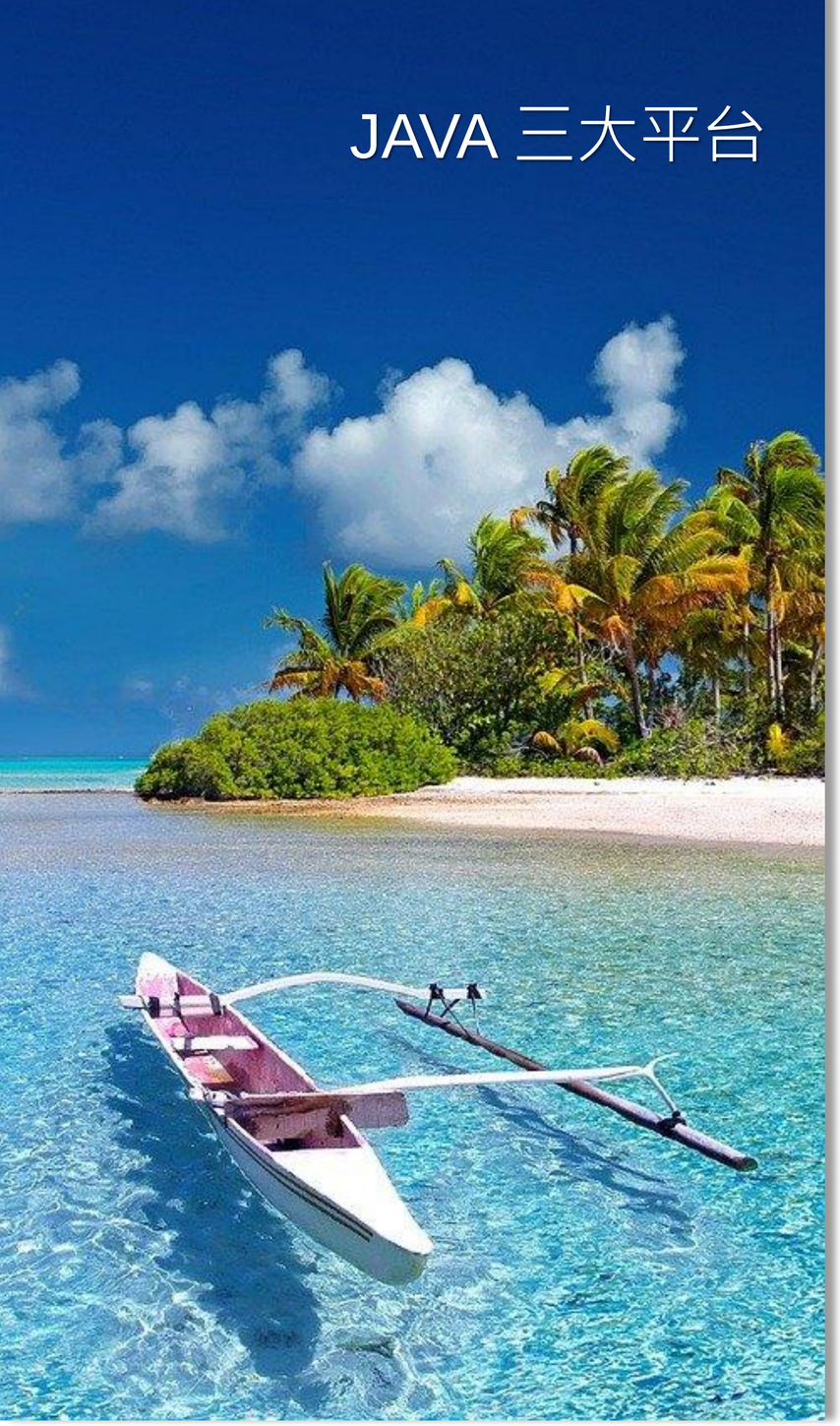
290641011OCPJSE11

JAVA 發展簡史

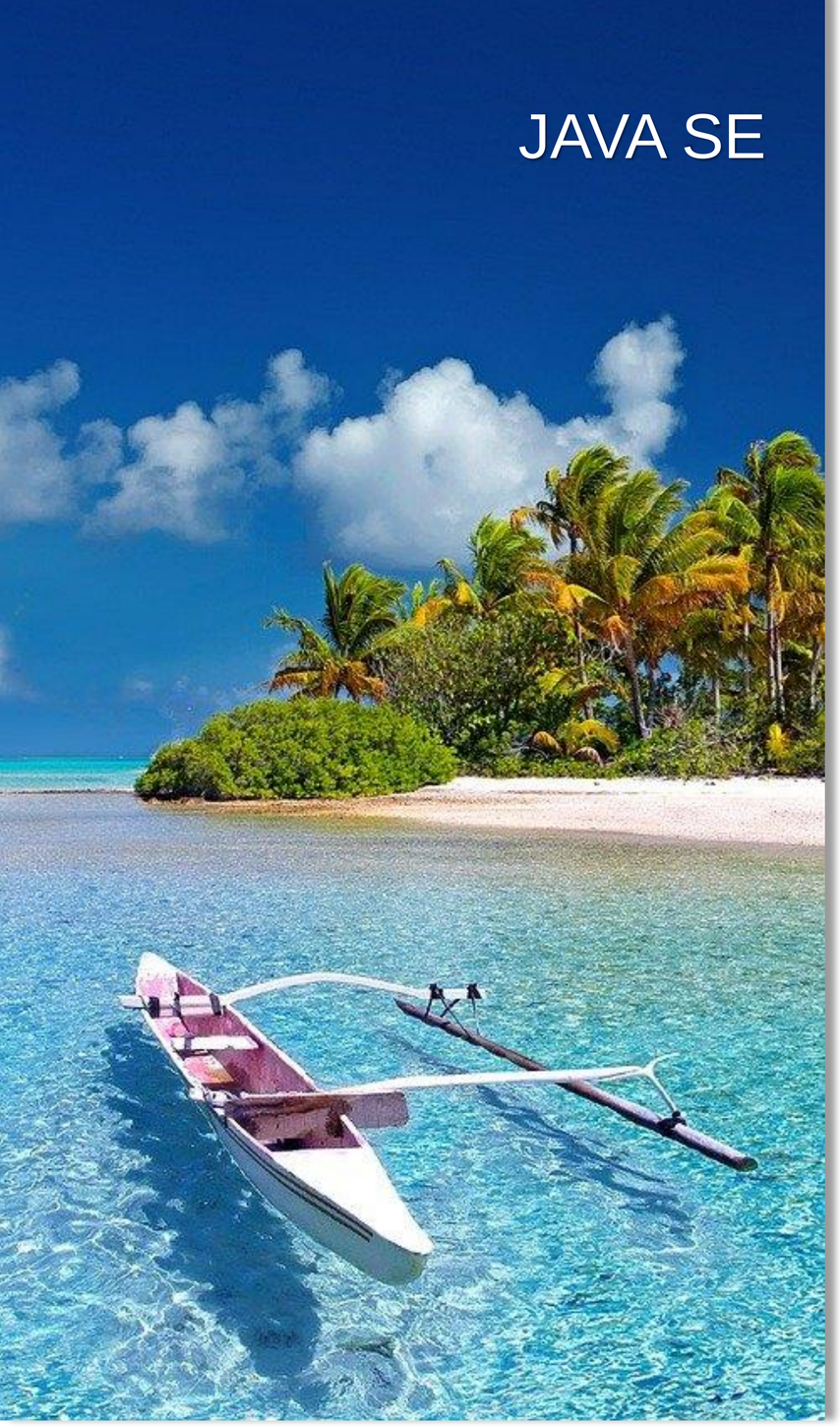


Green計畫	1991 Green計畫(Java 的前身 Oak)			
Sun	1995 Java 1.0	1996 JDK 1.0	1997 JDK 1.1	1998 J2SE 1.2
Sun	2000 J2SE 1.3	2002 J2SE 1.4	2004 J2SE 5.0	2006 Java SE 6
Oracle	2009 Oracle收購	2011 Java SE 7	2014 Java SE 8	2017 Java SE 9

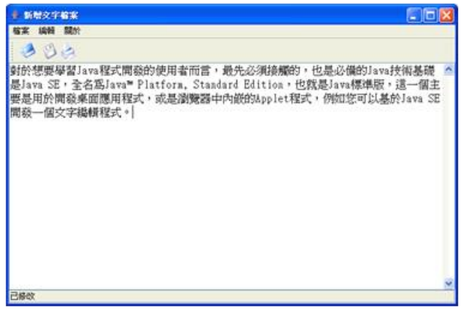
JAVA 三大平台

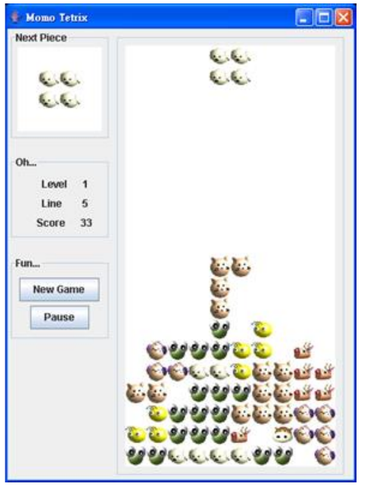


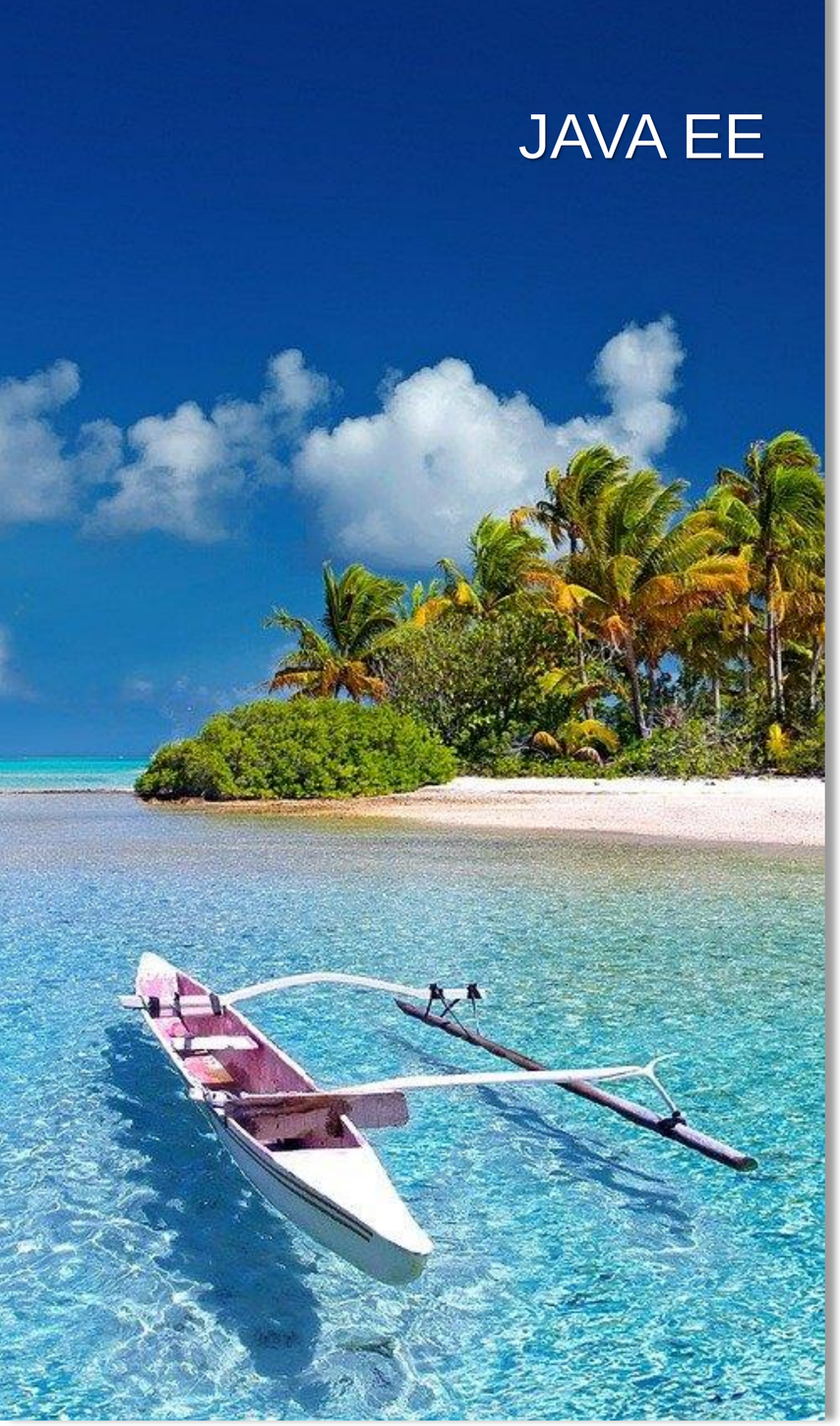
- Sun 公司在 1999 年 6 月公佈了 Java 三大平台架構
  - J2SE（Java 2 Standard Edition）
  - J2EE（Java 2 Enterprise Edition）
  - J2ME（Java 2 Micro Edition）
  - Java SE 6 後 Java不再帶有 "2"



- Java™ Standard Edition：Java 標準版
  - 主要用於開發單機桌面應用程式

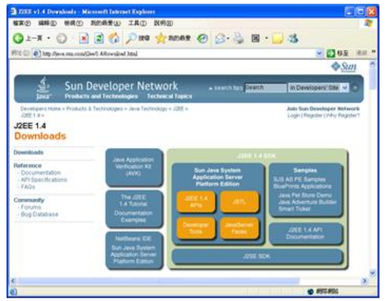


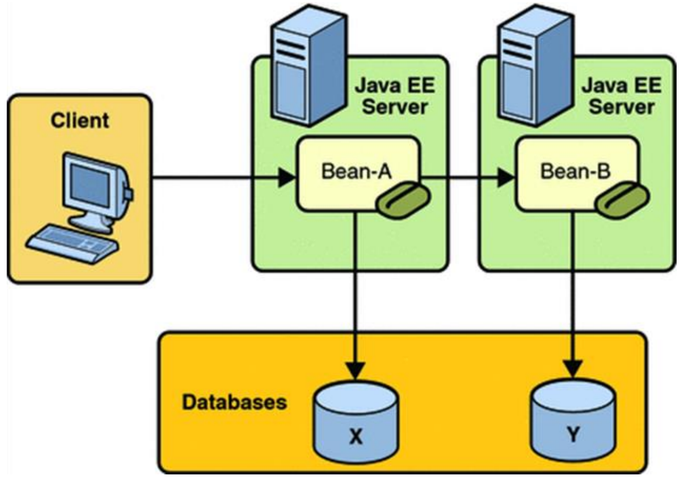




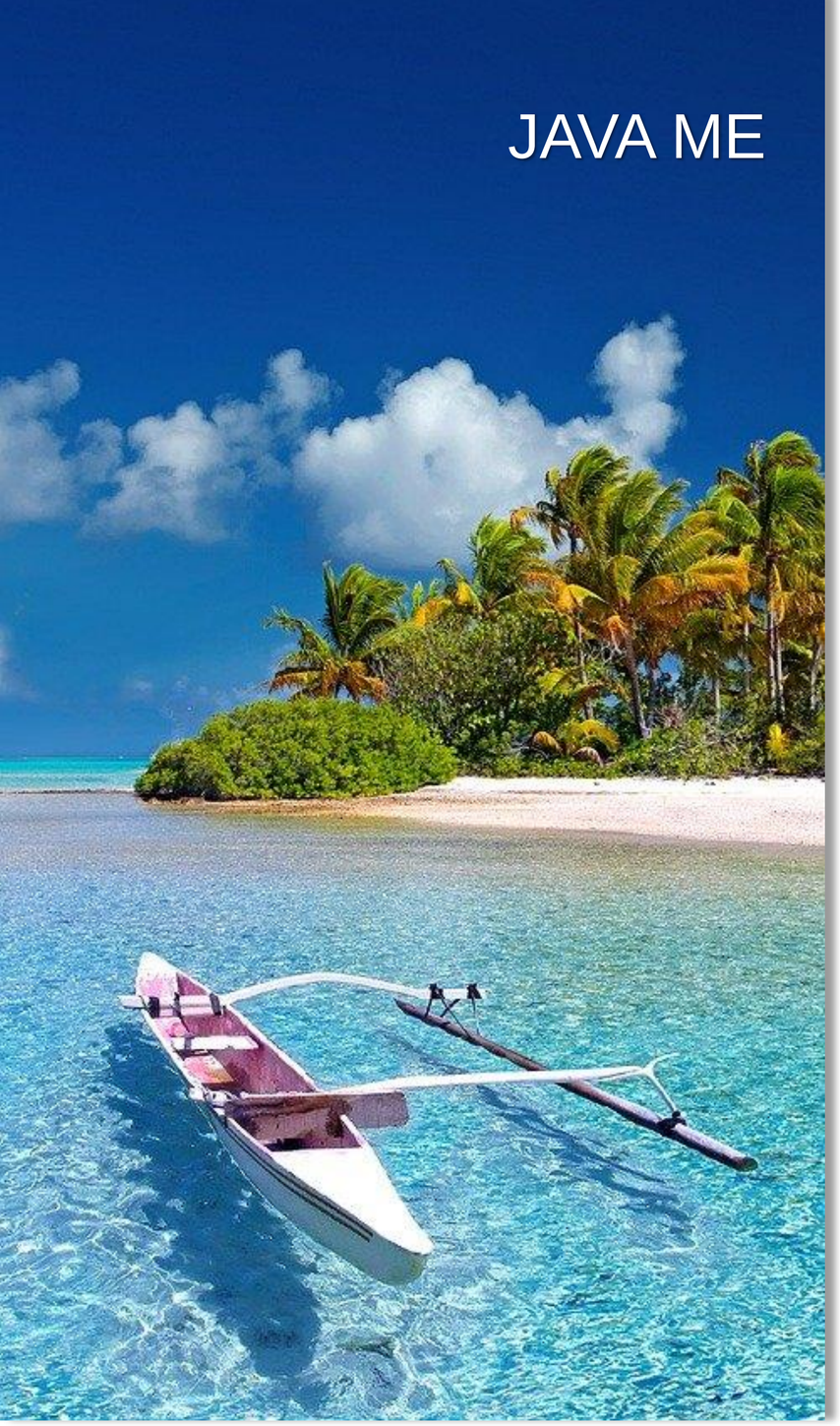
JAVA EE

- Java™ Enterprise Edition：Java 企業版
  - 包含 JSP、Servlet、EJB 等多種技術
  - 分散式架構
  - 以 Java SE 為基礎



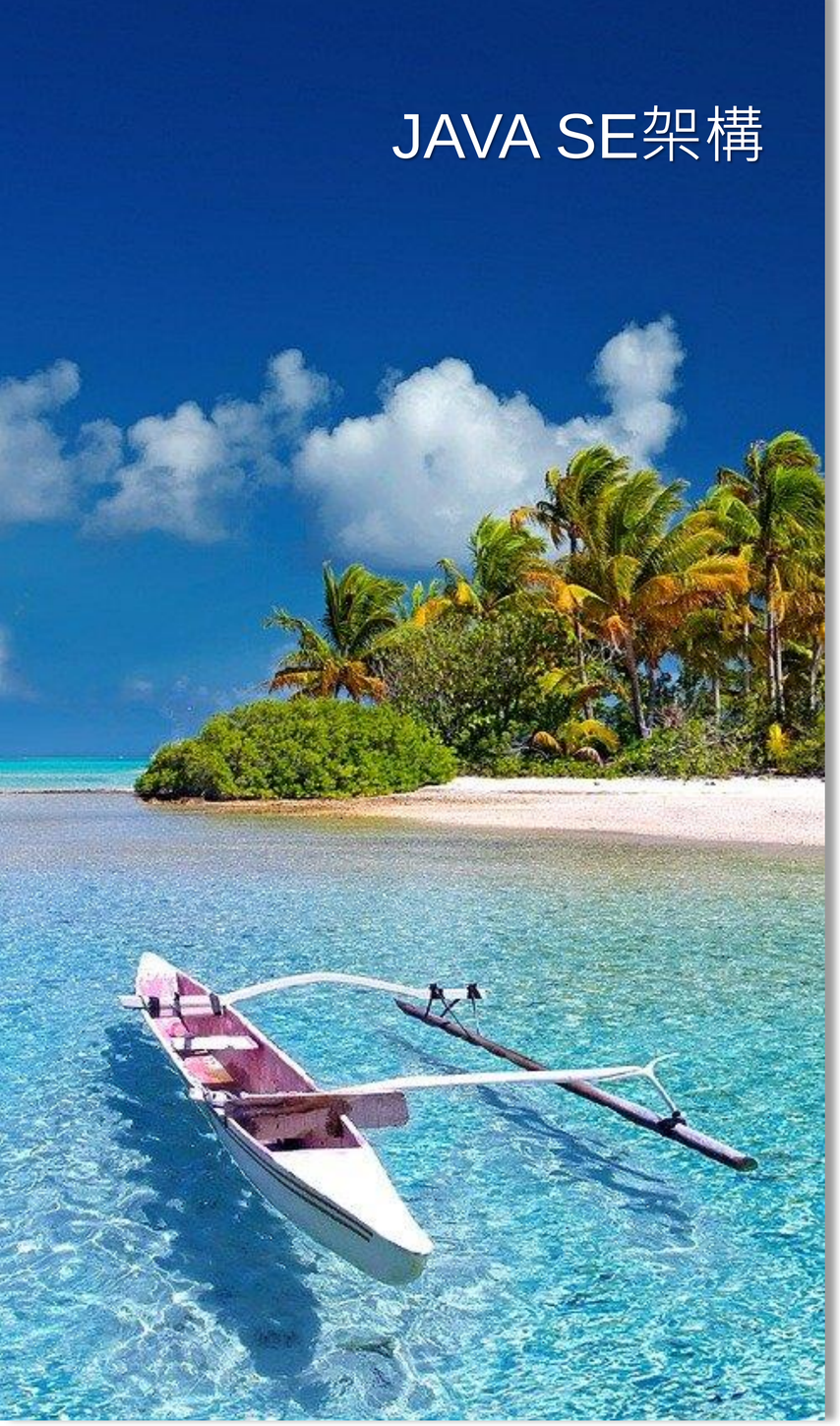


JAVA ME



- Java™ Micro Edition：Java 微型版
  - 針對消費性行動電子產品，例如手機或是 PDA
  - 物聯網嵌入式系統

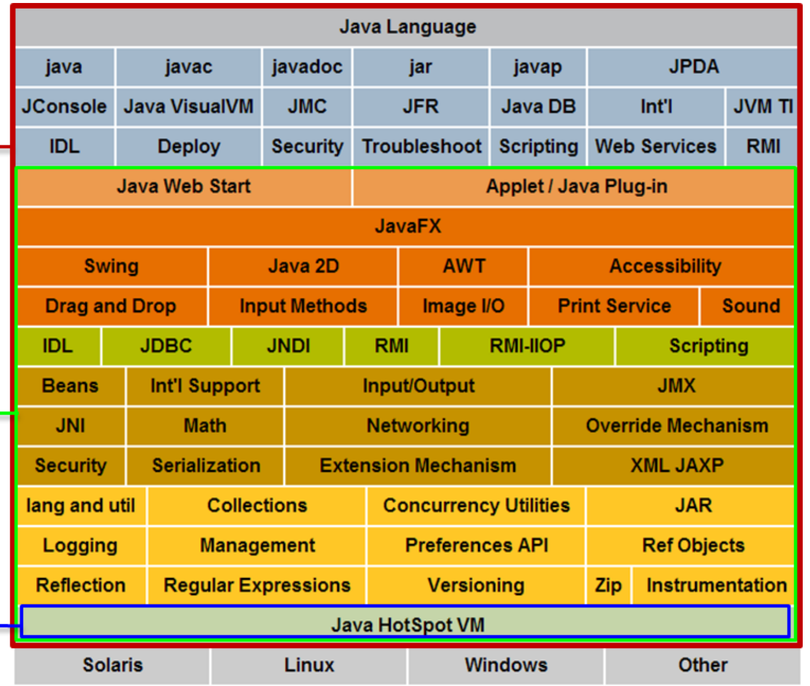
JAVA SE架構



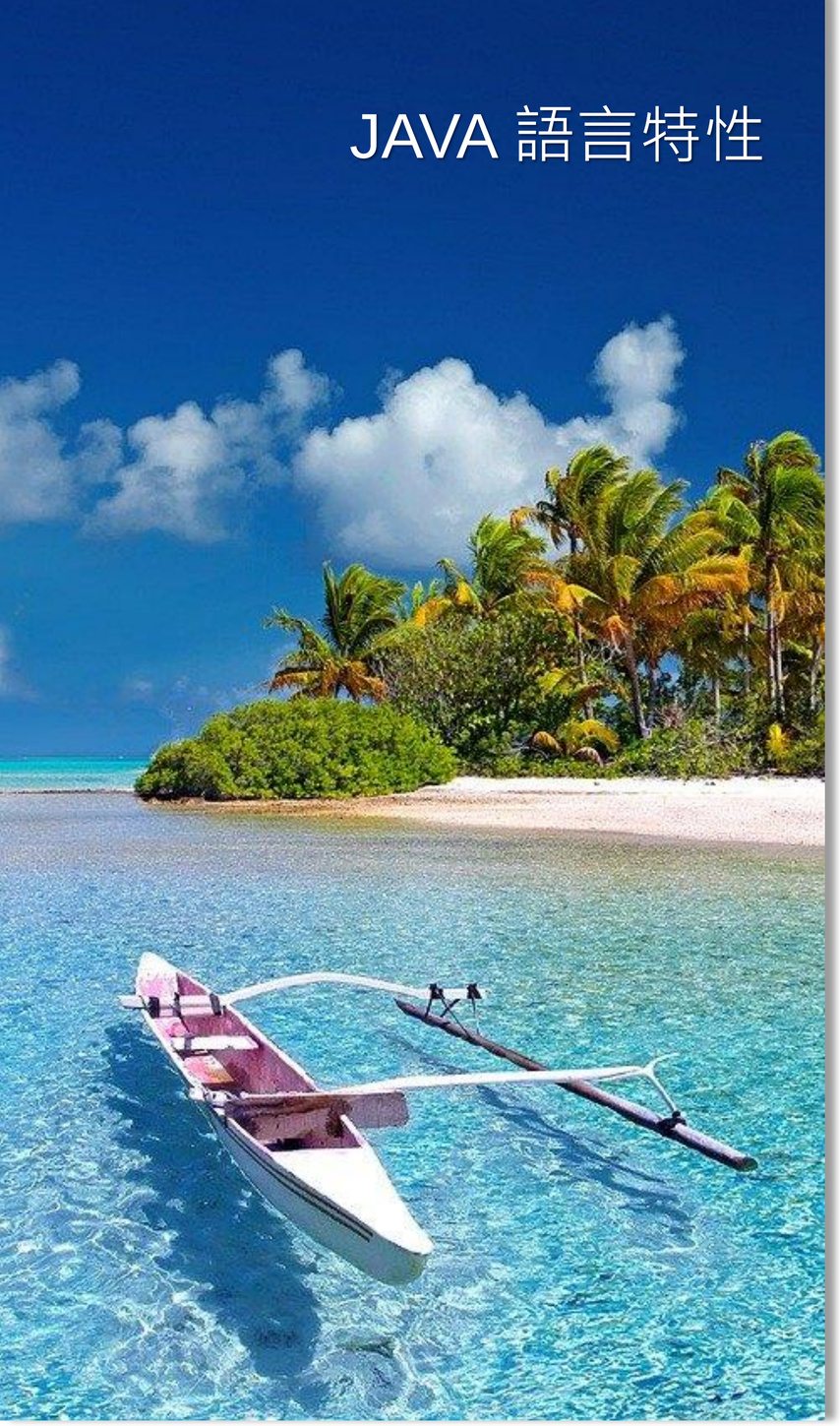
JDK Java標準版開發工具：
提供編譯器，及相關的開發、執行、測試等工具

JRE Java執行環境：
提供Java應用程式相關資源的一個環境
如標準程式庫、Java虛擬機器

JVM Java虛擬機器：
將編譯過的Java程式轉換為平台相依的原生碼

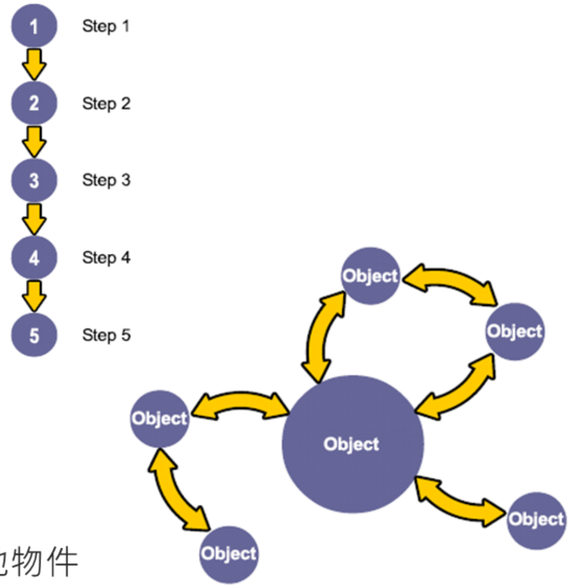


JAVA 語言特性

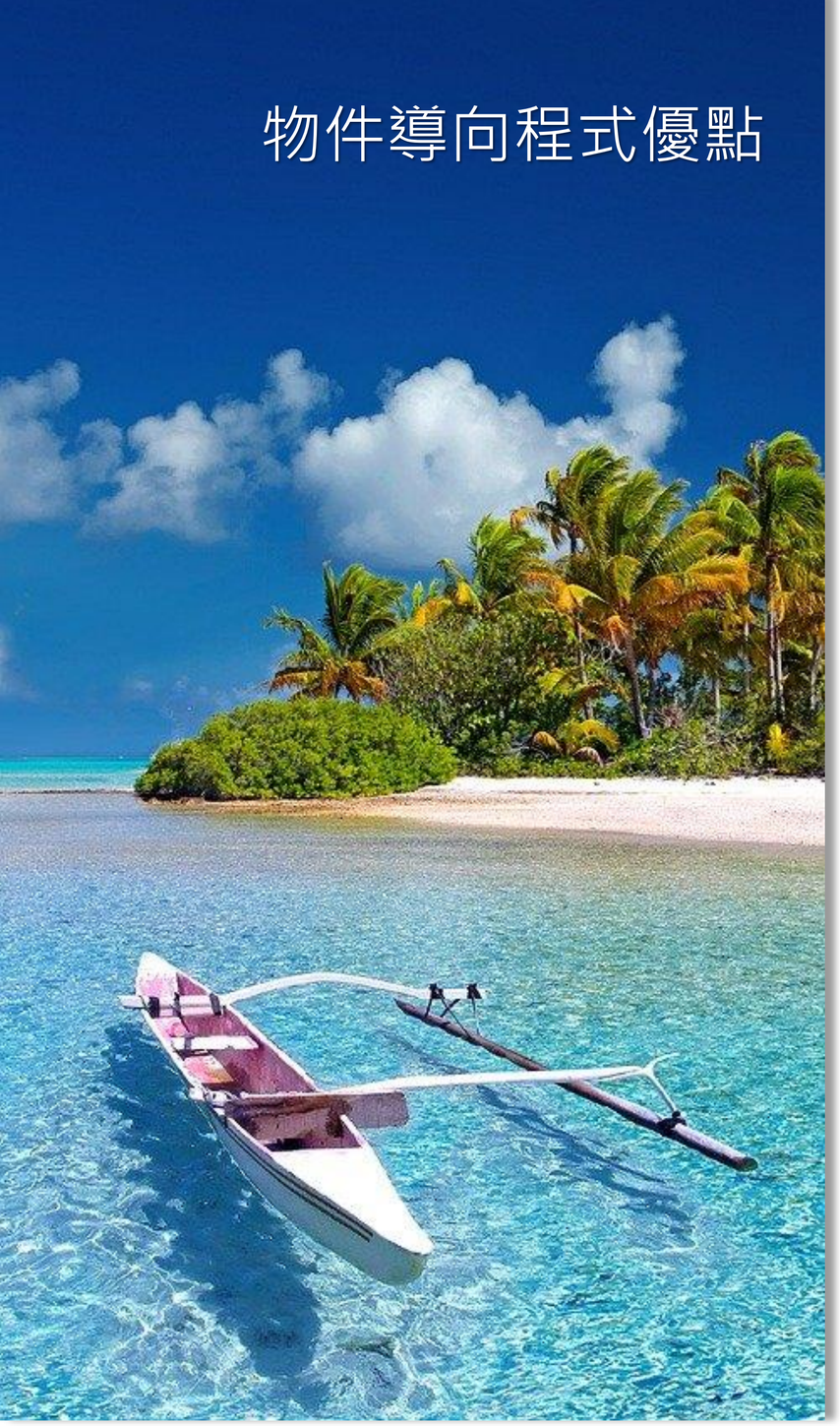


- 物件導向
- 跨平台

- 傳統程序式程式設計
  - 一系列對電腦下達的指令 (procedure 流程)
- 物件導向程式設計
  - 一種抽象且擬人化的程式設計的方法
  - 物件作為程式的基本單元
    - 將資料與操作封裝其中
  - 透過物件之間的交互作用來完成工作
    - 物件能夠接受資料、處理資料並傳遞資料給其他物件

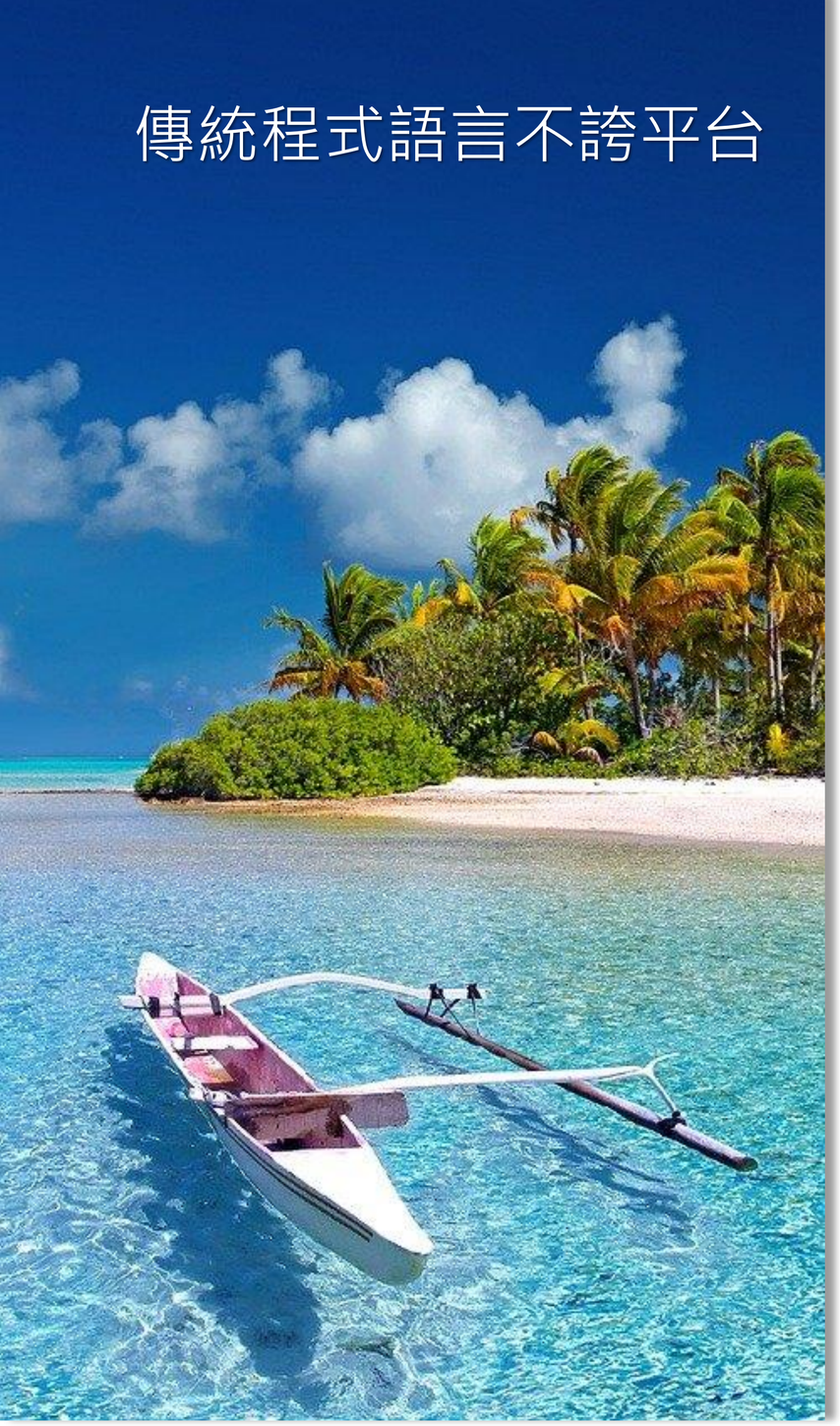


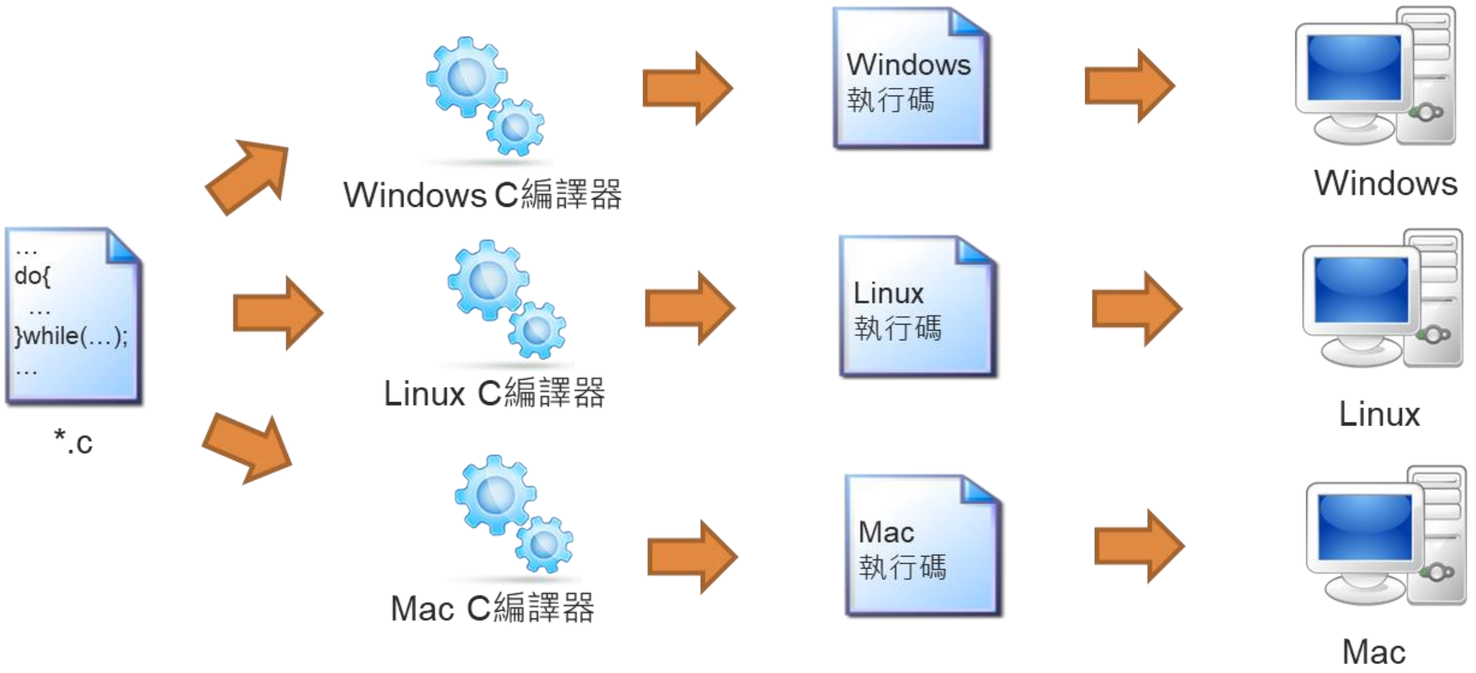
物件導向程式優點



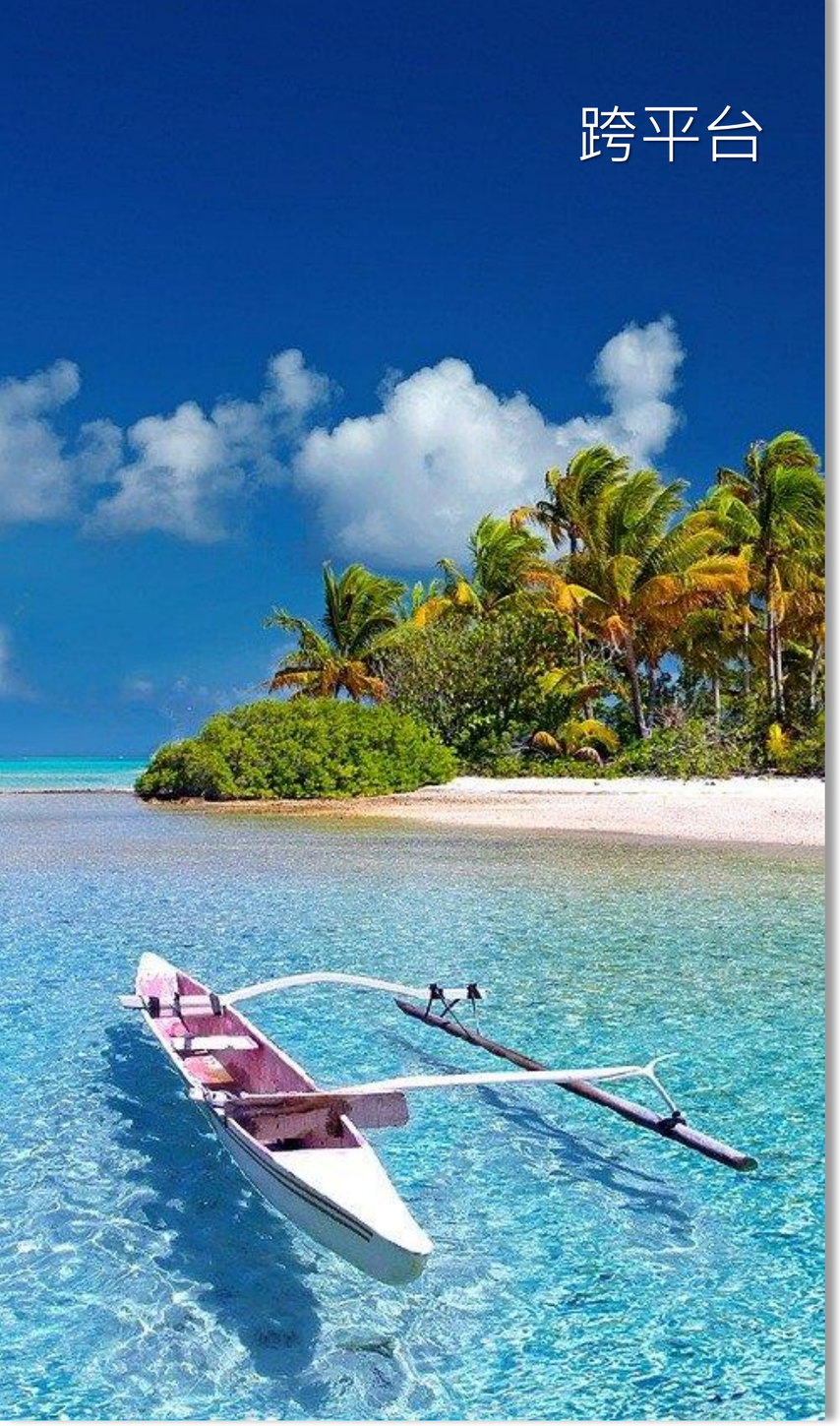
- 物件 (資料與流程) 重複使用
- 分散式開發
- 提高程式的靈活性和可維護性
- 在大型專案設計中廣為應用

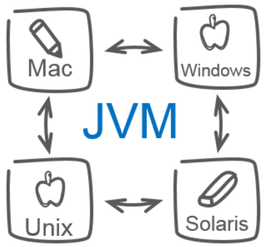
傳統程式語言不跨平台



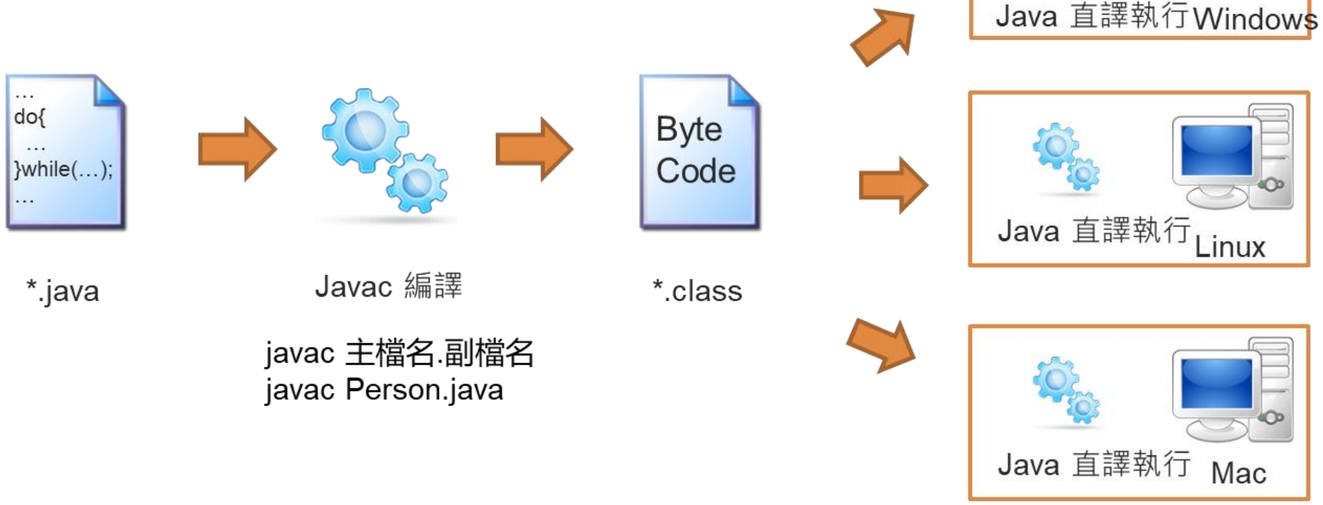

...
do{
...
}while(...);
...
*.c
Windows C編譯器
Linux C編譯器
Mac C編譯器
Windows
執行碼
Linux
執行碼
Mac
執行碼
Windows
Linux
Mac

跨平台

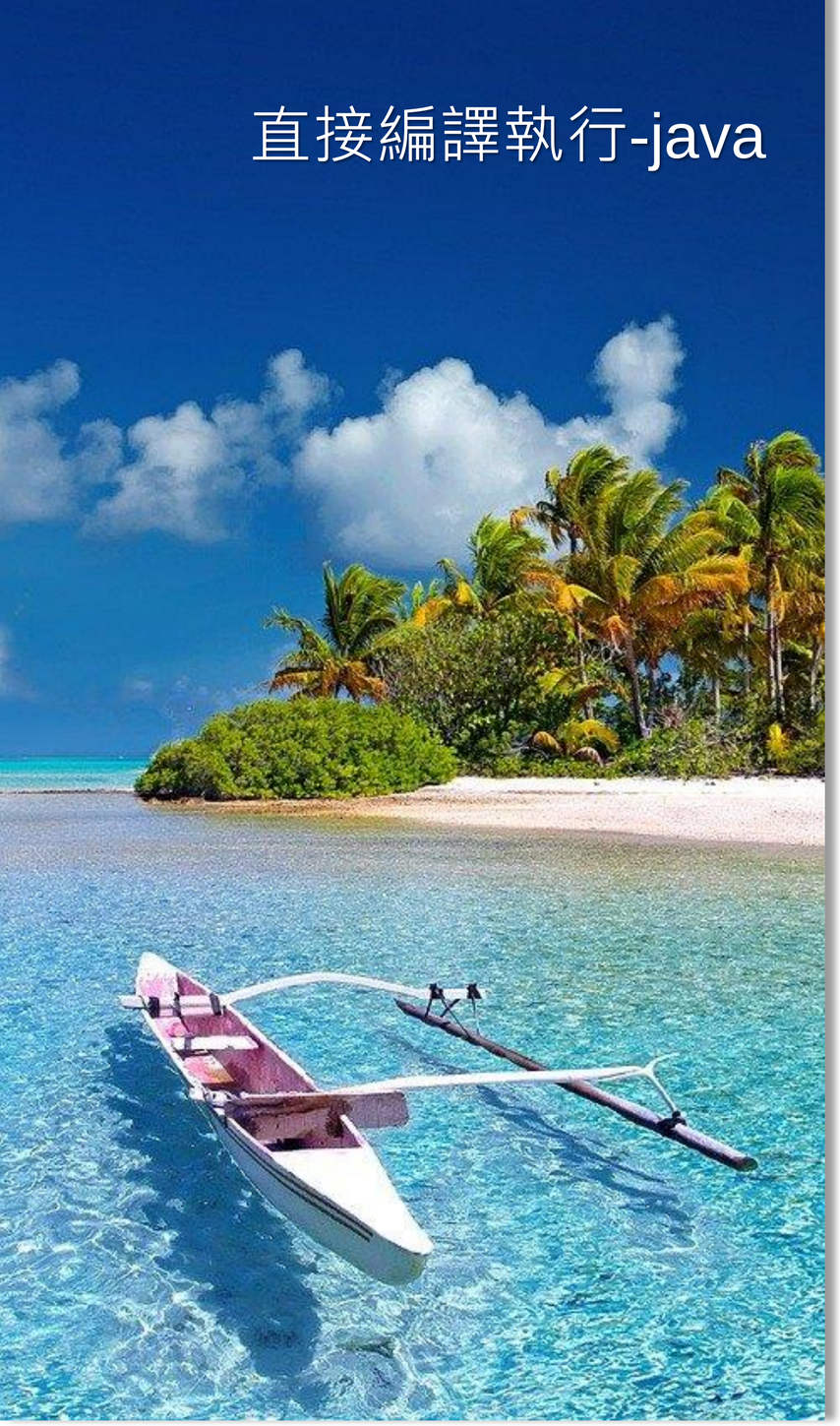




撰寫一次，到處執行 (write once, run anywhere)



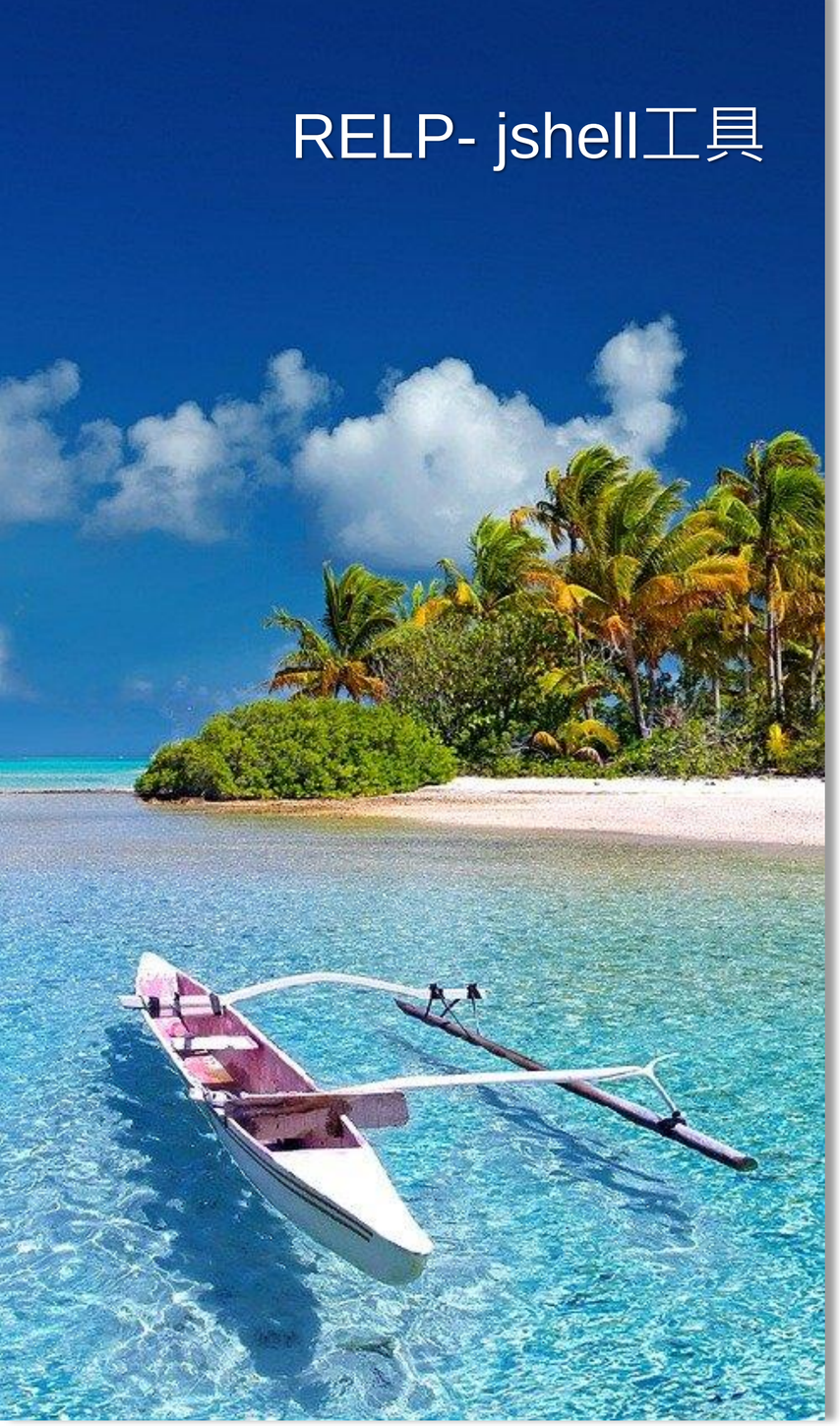
直接編譯執行-java



- 使用java xxx.java來編譯執行
  - class Test2{
  - public static void main(String args[]){
  - System.out.println("Test22");
  - }
  - }

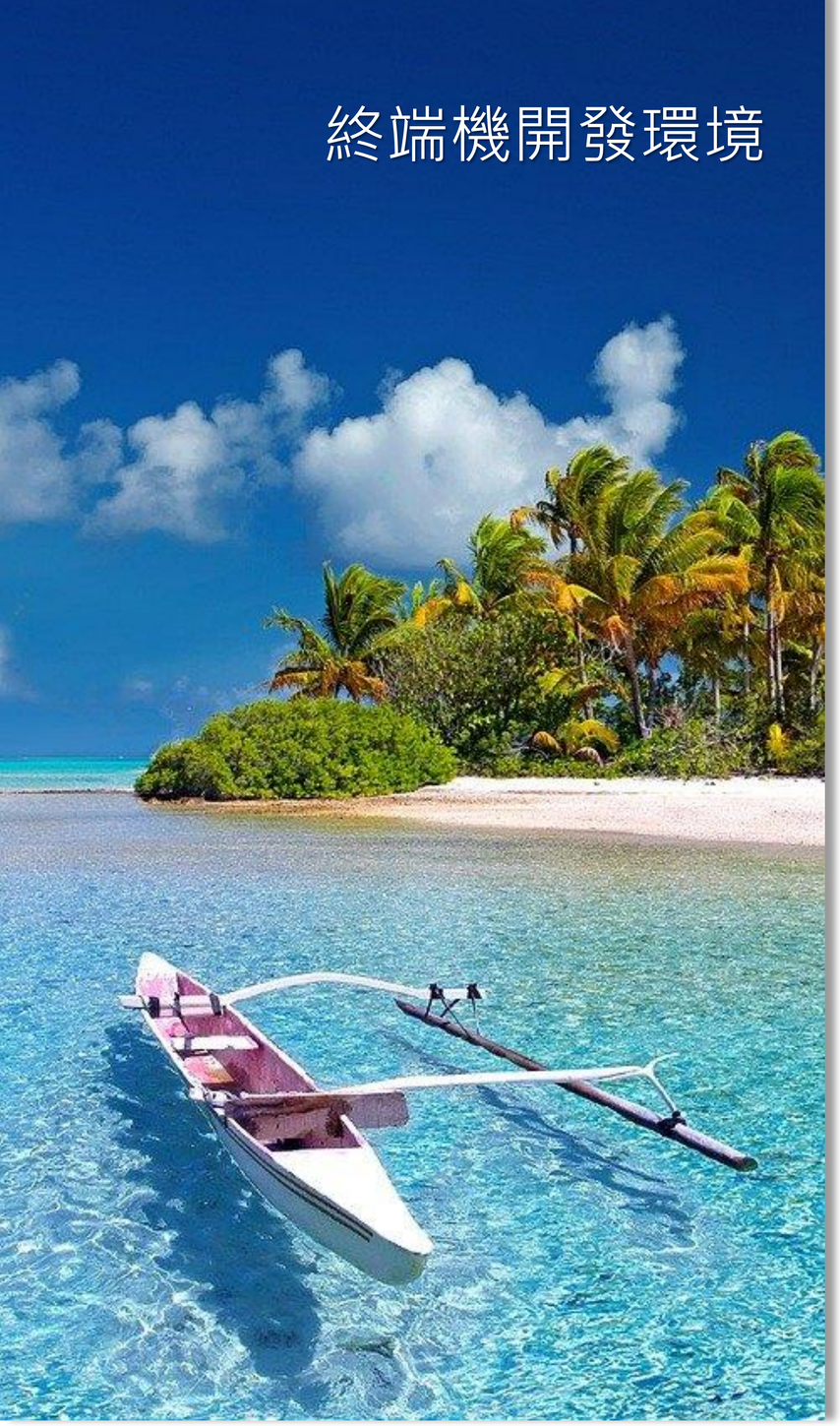
  - class Test{
  - public static void main(String args[]){
  - System.out.println("Test");
  - }
  - }
- 傳統方式只執行主類別,但此方式依主方法之順序為優先

RELP- jshell工具



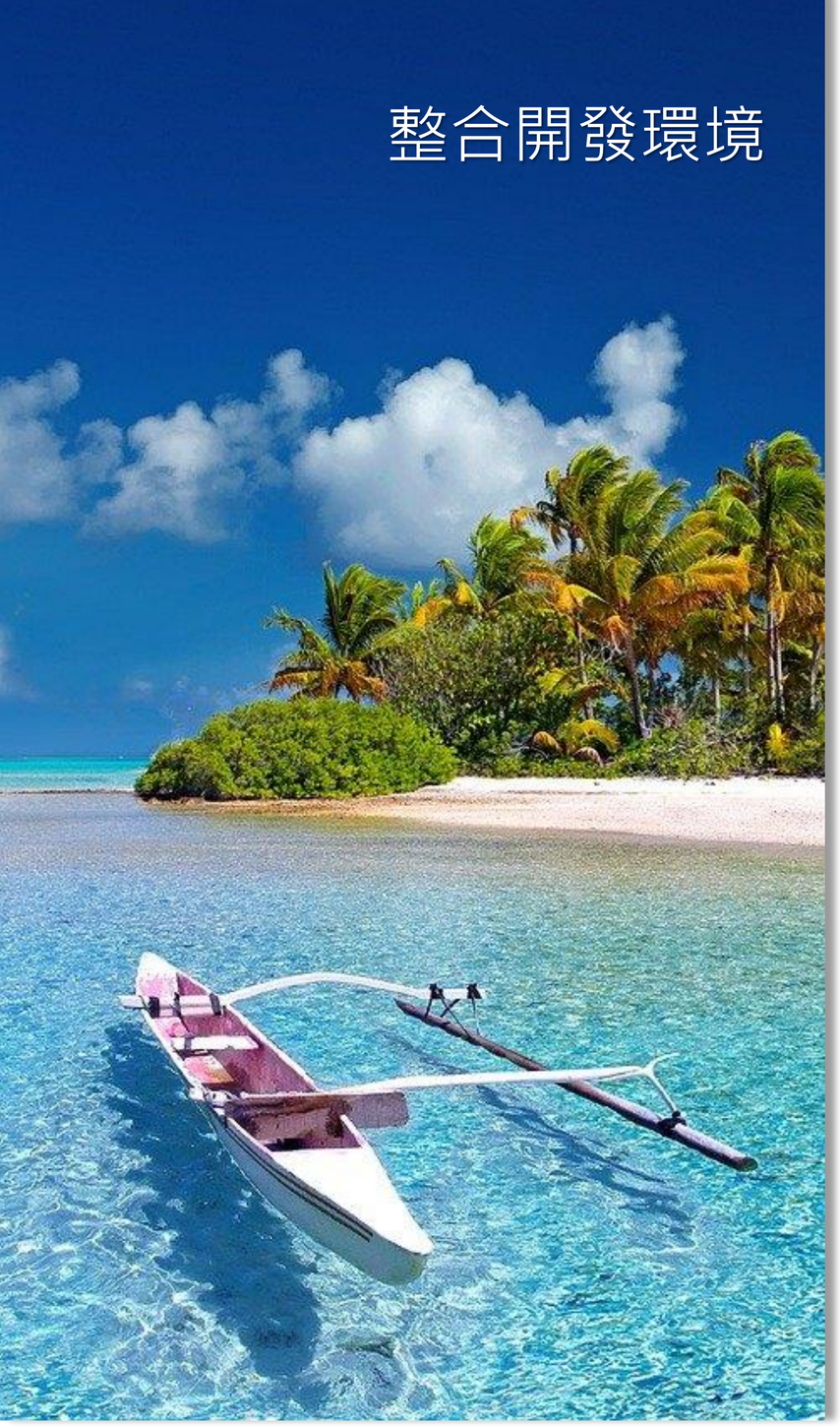
- REPL(Read-Evaluate-Print-Loop)
- 特性
  - 即寫即得
  - 快速運行
  - 直接在命令列撰寫執行
- 常用指令
  - /vars 列出所有宣告的變數
  - /list 列出所有下過的指令
  - /methods列出所有宣告的方法
  - /edit 編輯下過的指令
  - /exit 結束jshell

終端機開發環境



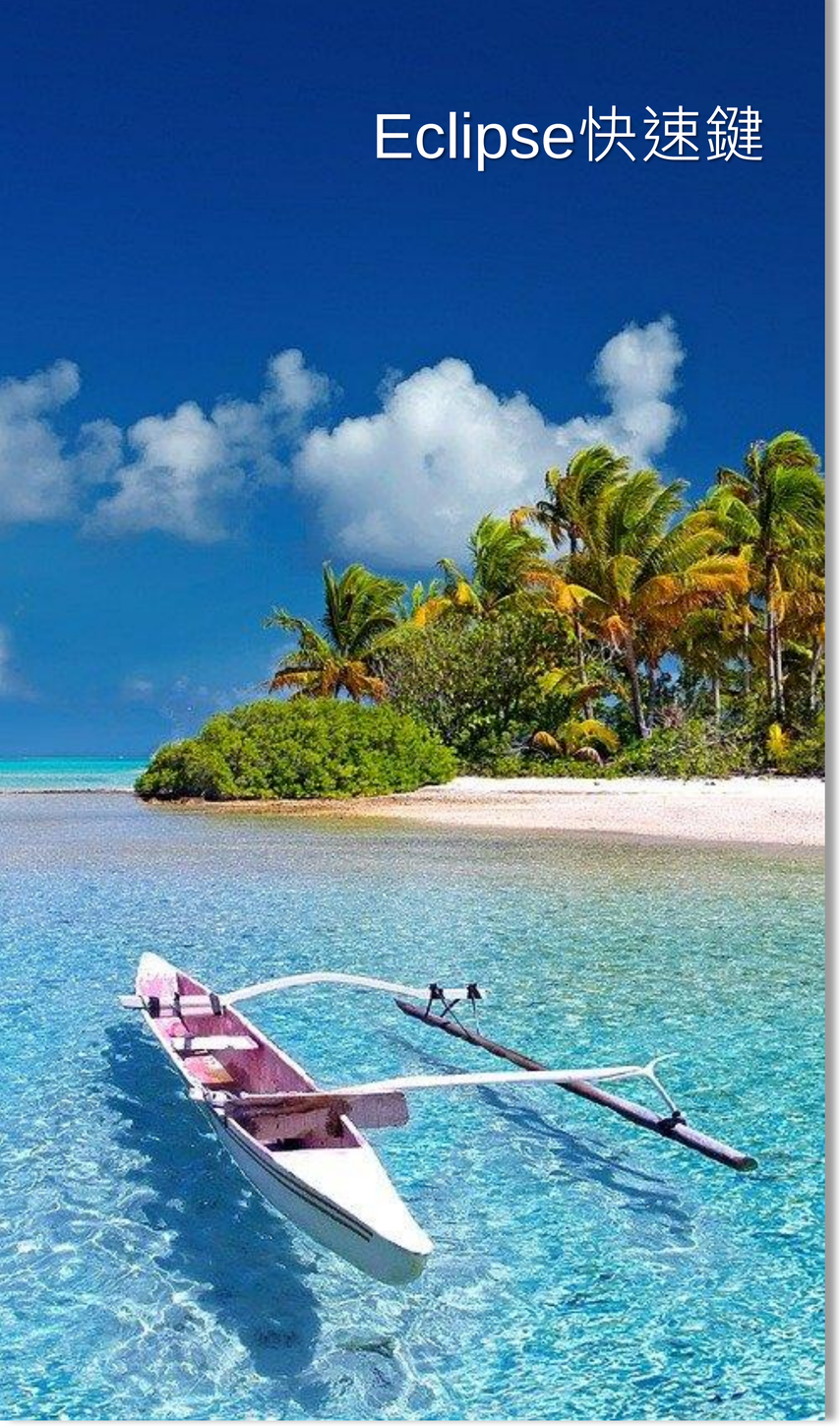
- 終端機模式的開發環境
  - 程式執行環境資料輸入和輸出都是命令列模式（Command-line Interface）
  - 文字模式的鍵盤輸入或單純文字內容的輸出。
- 開發環境建置十分容易
  - 安裝 Java 開發工具 Java Development Kit（JDK）
  - 設定環境變數
  - 配合純文字編輯工具 (vi、記事本或 UltraEdit 等)，就能開發 Java 應用程式

整合開發環境



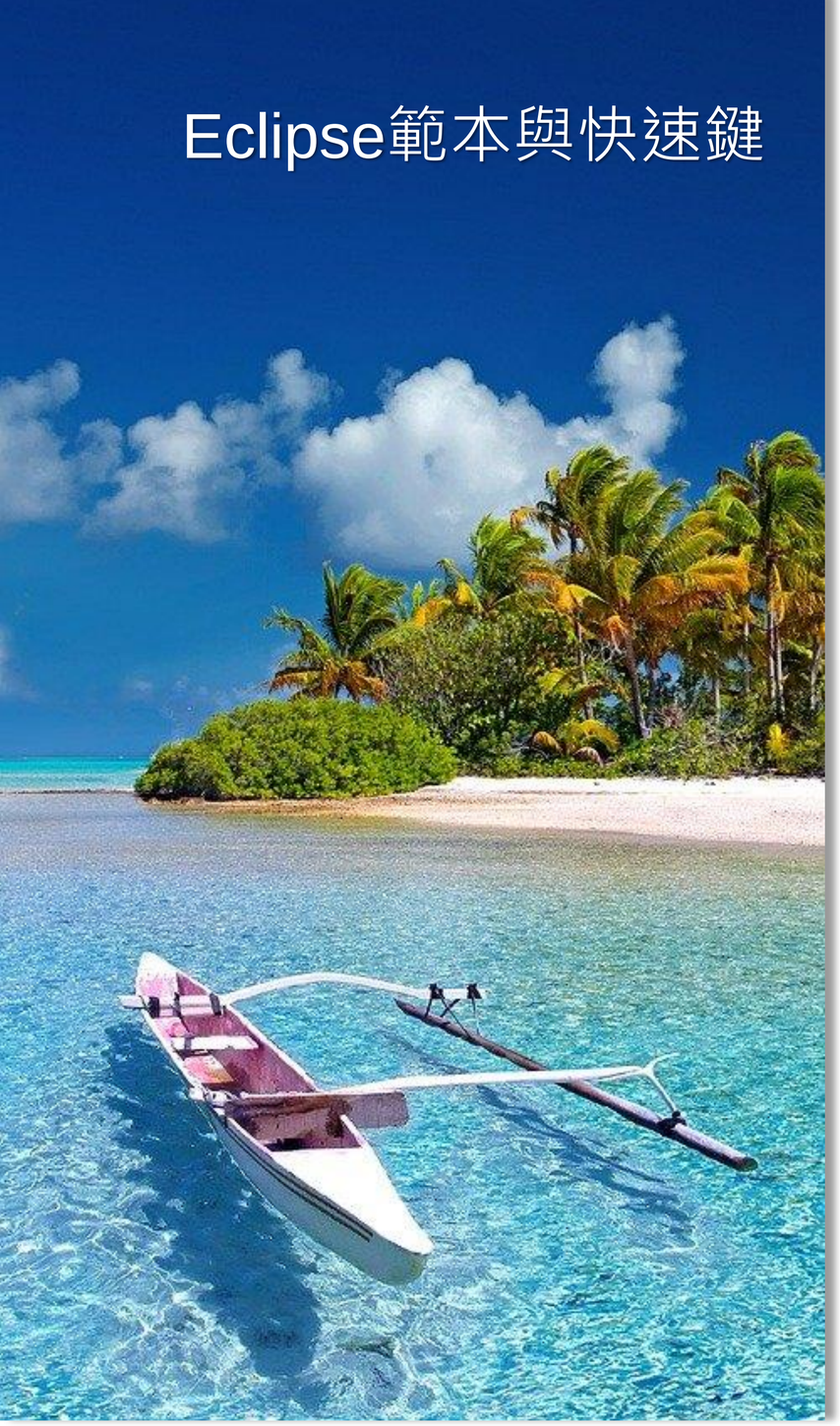
- 整合開發環境
  - 輔助程式開發人員開發程式的應用軟體
  - 提供編輯、編譯、測試執行、除錯及布署等功能
  - 提供圖形介面開發工具及資料庫伺服器管理
  - 仍需安裝 JDK
- 常見的 Java 整合開發環境
  - NetBeans IDE
  - Eclipse
  - IntelliJ IDEA / Android Studio

Eclipse快速鍵



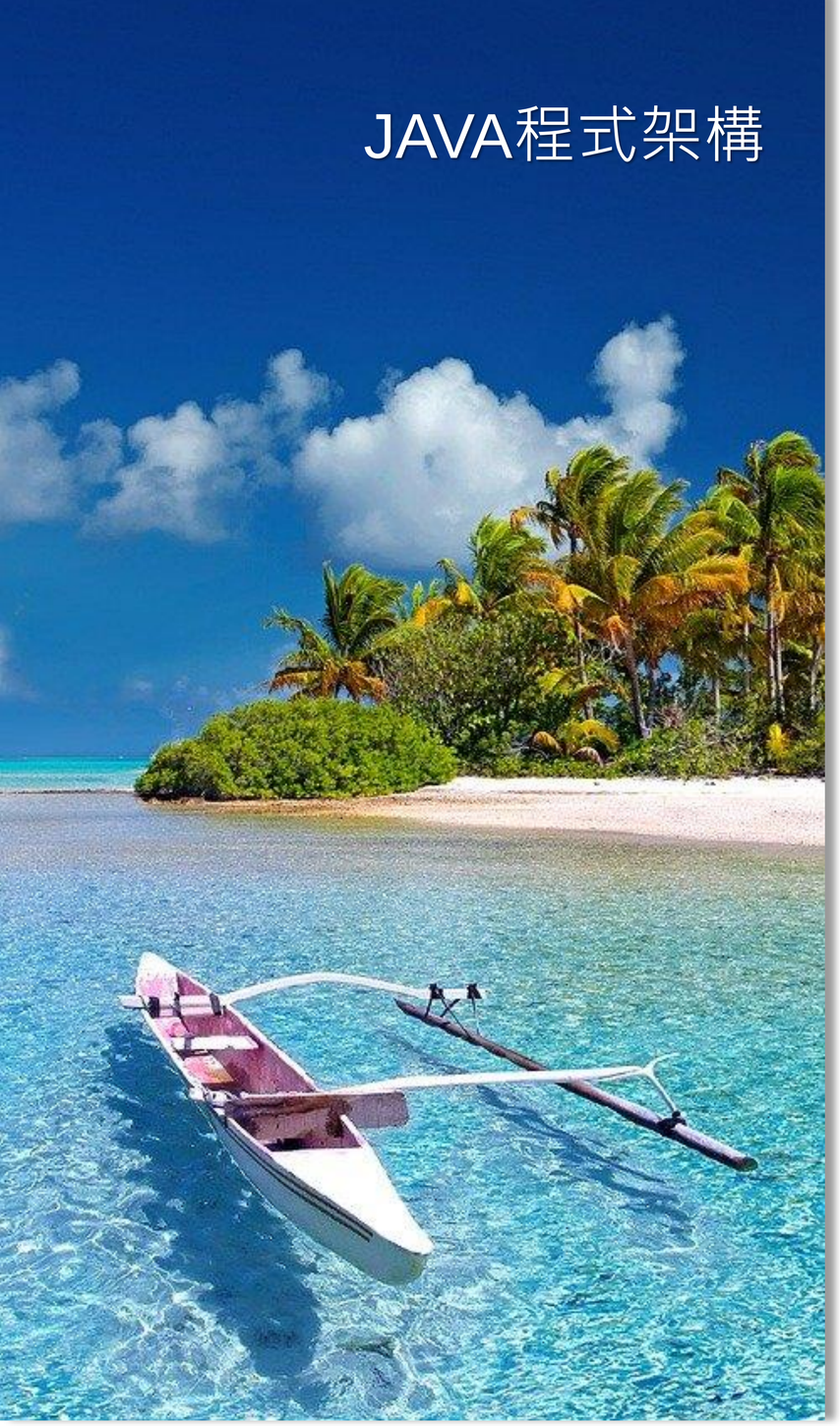
- Ctrl + D : deletes the current line. (刪除整行)
- Alt + Up: Moves up the current line (移動整行)
- Alt + Down: Moves down the current line
- Ctrl + Alt + Up: Copies and moves up the current (複製整行)
- Ctrl + Alt + Down: Copies and moves down the current
- Ctrl+Shift+F:Auto Format(自動格式化)
- Ctrl+”+”或 “-”:ZoomInOut(縮放程式碼)
- Ctrl+Shift+O:import relative package(自動載入相關套件)
- Ctrl+/:Comment(註解/取消註解)
- Ctrl+Shift+/:區塊註解 Ctrl+Shift+\取消區塊註解
- Shift+Alt+A:MutiColumn Edit(多欄編輯)
- Ctrl+F11: Run Project(編譯執行)

Eclipse範本與快速鍵

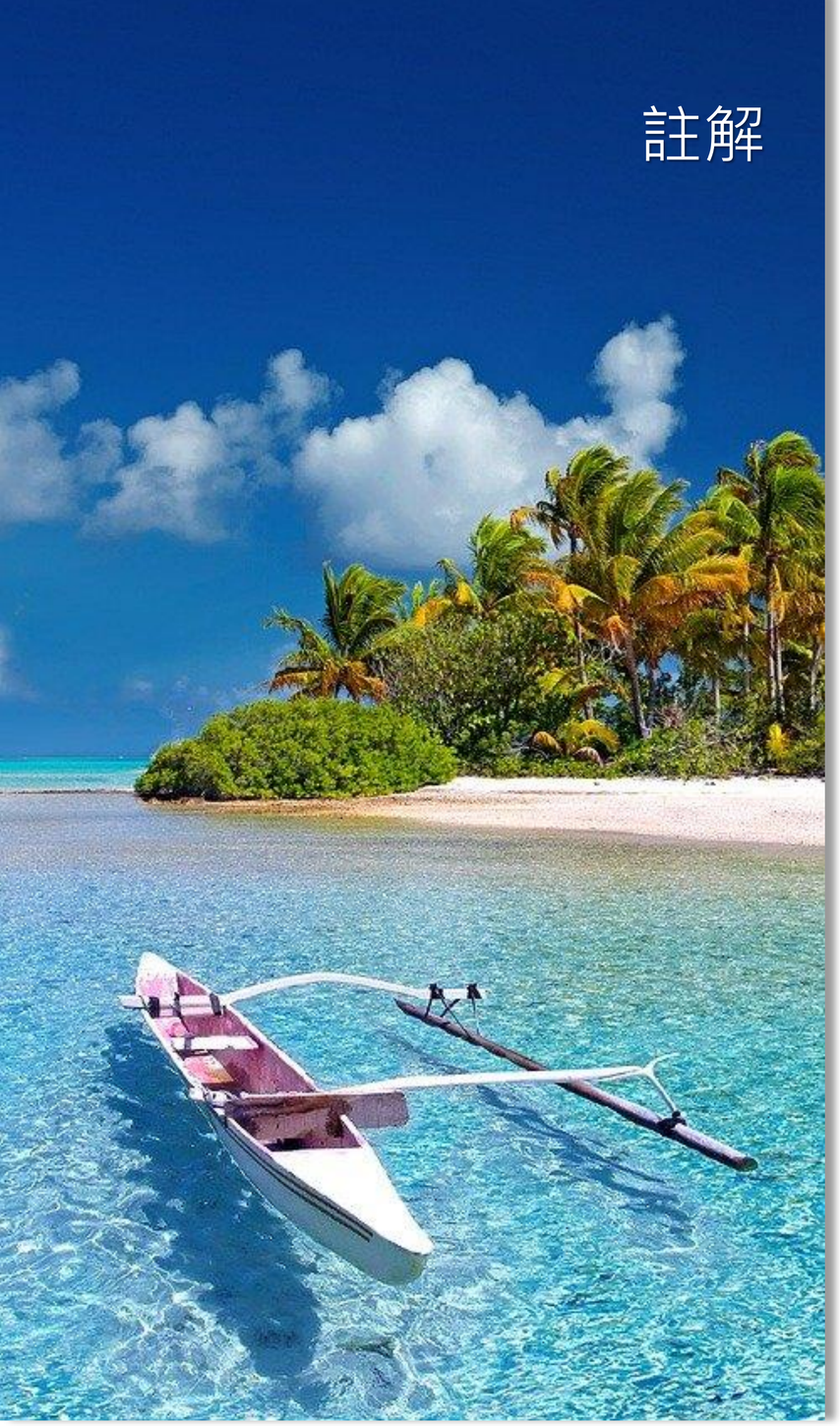


- 自訂縮寫
  - Windows>Preference>Java>Editor>Templates>New>更改完成後按Apply鈕及OK鈕即可
- 自訂快速鍵
  - Windows>Preference>General>Keys

JAVA程式架構



```
/*
 * 檔名：MainClass.java
 * 用途：認識 Java 程式結構
 */
package com.sample;
import java.lang.*;
//主類別
public class MainClass {
   /** 主方法，程式進入點 **/
   public static void main(String[] args) {
      System.out.println("Hello!World!");
   }
}
```



註解

多行註解
不可以寫成巢狀註解

單行註解

文件註解

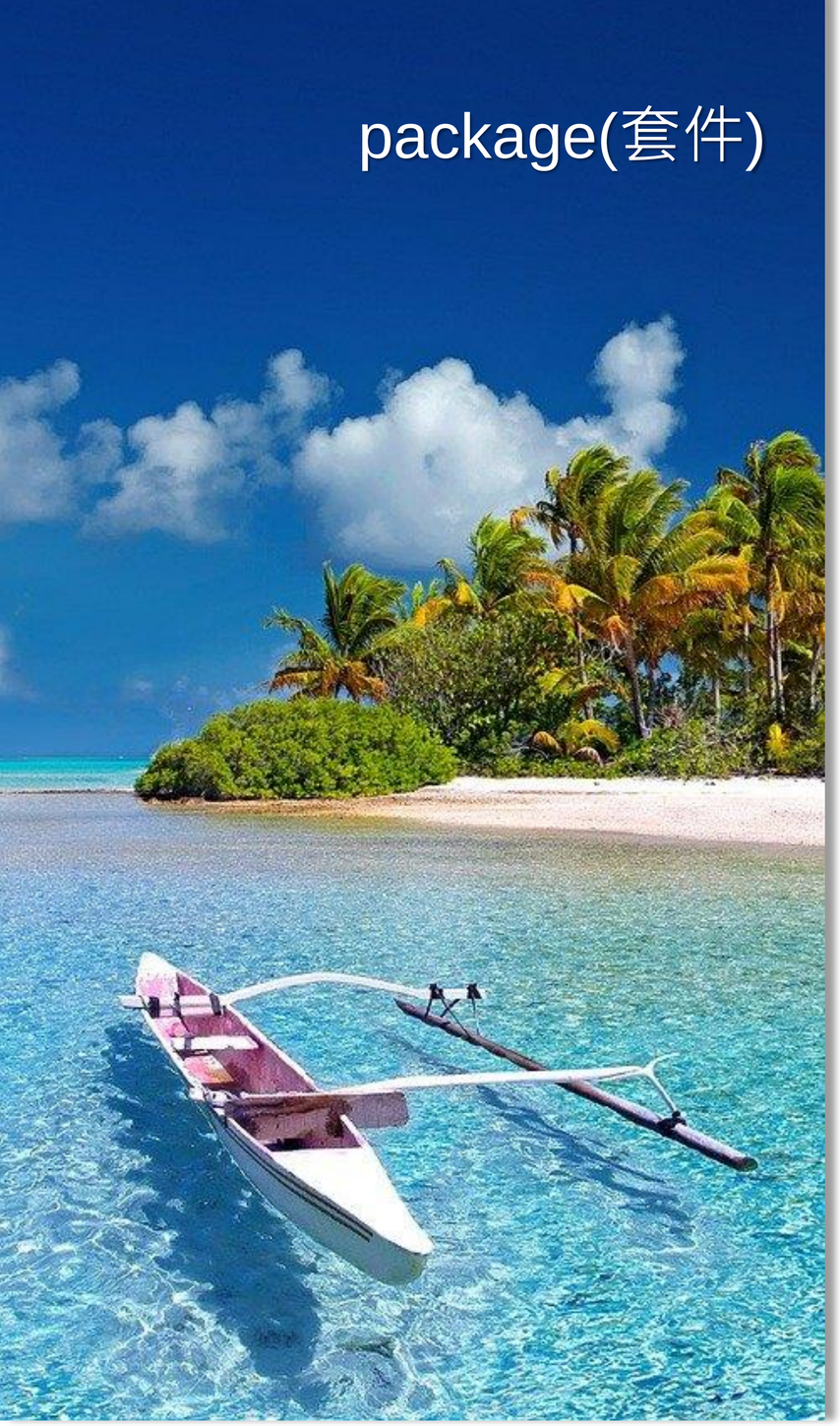
註釋註解@annotation
@FunctionalInterface

```
/*
 * 檔名：MainClass.java
 * 用途：認識 Java 程式結構
 */
package com.sample;

import java.lang.*;

//主類別
public class MainClass {
    /** 主方法，程式進入點 */
    public static void main(String[] args) {
        System.out.println("Hello!World!");
    }
}
```

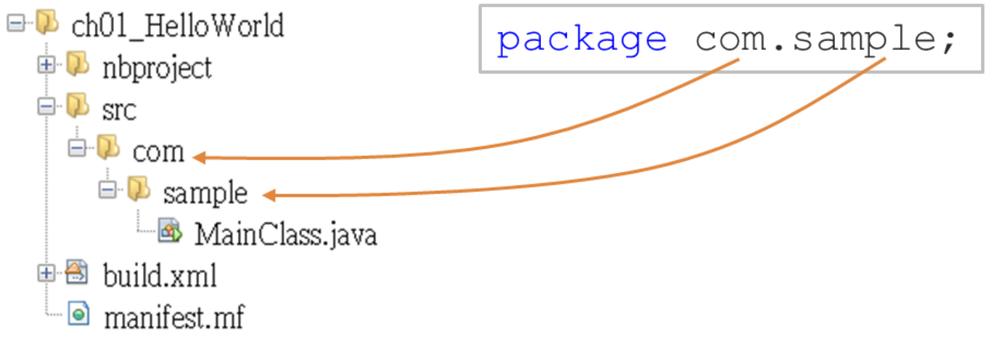
javadoc用來產生說明文件

語法 Javadoc –d 資料夾名稱 檔案名稱.java –private –encoding utf8

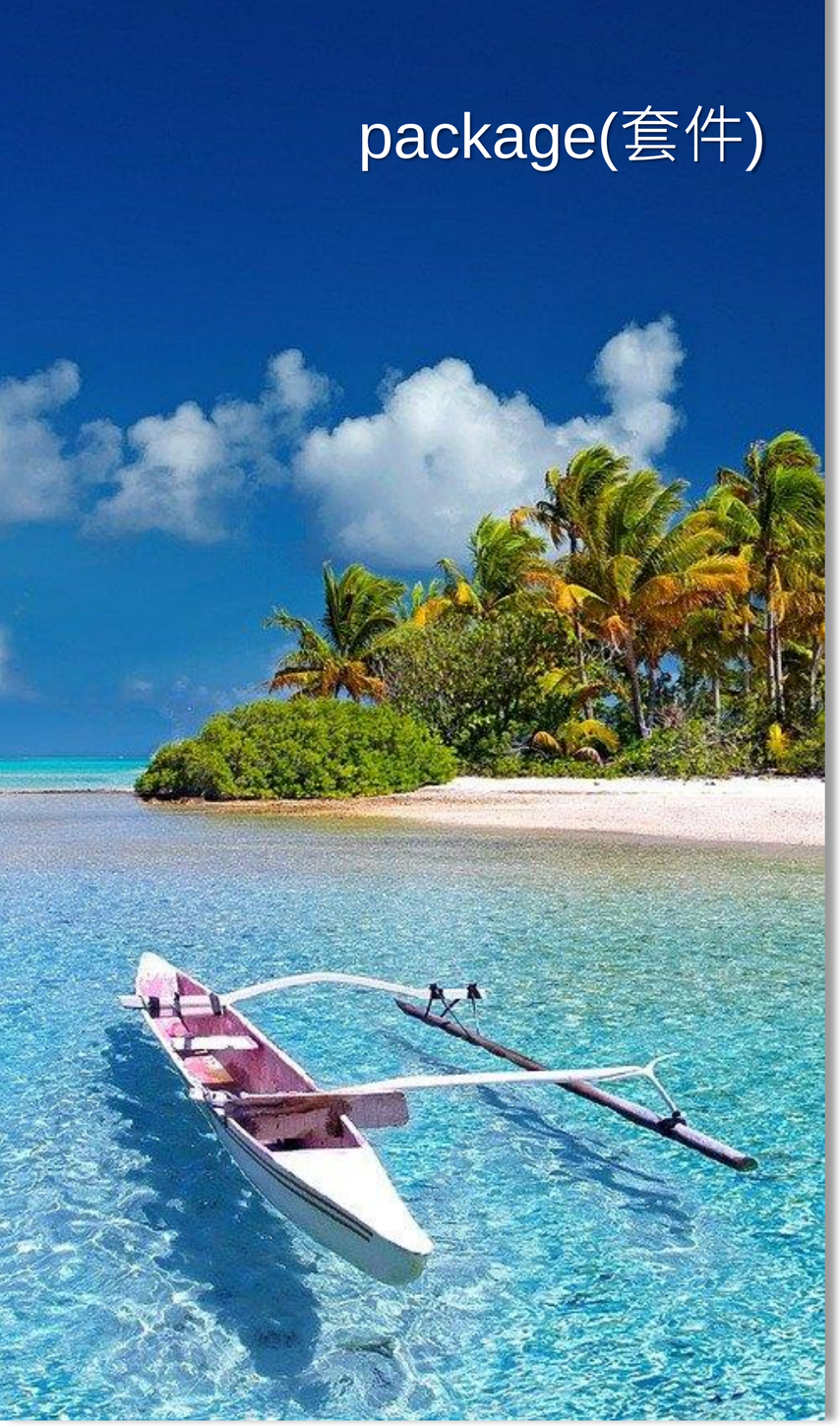
package(套件)



- Java 使用 package 來管理類別，用來宣告該檔案中的類別是屬於哪一個 package，必須寫在程式的第一行。
- package 套件的敘述可以和檔案系統結構相對應。

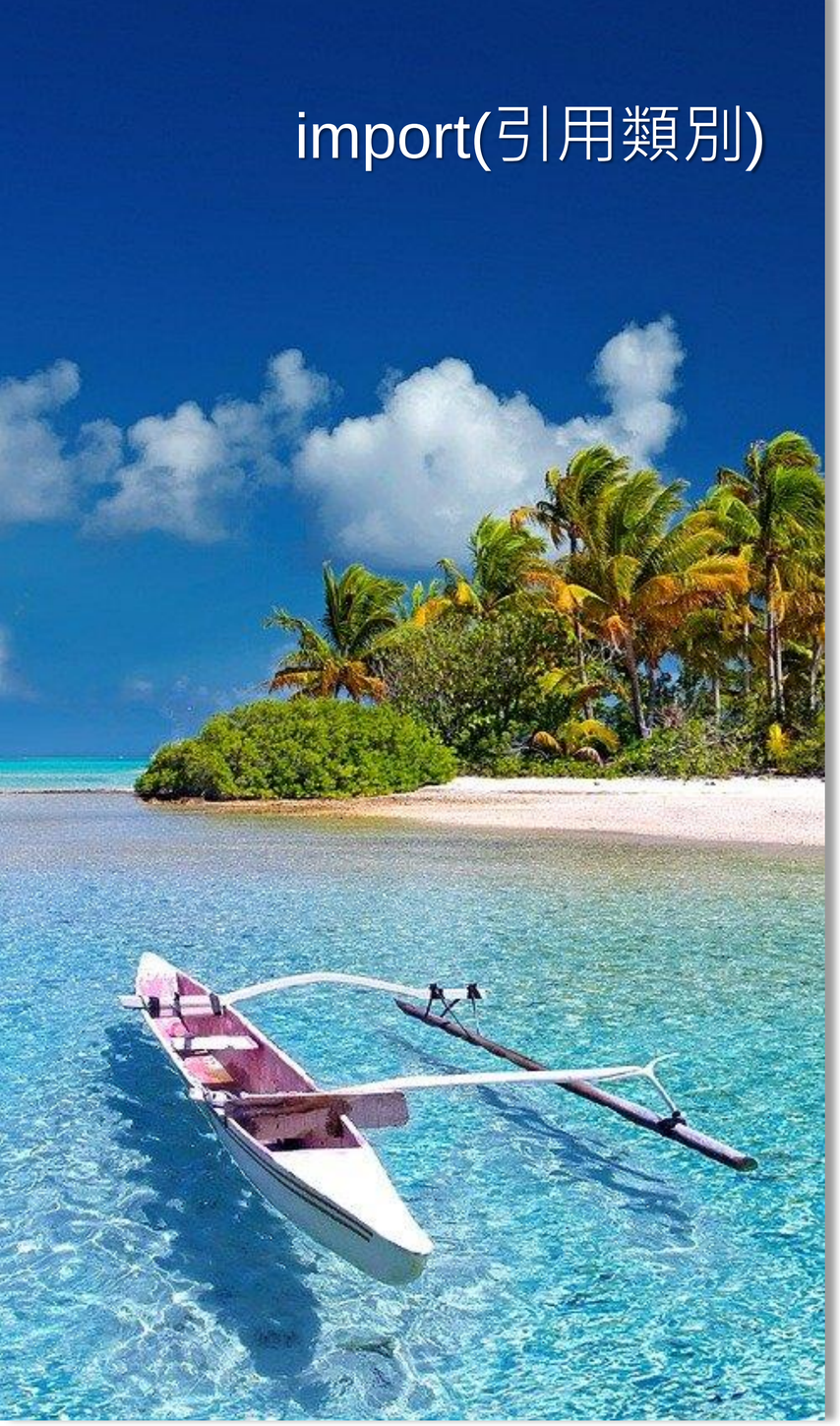


package(套件)



- 含package路徑之編譯與執行
  - javac –d . 主檔名.java 在當前路徑上產生package相關資料夾及子資料夾,並將編譯完成之class檔儲存於此資料夾內.若有多個.java檔要同時編譯,則語法如下:
    - javac –d . *.java
  - 執行含有package之class檔
    - java package.主檔名

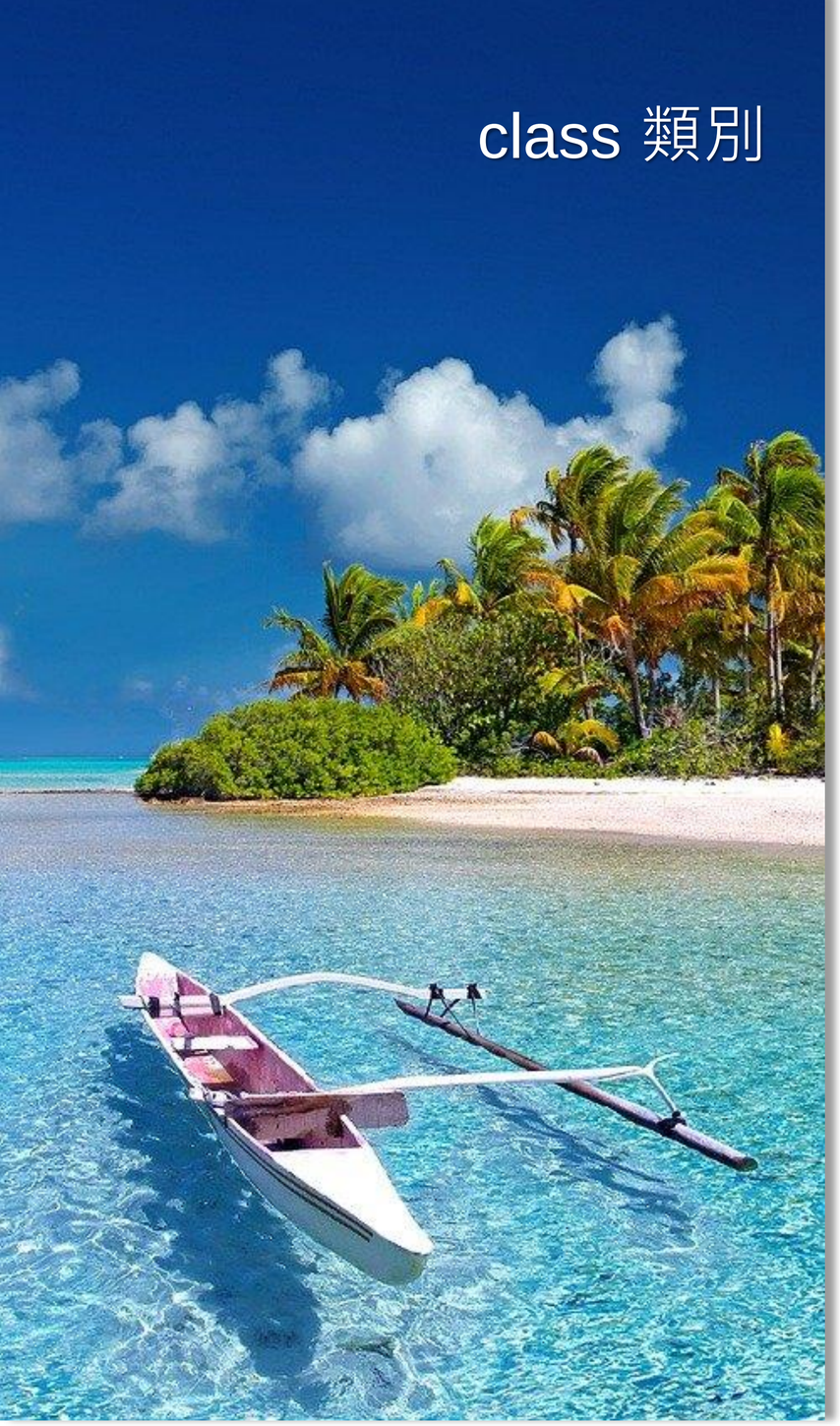
import(引用類別)



關鍵字 import (引用類別)

```java
/*
 * 檔名：MainClass.java
 * 用途：認識 Java 程式結構
 */
package com.sample;

import java.lang.*;

//主類別
public class MainClass {
    /** 主方法，程式進入點 **/
    public static void main(String[] args) {
        System.out.println("Hello!World!");
    }
}
```

- 引用現成 API 或廠商或自己開發的類別時，可以先在 import 區宣告，告知編譯器要使用的類別是位於哪一個套件，讓編譯器找到這些類別所在的位置。
- Java.lang 是預設引用的套件，日後使用此套件下的類別可以直接使用，因此可以省略 import 的敘述。
- import 敘述寫在 package 後方，類別宣告的前方，可以有多個。



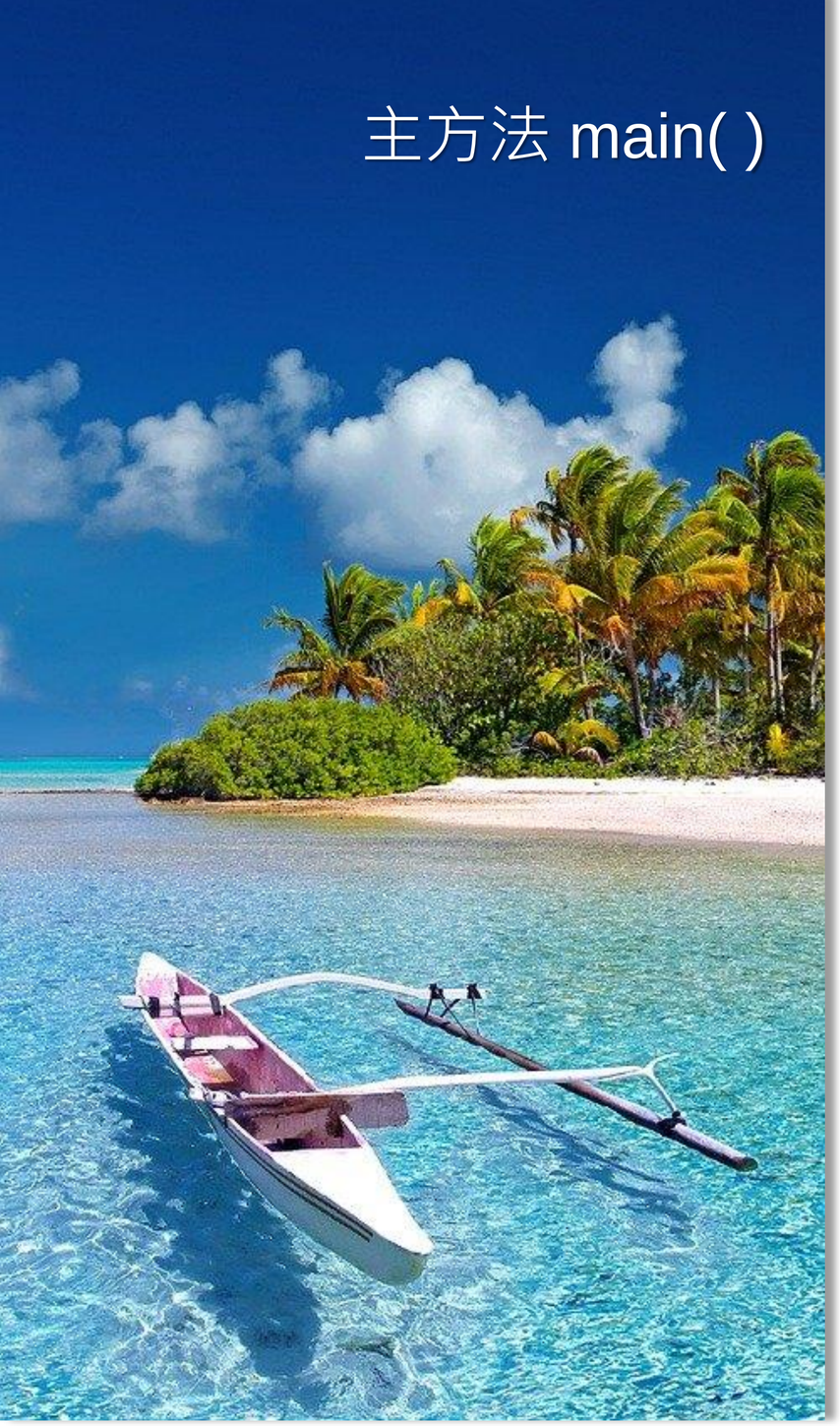
class 類別

類別 class

```java
/*
 * 檔名：MainClass.java
 * 用途：認識 Java 程式結構
 */
package com.sample;

import java.lang.*;

//主類別
public class MainClass {
    /** 主方法，程式進入點 **/
    public static void main(String[] args) {
        System.out.println("Hello!World!");
    }
}
```

- Java 程式由類別組成，因此 Java 程式至少須包含一組類別的宣告。
  - public 存取修飾字，表示公開使用
  - class 宣告類別關鍵字
  - MainClass 類別名稱(自訂)
  - [注意]：類別若宣告public，則存檔時的主檔名必須與類別名稱相同
  - 一個 java 檔案可以有多的類別宣告，但只有一個可以宣告 public。

主方法 main( )

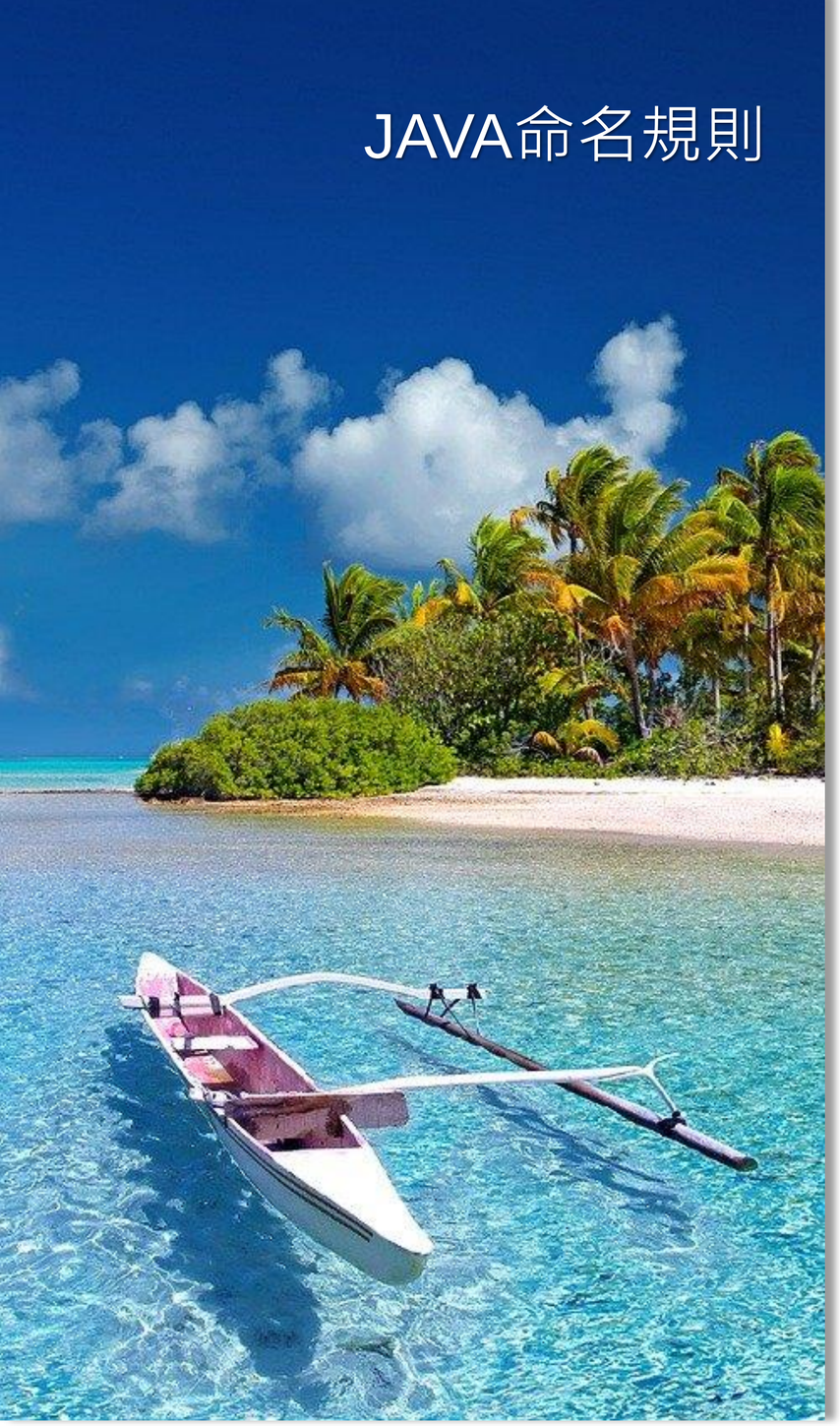


main( ) 程式進入點

```java
/*
 * 檔名：MainClass.java
 * 用途：認識 Java 程式結構
 */
package com.sample;

import java.lang.*;

//主類別
public class MainClass {
    /** 主方法，程式進入點 **/
    public static void main(String[] args) {
        System.out.println("Hello!World!");
    }
}
```

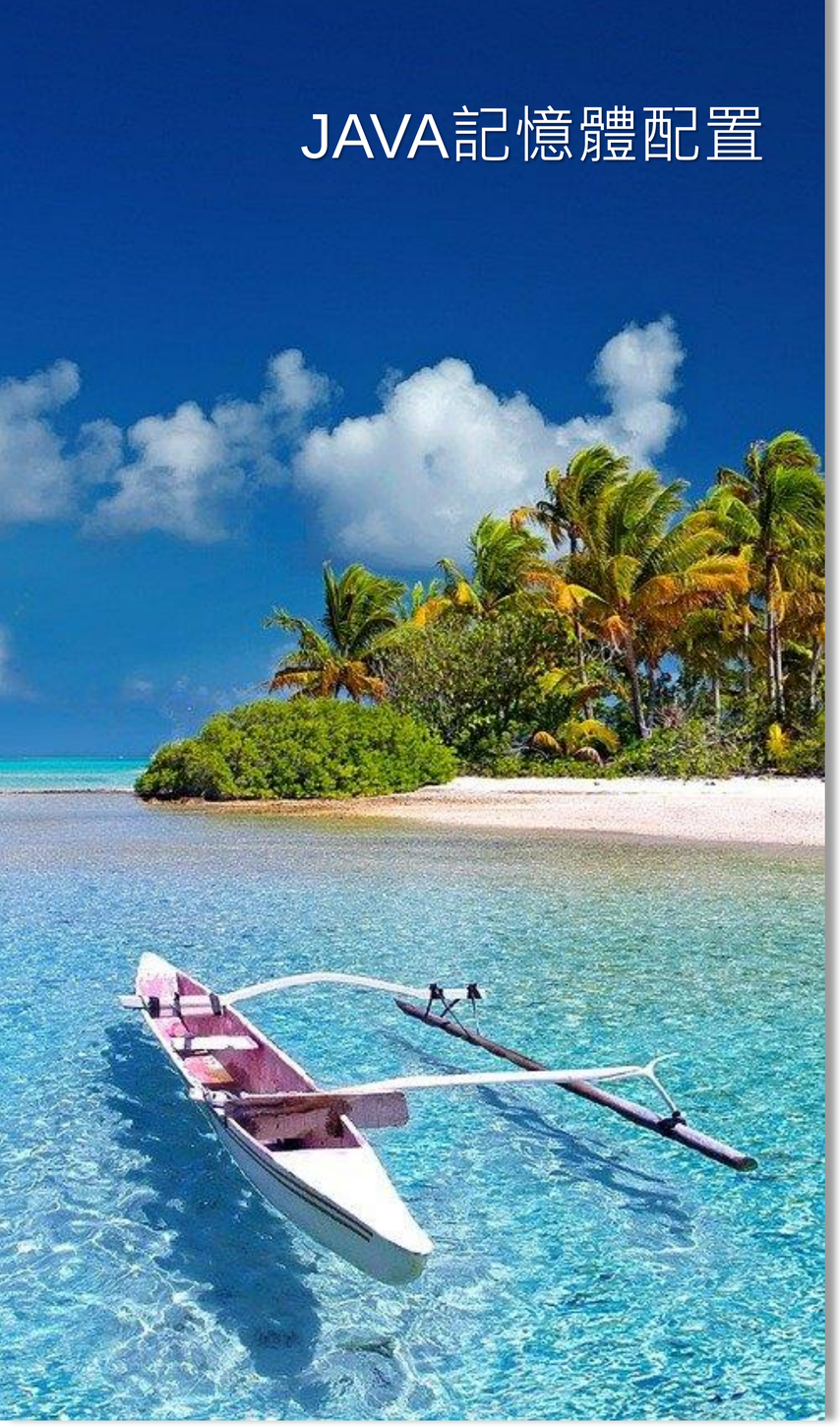
- Java 應用程式執行時的進入點。
  - public 存取修飾字，表示公開使用
  - static 代表 main 方法是類別成員，不需要產生物件就能被執行
  - void表示 main( ) 執行結束後不傳回任何值
  - main 方法名稱(不可以修改)
  - String[ ] args 方法參數列：String[ ] 字串陣列，args 陣列名稱，在此用來接收命令列傳入的參數

JAVA命名規則

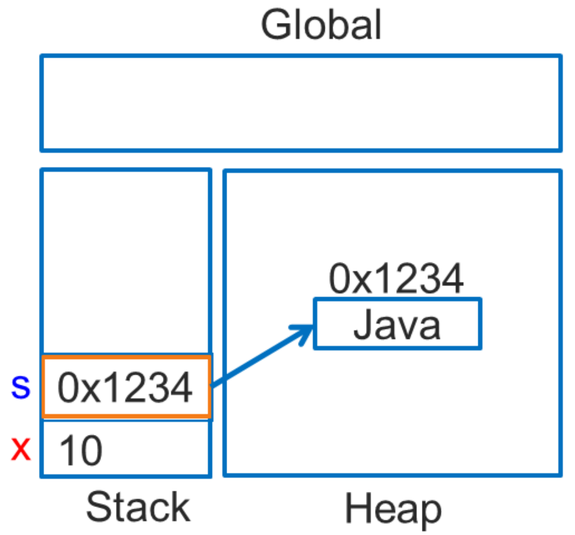


- constant 常數 全部大寫 MAX_VALUE
- package 套件 全部小寫 java.lang
- class 類別 大駝峰 String JOptionPane
- method 方法 variable變數 小駝峰 showInputDialog toUpperCase
- 程式設計時，替套件、類別、方法或變數命名，以便在設計中識別，這些命名稱為識別字
- 識別字第一個符號必須是字母 (A~Z或a~z)、底線 (_) 或錢字號 ($)
- 識別字第二個之後符號可以是字母 (A~Z或a~z)、數字 (0~9)、底線 (_) 或錢字號 ($)

JAVA記憶體配置

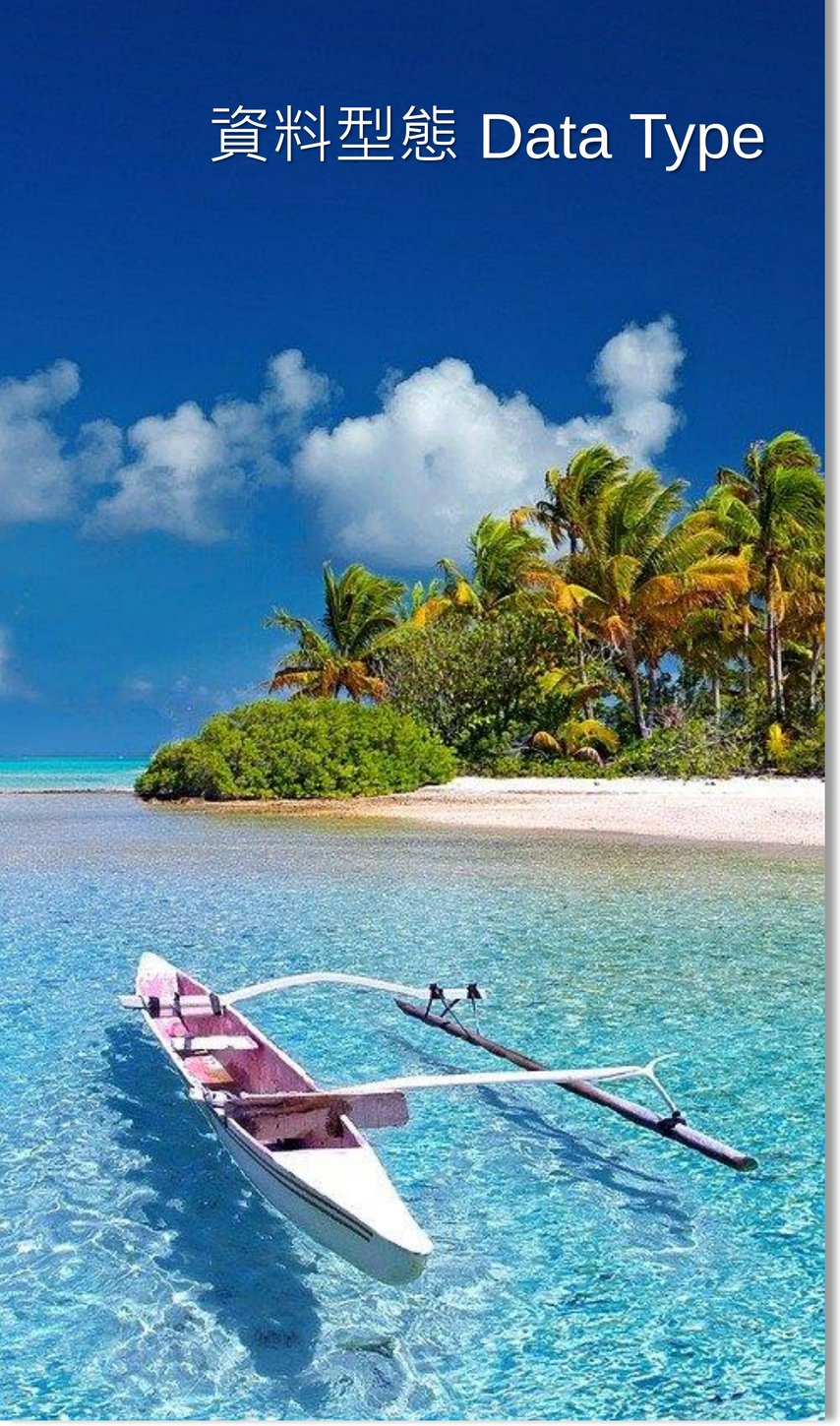


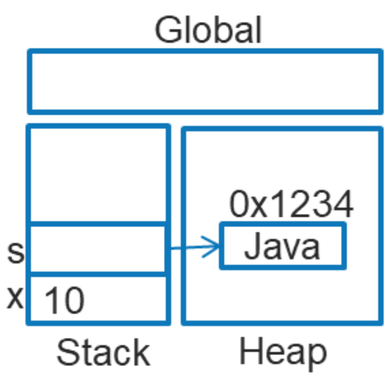
- Global (全域)
  - 放置 static 的類別成員變數
- Stack (堆疊)
  - 放置基本型別的變數內容或參考型別變數的參考內容值
- Heap (堆積)
  - 放置參考型別的變數內容

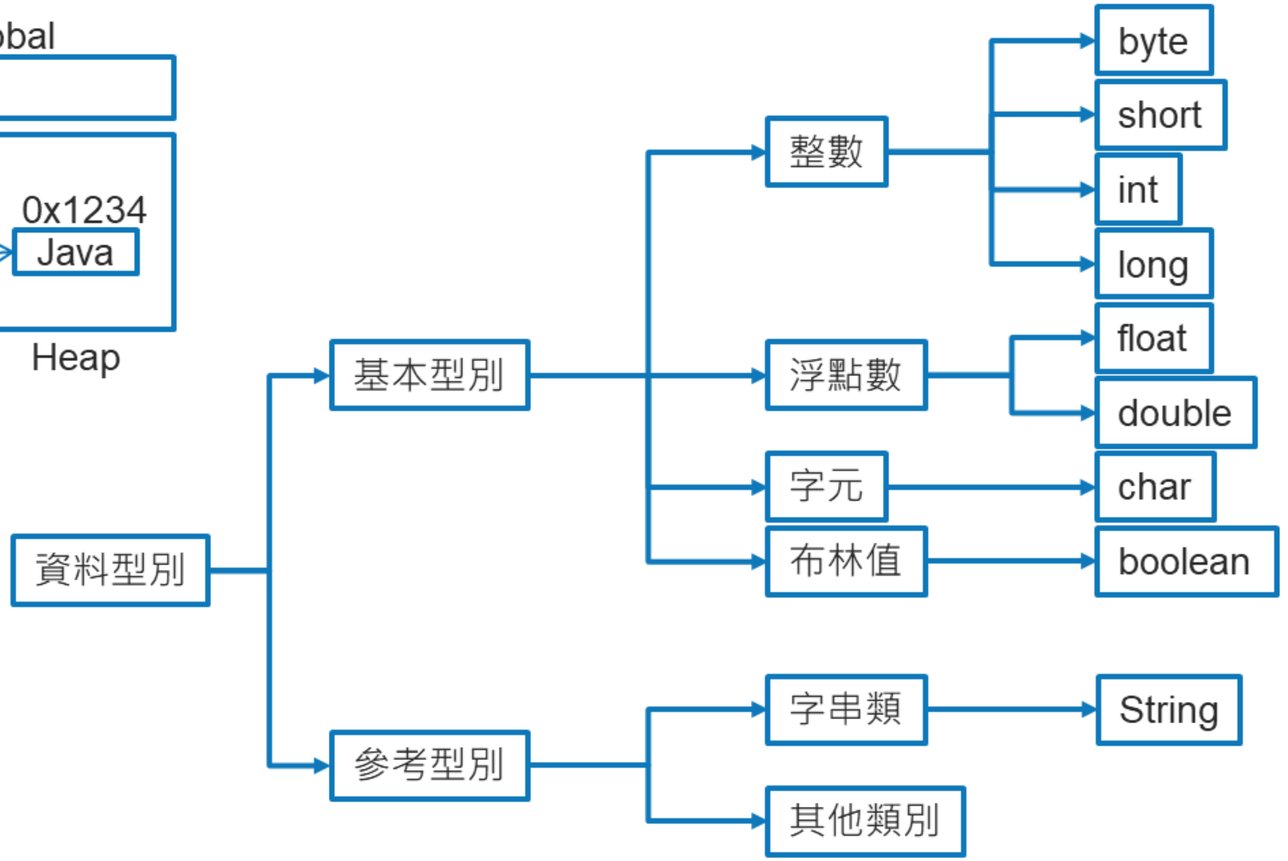



```
int x = 10;
String s = new String("Java");
```

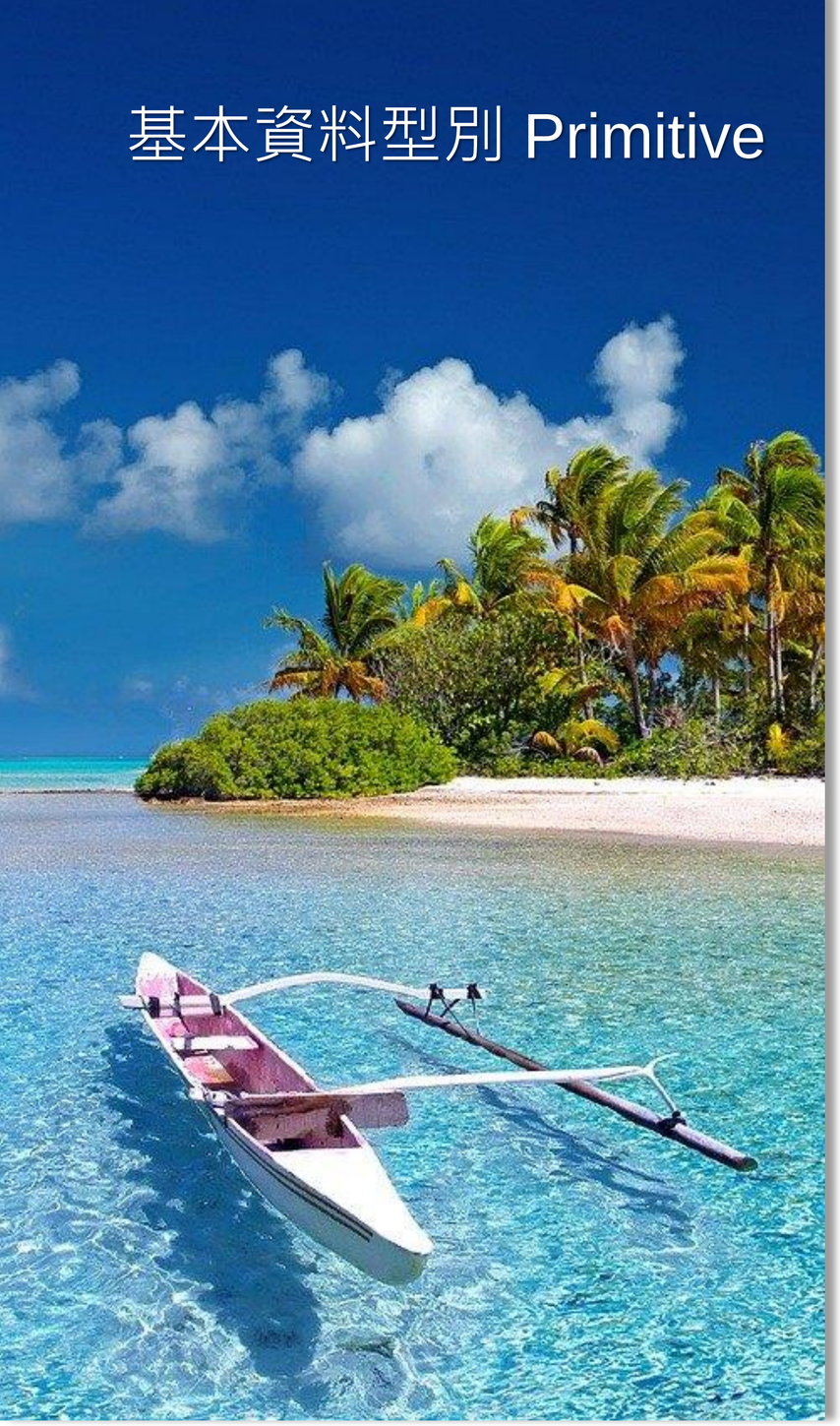
資料型態 Data Type







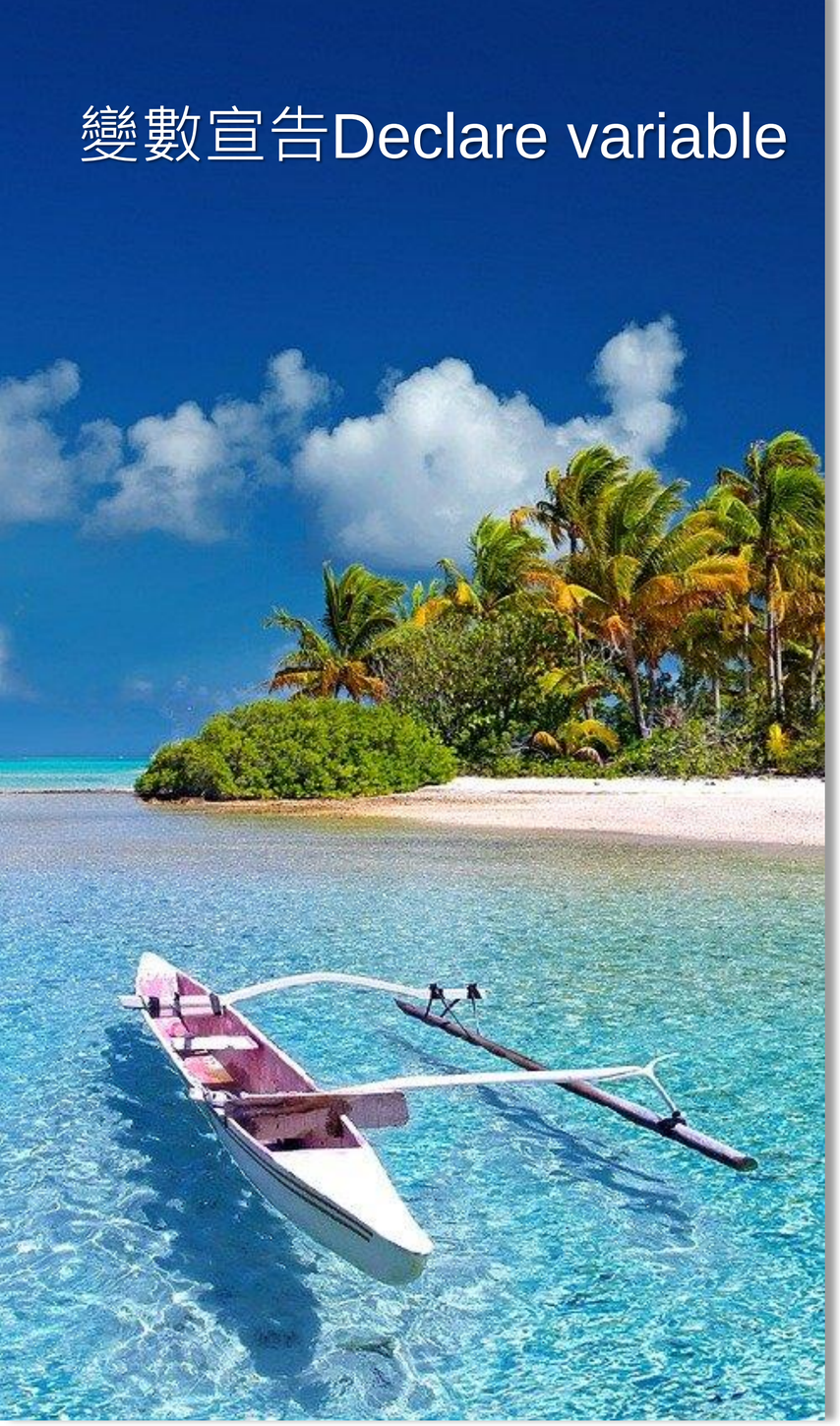
基本資料型別 Primitive



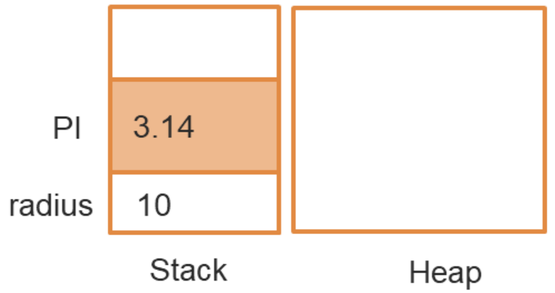
- Java 的基本型別可分為四類：整數、浮點數、布林值、字元

資料別	資料型態	位元組	範圍	初始值
整數	byte	1	-128~127	0
	short	2	-32,768~32,767	0
	int	4	-2^{31}~$2^{31}-1$	0
	long	8	-2^{63}~$2^{63}-1$	0
浮點數	float	4	-3.4E+38~3.4E+38	0.0F
	double	8	-1.7E+308~1.7E+308	0.0
布林值	boolean	VM給定	true或false	false
字元	char	2	'\u0000'~'\uFFFF'	'\u0000'

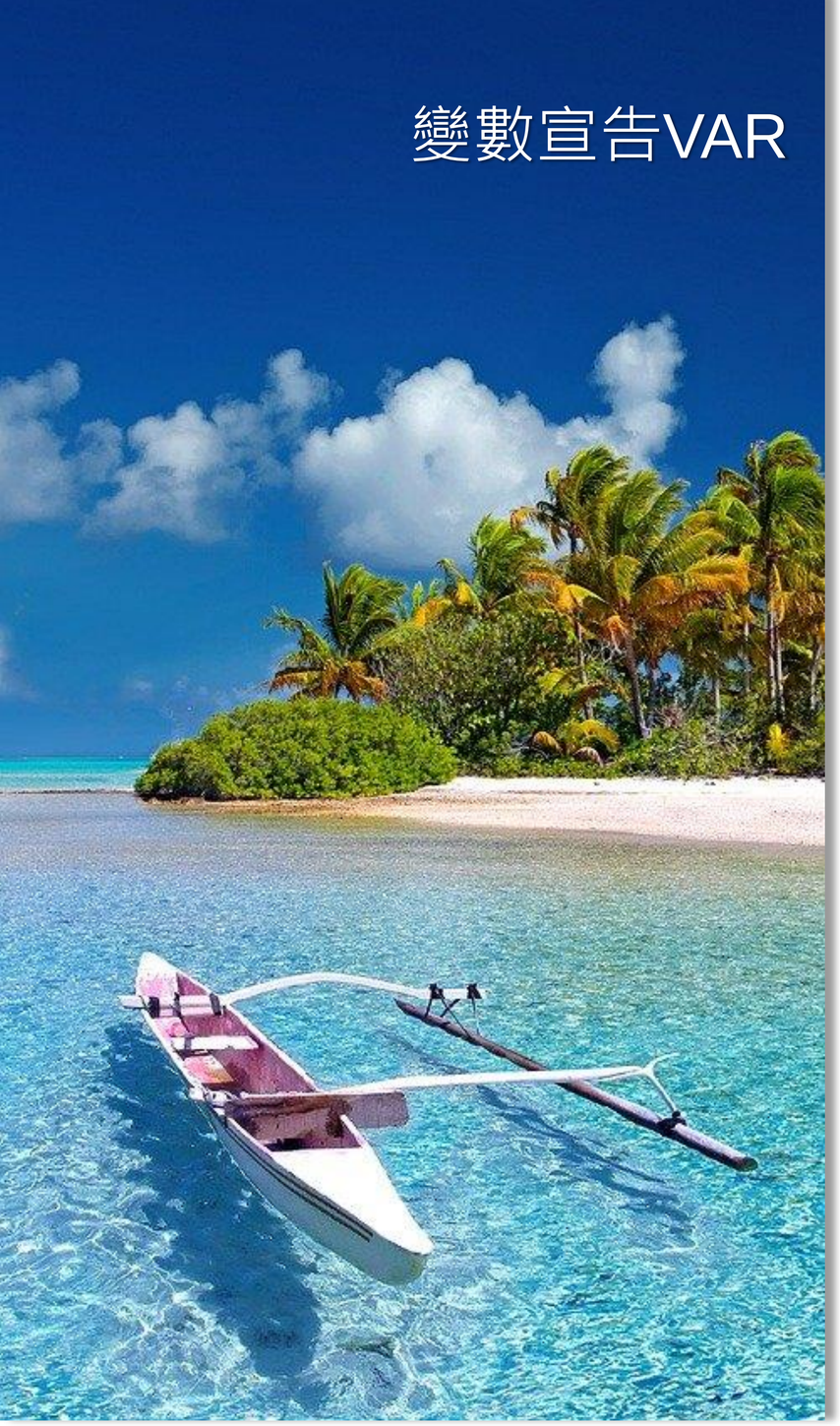
變數宣告Declare variable



- 宣告一個記憶體空間，告知編譯器這個空間將儲存資料的型態，並給予記憶體空間一個名稱，在執行程式時，使用這個名稱，將指定資料存入預先宣告的記憶空間中， 這個名稱與空間叫作變數，而宣告的動作稱為變數宣告。

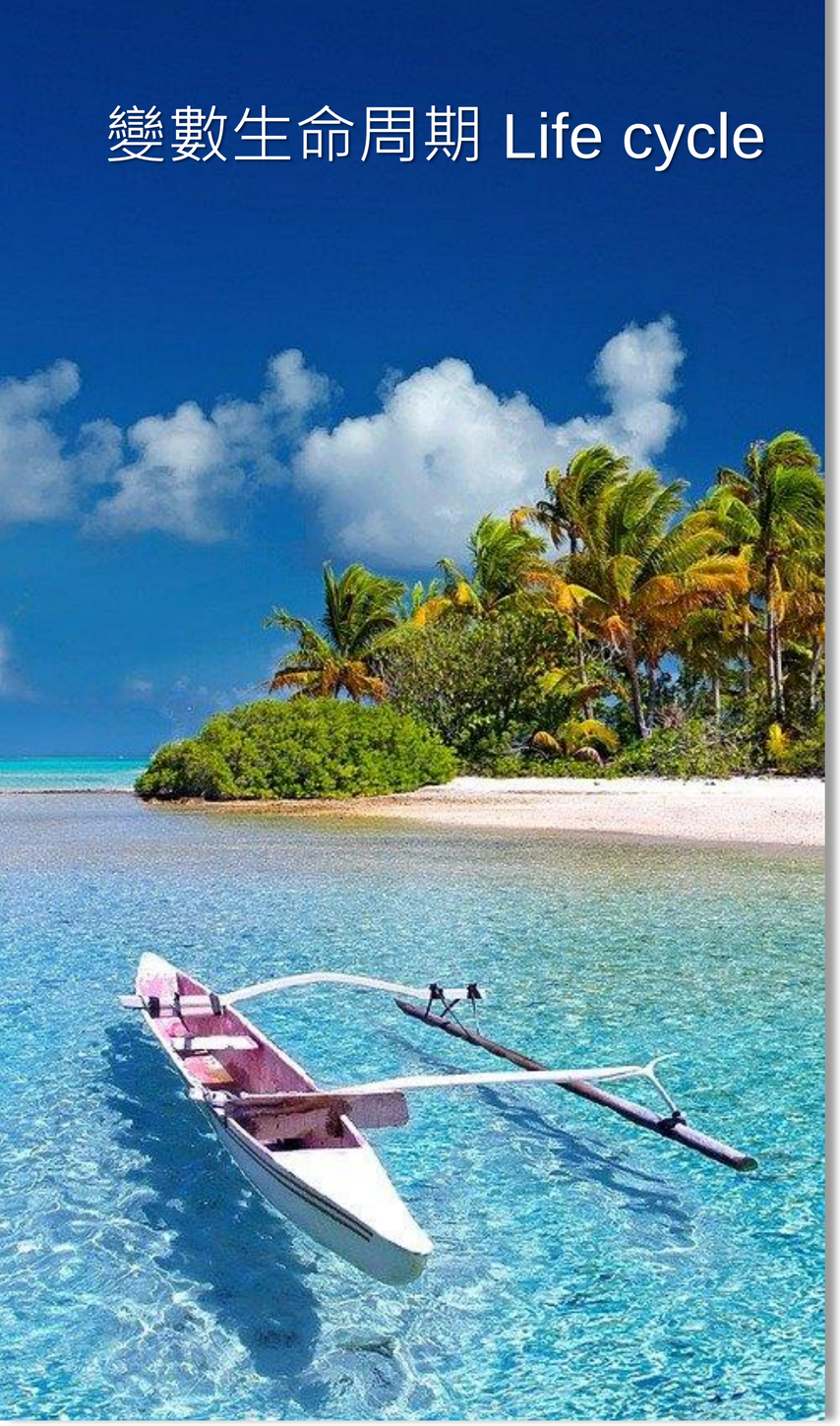


變數宣告VAR



- var 是JDK 10新增宣告方式
- 區域變數類型推定宣告
  - 所以只能在方法內宣告,且一定要給定初值才有辦法類型推定
  - 不能宣告類別變數或物件變數
  - 不能在方法之參數及回傳型態上使用
- 語法 var 變數名稱=初值;
- var a=10; //此時a變數為int資料型態
- var b;//錯誤之宣告方式,無法推定b為何種資料型態

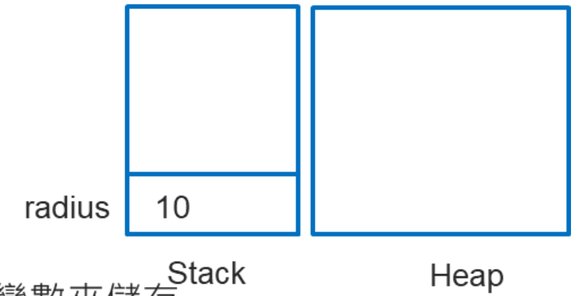
變數生命周期 Life cycle



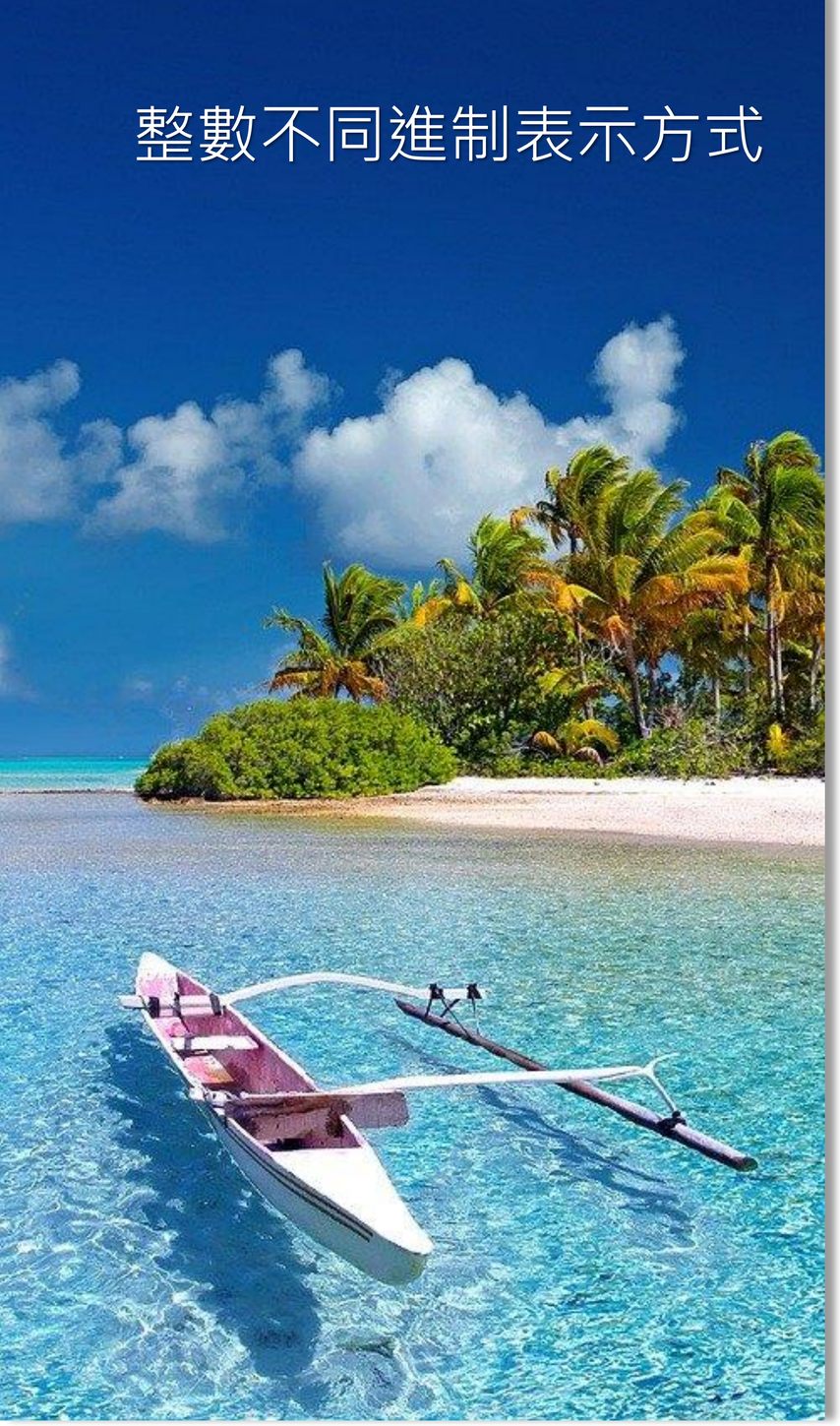
- 變數的類型
  - 全域變數(global)=>在類別宣告的變數,生命週期函蓋整個類別
  - 區域變數(local)=>在方法中宣告的變數,生命週期只存於此方法中
  - 區塊變數(block)=>在迴圈中宣告的變數,生命週期只存於此迴圈中
- 語法

```
int radius = 10;
```

  - 資料型別 變數名稱 [= 初始值];
    - 資料型別：基本型別與參考型別
    - = 是指定運算子，用來將右邊的值指定給左邊的變數來儲存



整數不同進制表示方式



- 十進位 (0~9)

```
System.out.println(12345); //顯示12345
```

- 二進位 (0b 開頭， 0~1)

```
System.out.println(0b11000000111001); //顯示12345
```

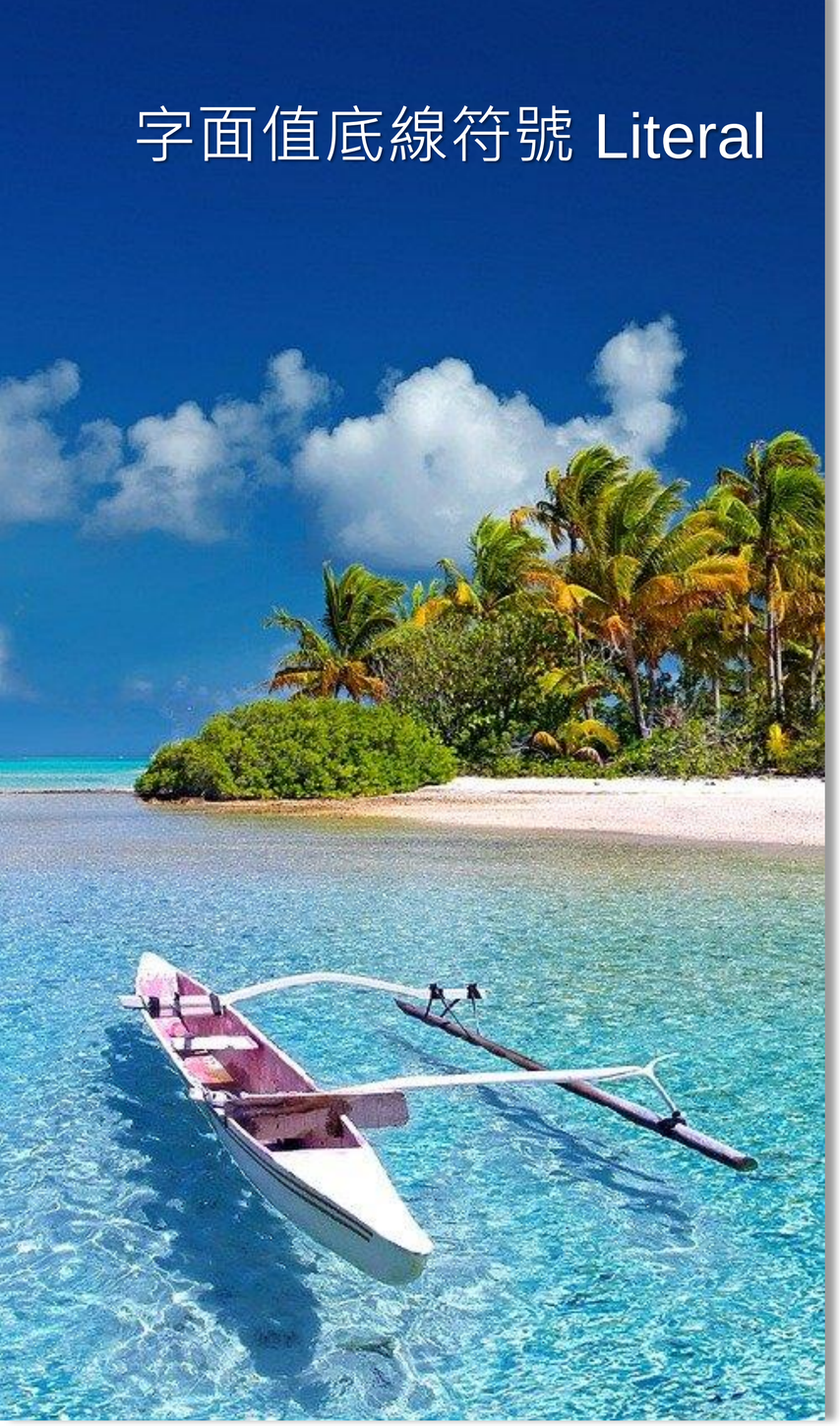
- 八進位 (0 開頭， 0~7)

```
System.out.println(030071); //顯示12345
```

- 十六進位 (0x 開頭， 0~9, 10~15 以 a~f 表示)

```
System.out.println(0x3039); //顯示12345
```

字面值底線符號 Literal



- _(底線) 用於程式撰寫階段讓字面值容易閱讀，程式編譯時會將底線去除
  - 例如：

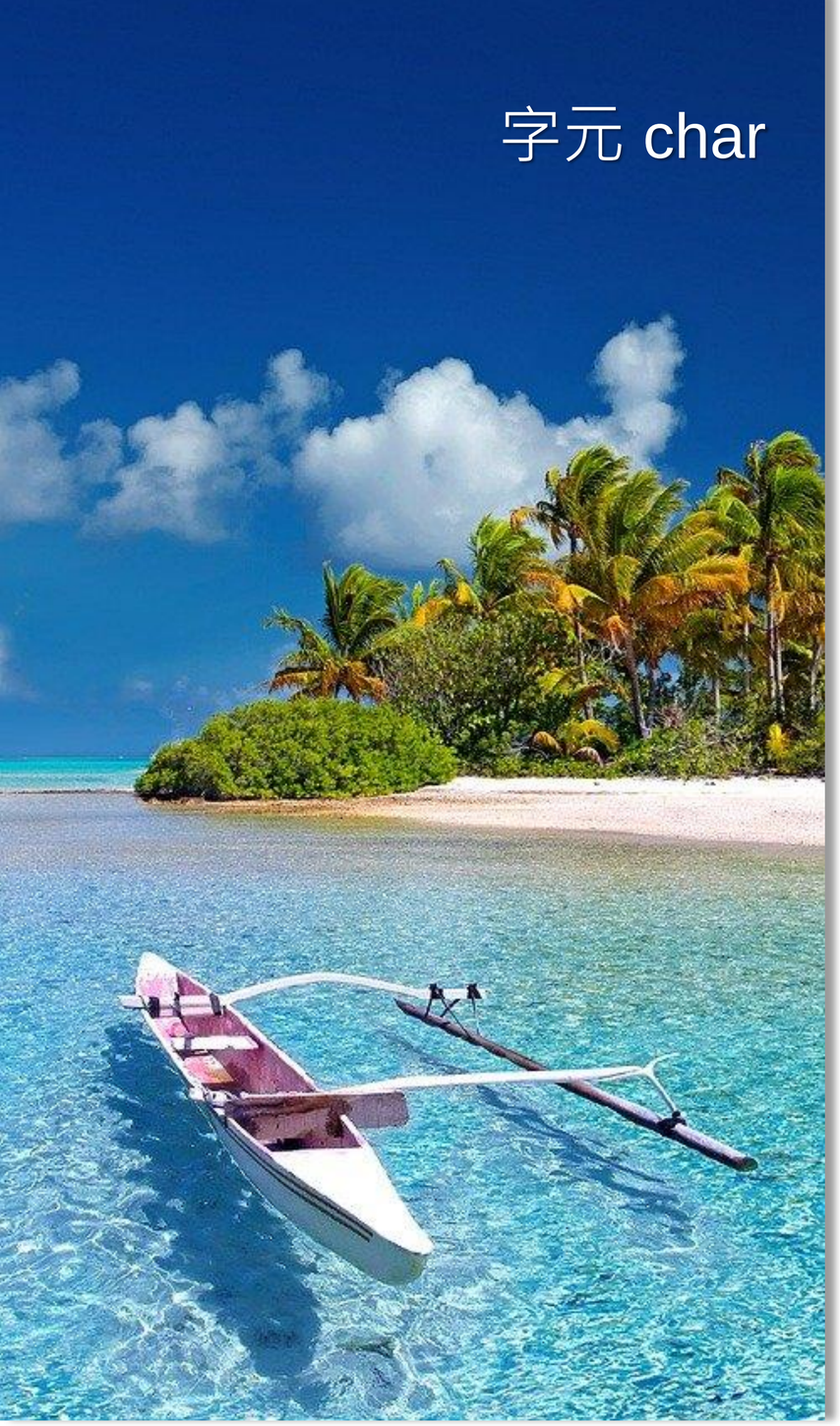
```
System.out.println(0b11000000111001); //顯示12345
```

  - 結合底線：

```
System.out.println(0b11_0000_0011_1001); //顯示12345
```

- 注意，底線前後必須是數字

字元 char



- char 表示單一個字元符號，字元資料須使用單引號『''』括住符號
- Java 字元 char 是以 16 位元 Unicode* 方式編碼
- 使用單個字元來指定字元型態的值

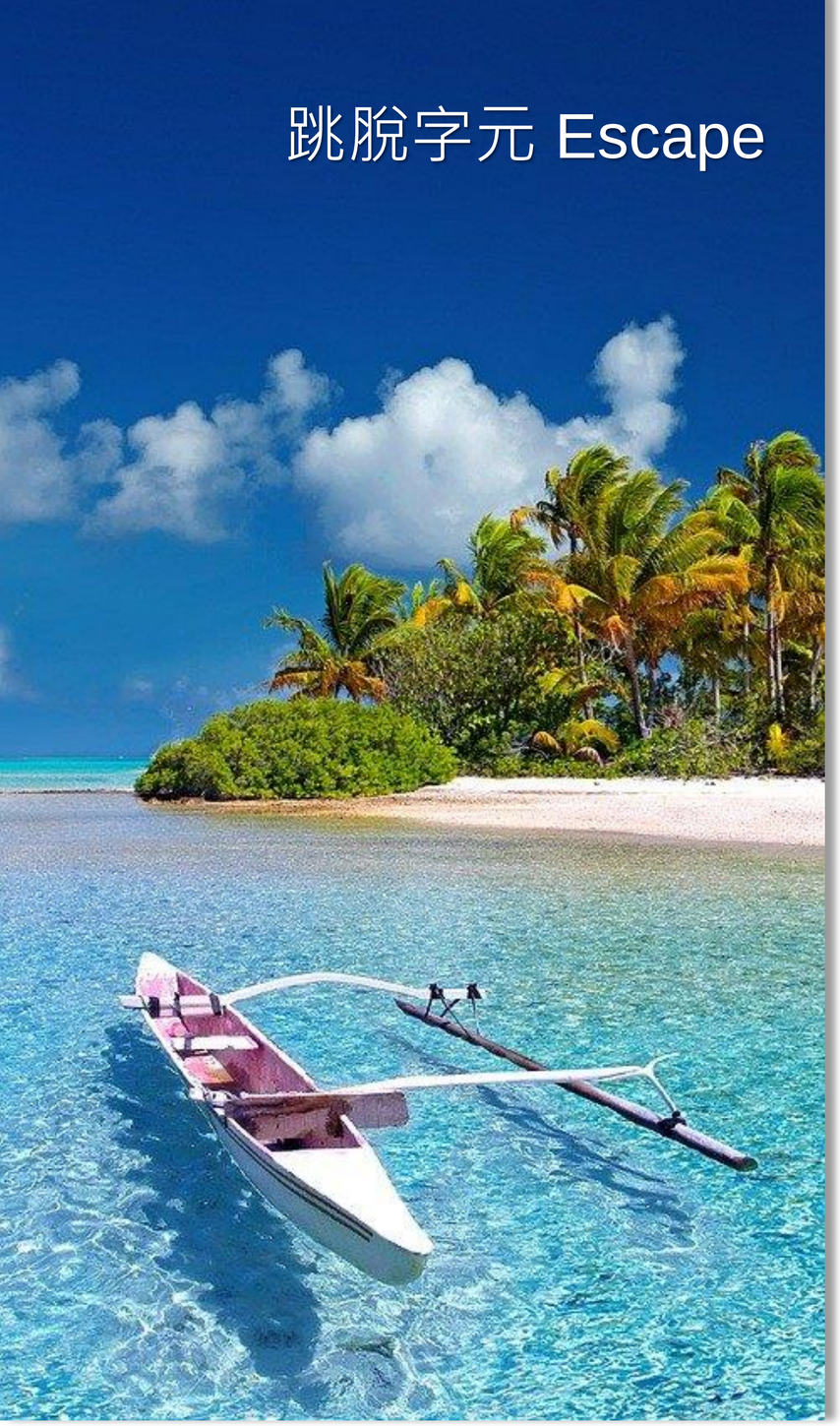
```
System.out.println('A');  //顯示 A
System.out.println('熱');  //顯示 熱
```

- 使用 Unicode 值表示字元型態的值

```
System.out.println('\u0041');  //顯示 A
```

* Unicode 編碼儲存現今世界的各種語言中大部份必須顯示的字元

跳脫字元 Escape

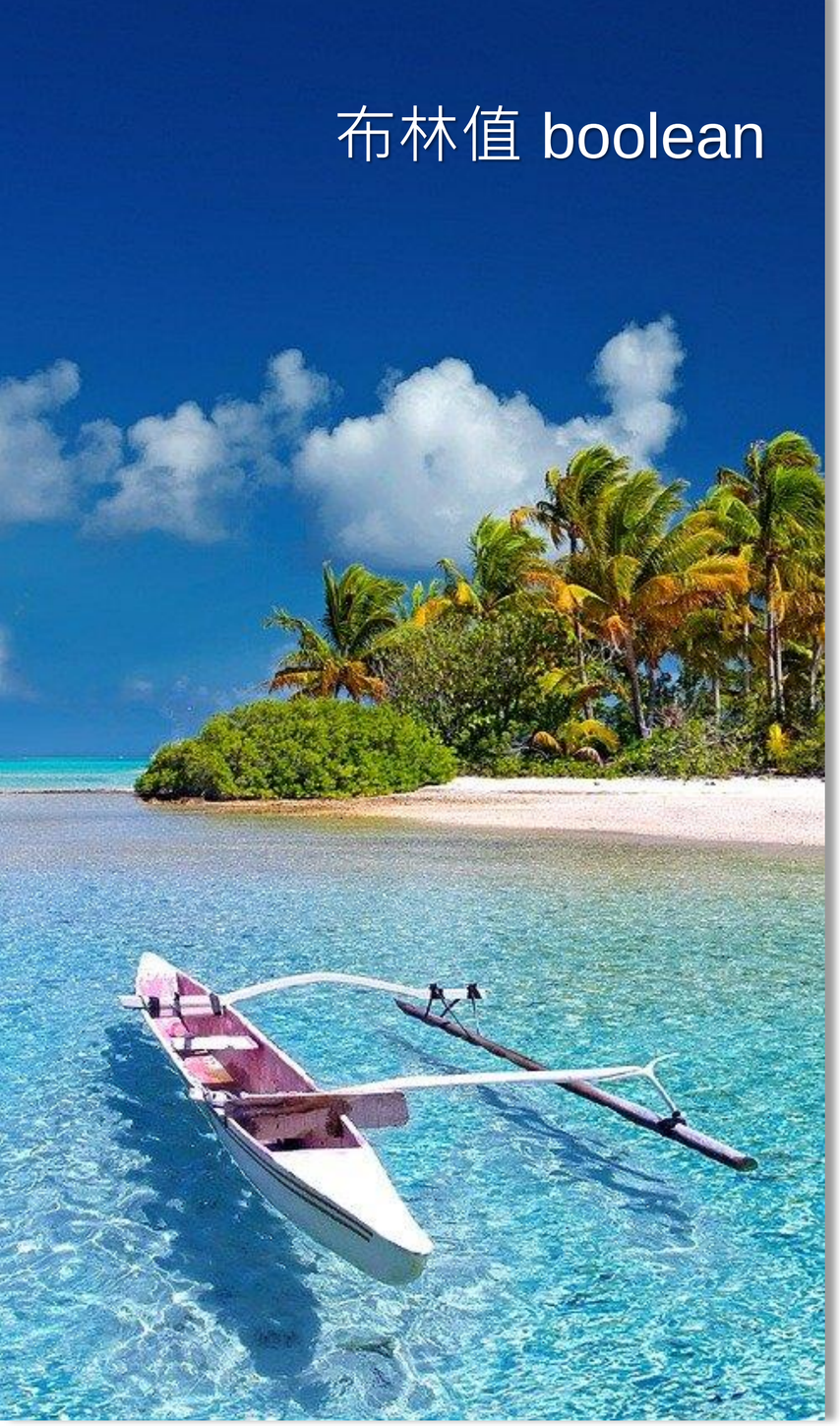


- 使用跳脫字元表示特殊字元型態的值

跳脫字元	解釋	Unicode表示
\b	退格鍵字元	\u0008
\n	換行字元	\u000a
\r	輸入字元	\u000d
\t	Tab鍵字元	\u0009
\”	取得雙引號	\u0022
\’	取得單引號	\u0027
\\	取得反斜線	\u005c

```
System.out.println("Java\t8.0\nAndroid\t7.1.1");
//顯示 Java     8.0
       Android  7.1.1
```

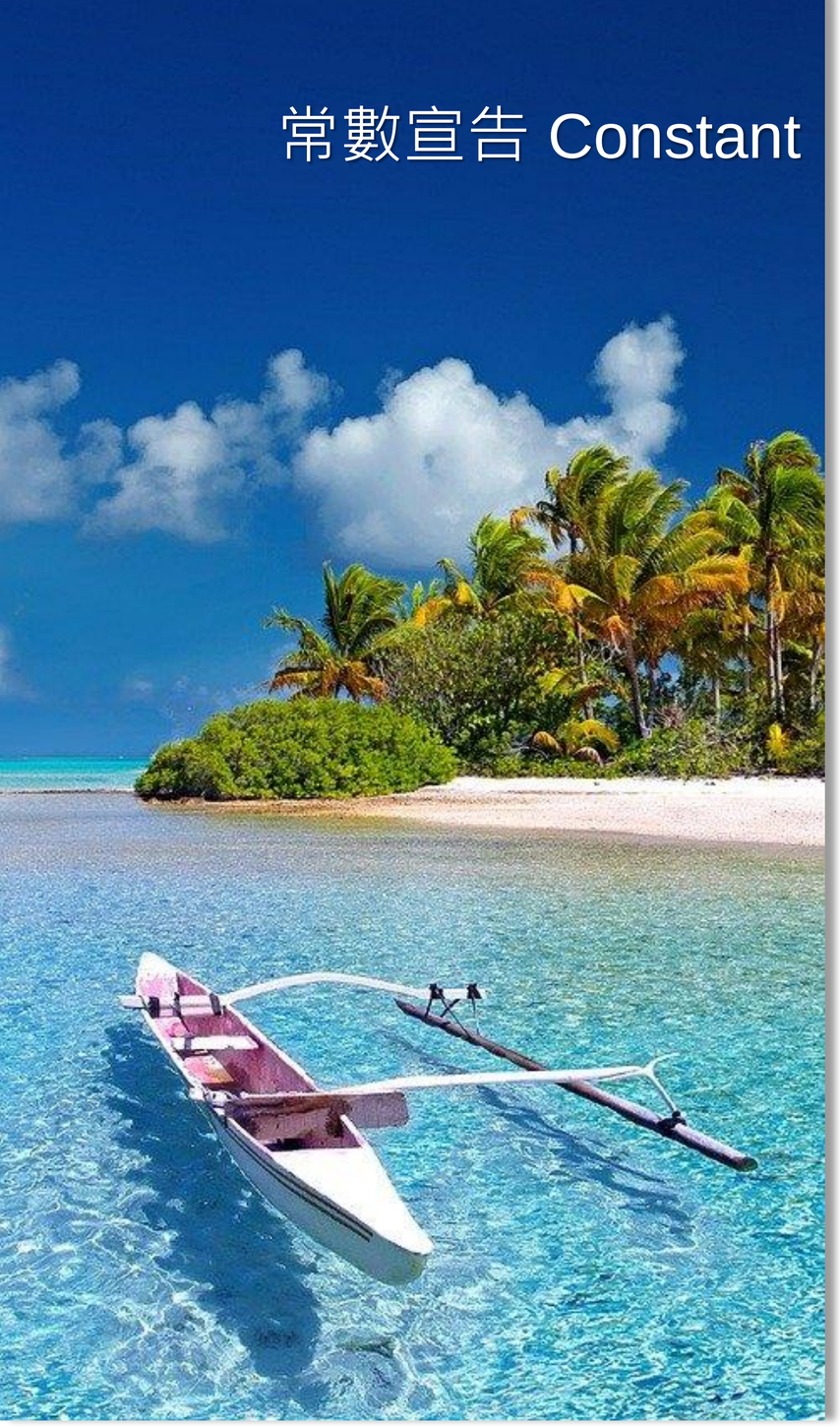
布林值 boolean



- 布林值 boolean 資料型別的值只能使用 true 與 false，不可以使用數值資料 0 或 1來表示
- 應用於條件運算時，運算式的結果可為 true 或 false

```
System.out.println(true);   //顯示true
System.out.println(!true); //顯示false，!-->Not(反)
```

常數宣告 Constant



- 常數：將數值指定給變數後，就不可以再更改，宣告時可以使用
- final關鍵字
- 語法
  - final 資料型別 常數名稱 = 值;

```
final double PI = 3.14;
```

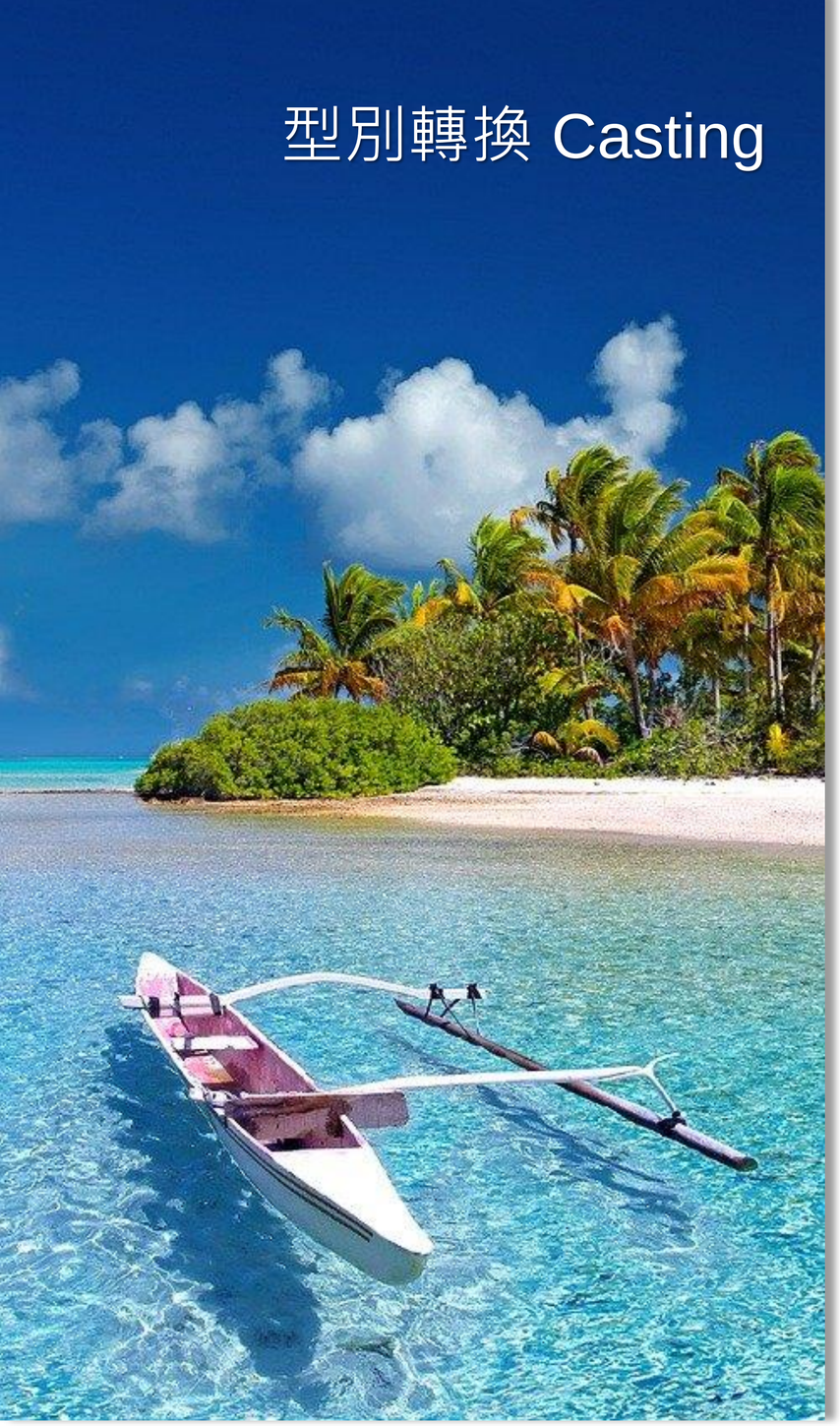
關鍵字　資料型別　常數名稱 = 值

```
PI = 3.14159;
```

cannot assign a value to final variable PI

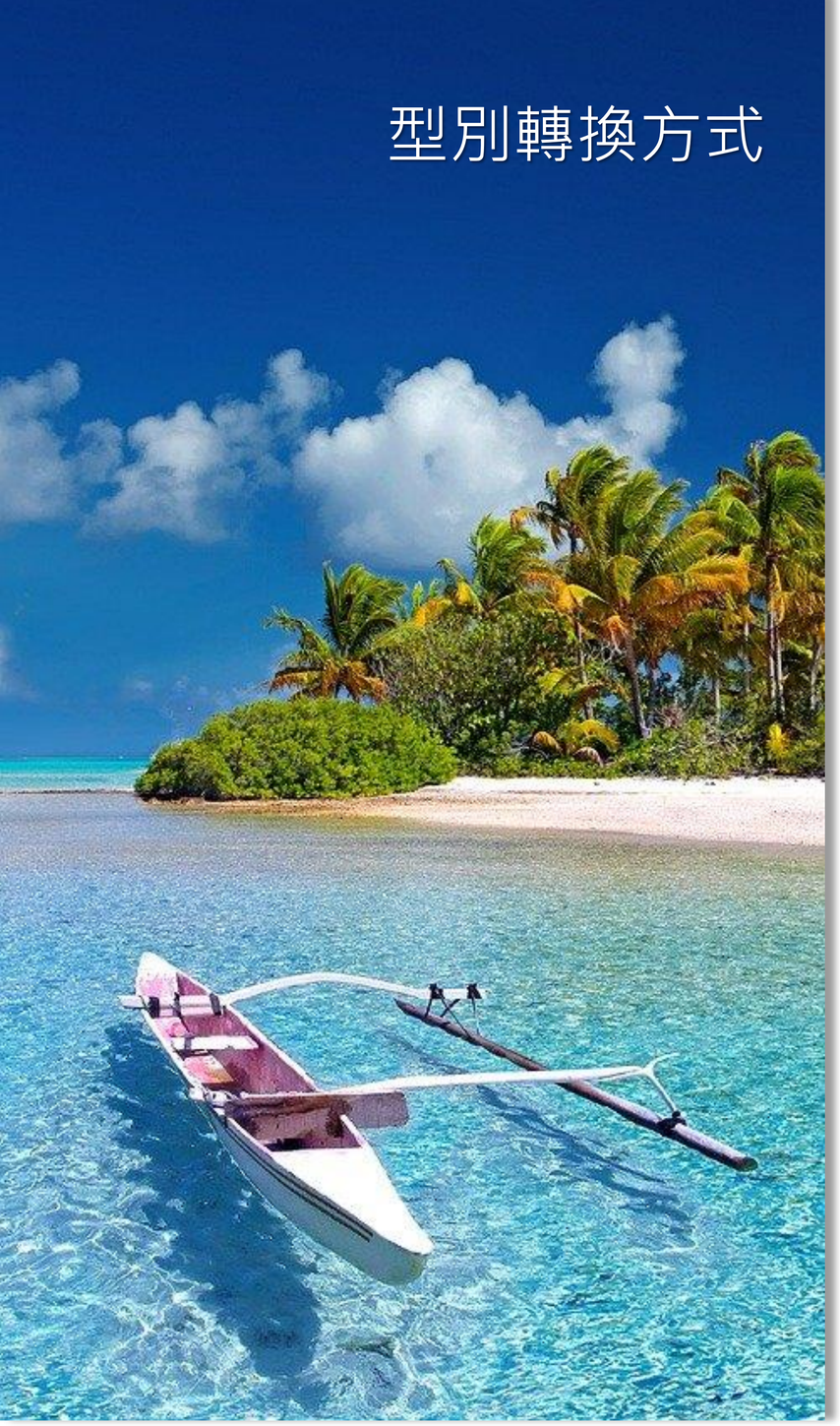
(Alt-Enter shows hints)

型別轉換 Casting



- Java 資料型別轉換有隱含轉換與強制轉換
- 隱含轉換(implicit casting)
  - 以較小的資料型別轉成較大的資料型別，無風險，系統會根據程式的需要自動且適時地做轉型
- 強制轉換(explicit casting)
  - 以較大的資料型別切割成較小的資料型別，有風險，必須在程式中給予明確的指令，系統並不自動轉型

型別轉換方式



- 轉換模式

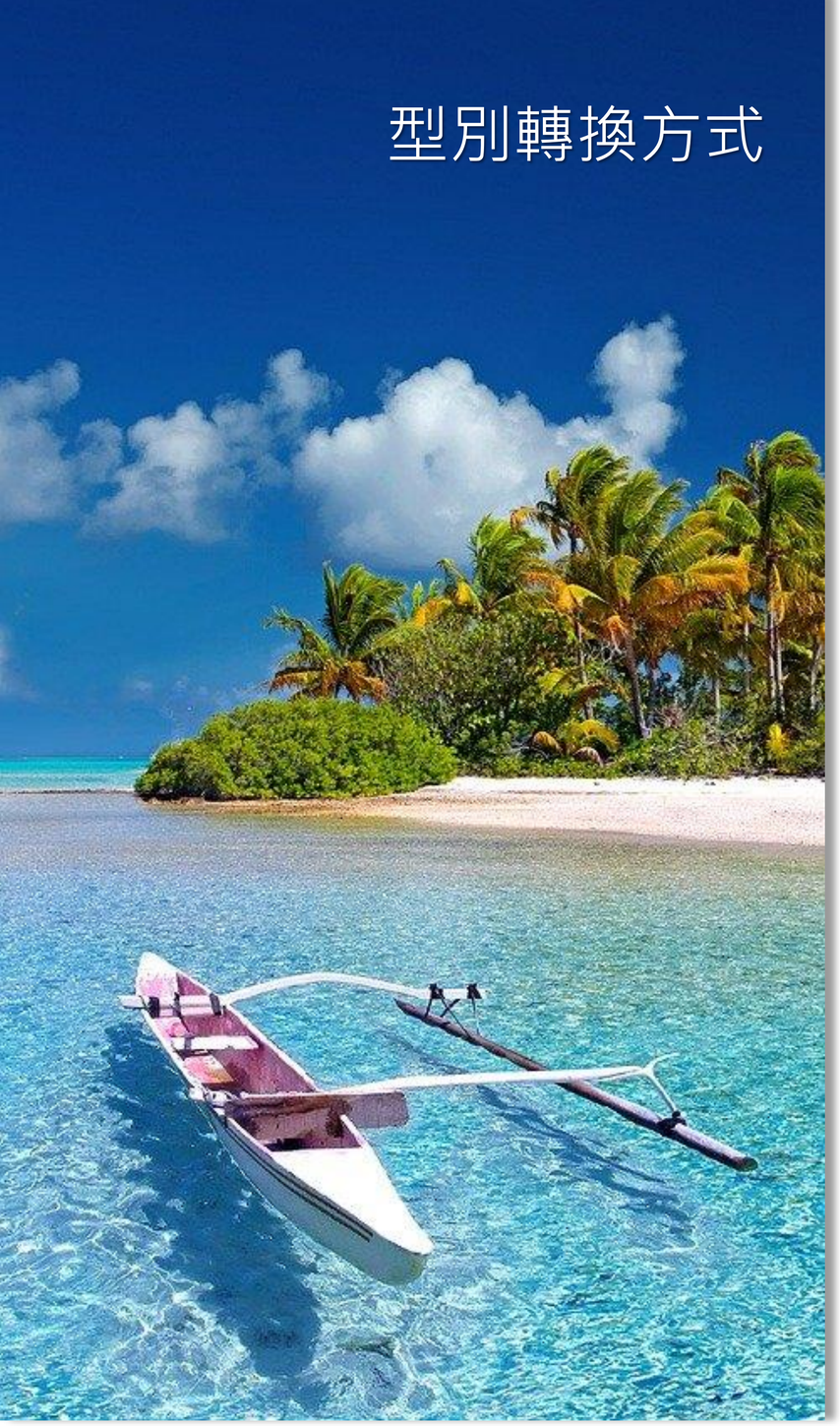
隱含轉換 ：小轉大 無風險　　long numA = 123;

byte - short - char 〉 int - long - float - double

強制轉換 ： 大轉小 有風險　　int numB = (int) 1.23;

- 語法
  - (目標型別) 值

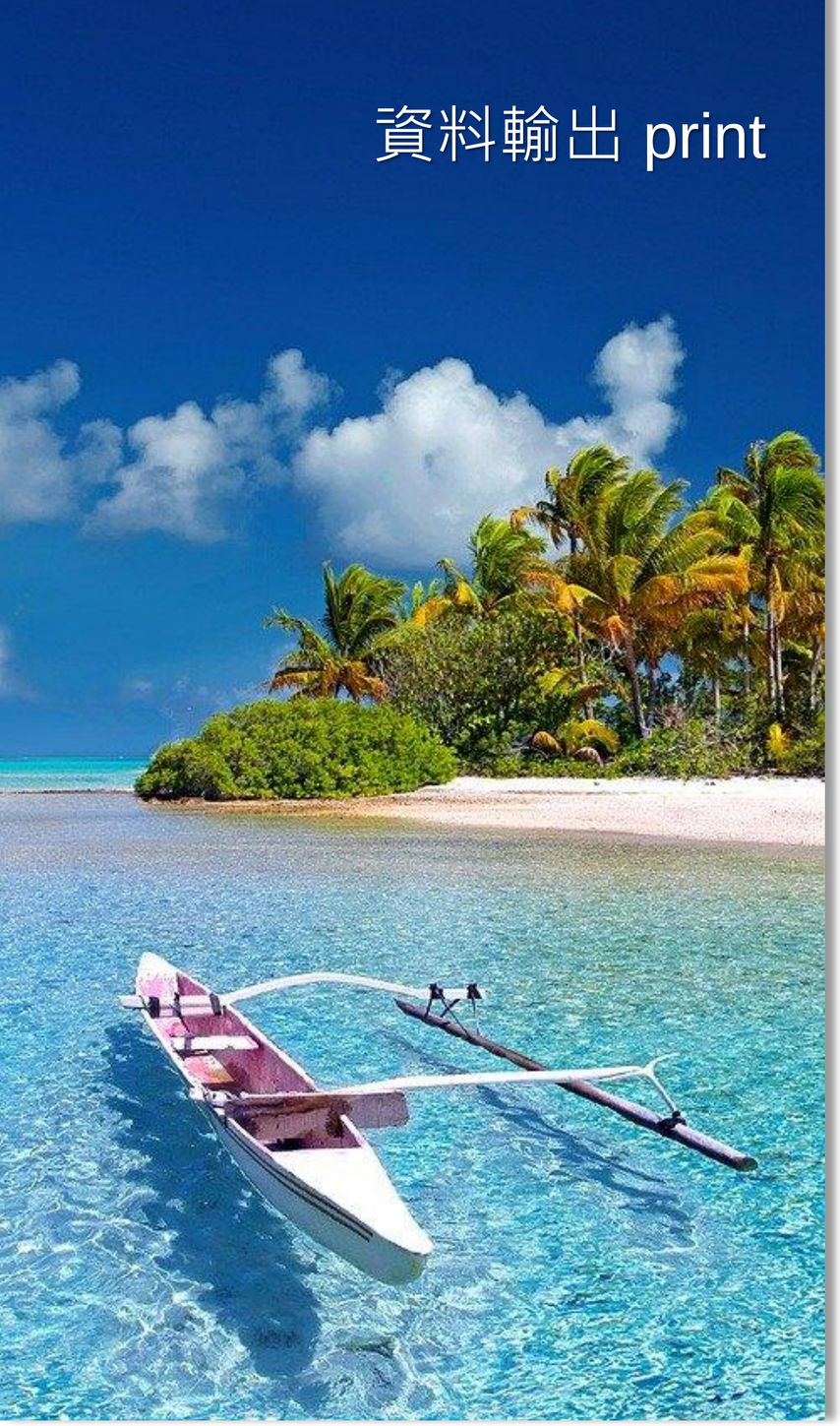
型別轉換方式



- 假設

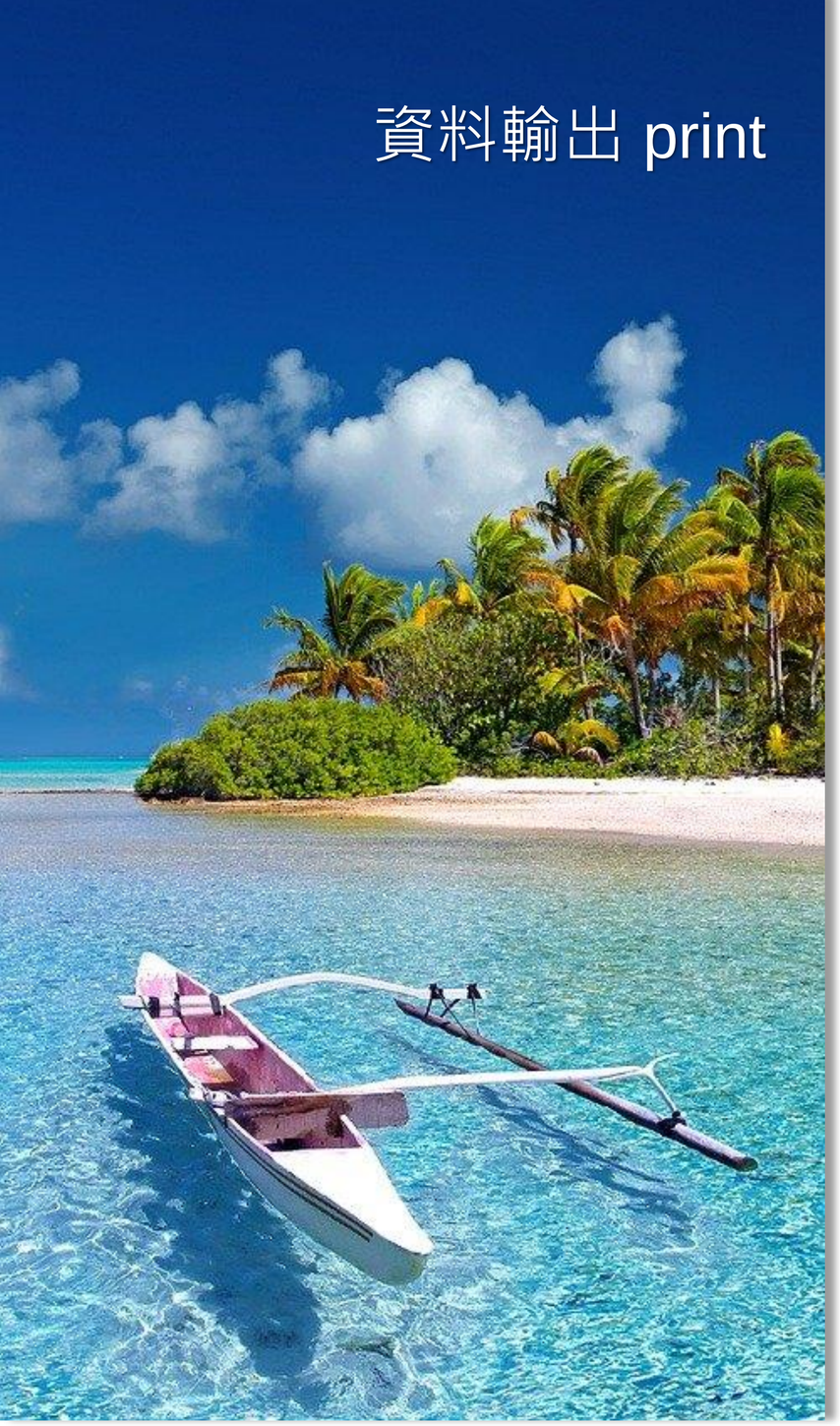
```
int  i = 1;
double d;
float f;
d = i; //將 i 變數的內容值指派給 d。合法
i = d; //將 d 變數的內容值指派給 i。不合法
       //程式需要作 強制型別轉換。如：i = (int)d;
f = 2.5;
    //將 2.5 指派給 f。不合法，在 Java 語言中
    //字面值浮點數預設的資料型別是 double，
       //所以必須改寫成 f = (float)2.5; 或 f = 2.5f;
```

資料輸出 print



- System.out.print( ) 與 System.out.println( )
- 將資料輸出至螢幕時，使用 System.out 物件中的 print( ) 與 println( ) 方法，兩者差異在於 println( ) 顯示資料後會換行，print( ) 不會換行
- 語法
  - System.out.print(要輸出的資料或變數);
  - System.out.println(要輸出的資料或變數);

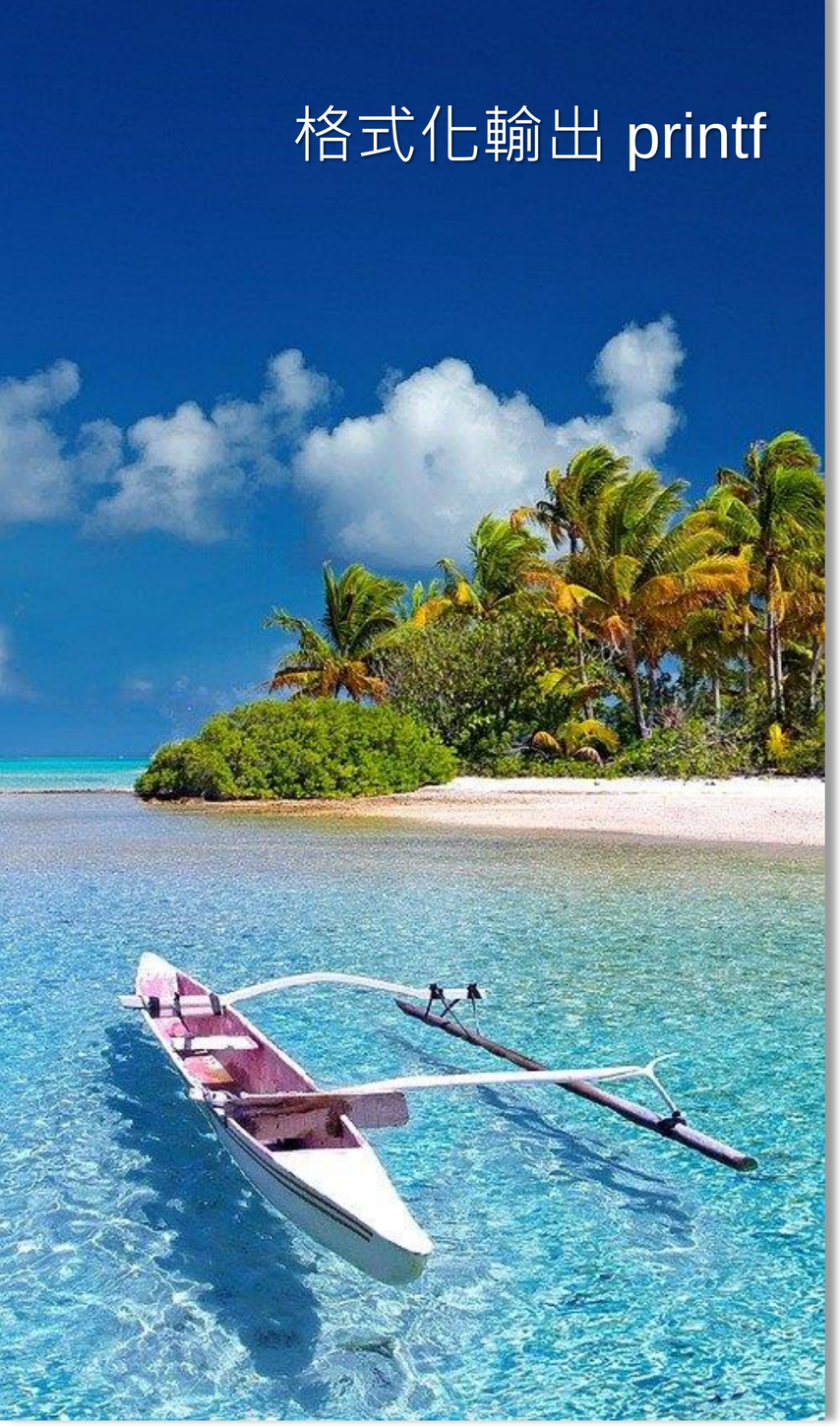
資料輸出 print



- 範例

```
System.out.println("第1行，資料顯示完成換行!");
System.out.print("第2行，資料顯示完成不換行!");
System.out.print("\n第3行，使用\"\\n\"換行\n");

System.out.println(1+3+5);     //加號 9
System.out.println("1"+3+5);   //字串串接 "135"
System.out.println("1"+(3+5));// "18"
```

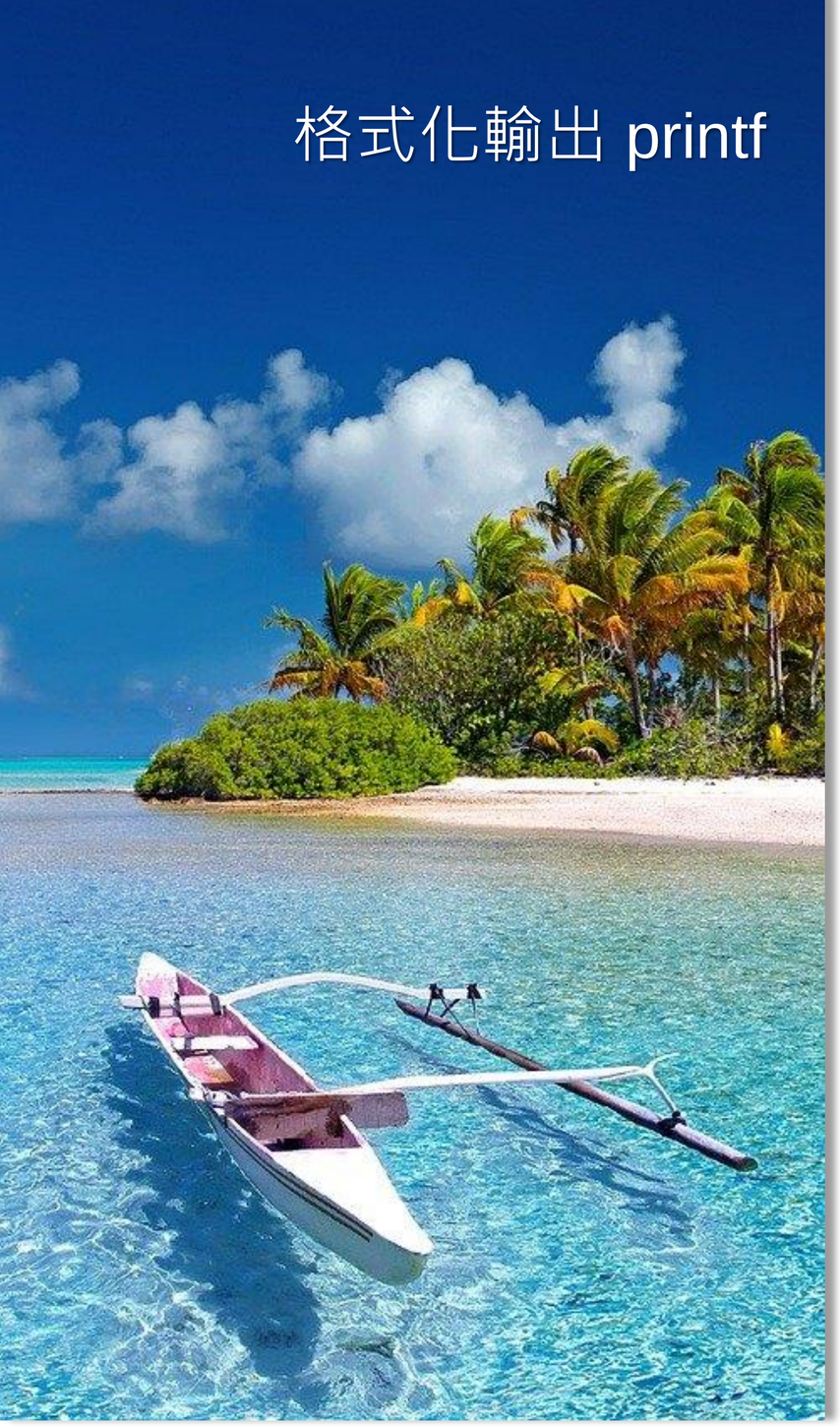
格式化輸出 printf



- 格式化輸出 printf( )
- 語法
  - System.out.printf(String format,Object… args)
  - String format 字串內容可含有特定的格式化字串
  - Object… args 是不定量參數，args 的每一個參數可被 format 字串中的特定的格式化字串對應到
- 範例

```
System.out.printf("我是%s，今年%d歲%n","Tom",10);
//顯示 我是Tom，今年10歲
```

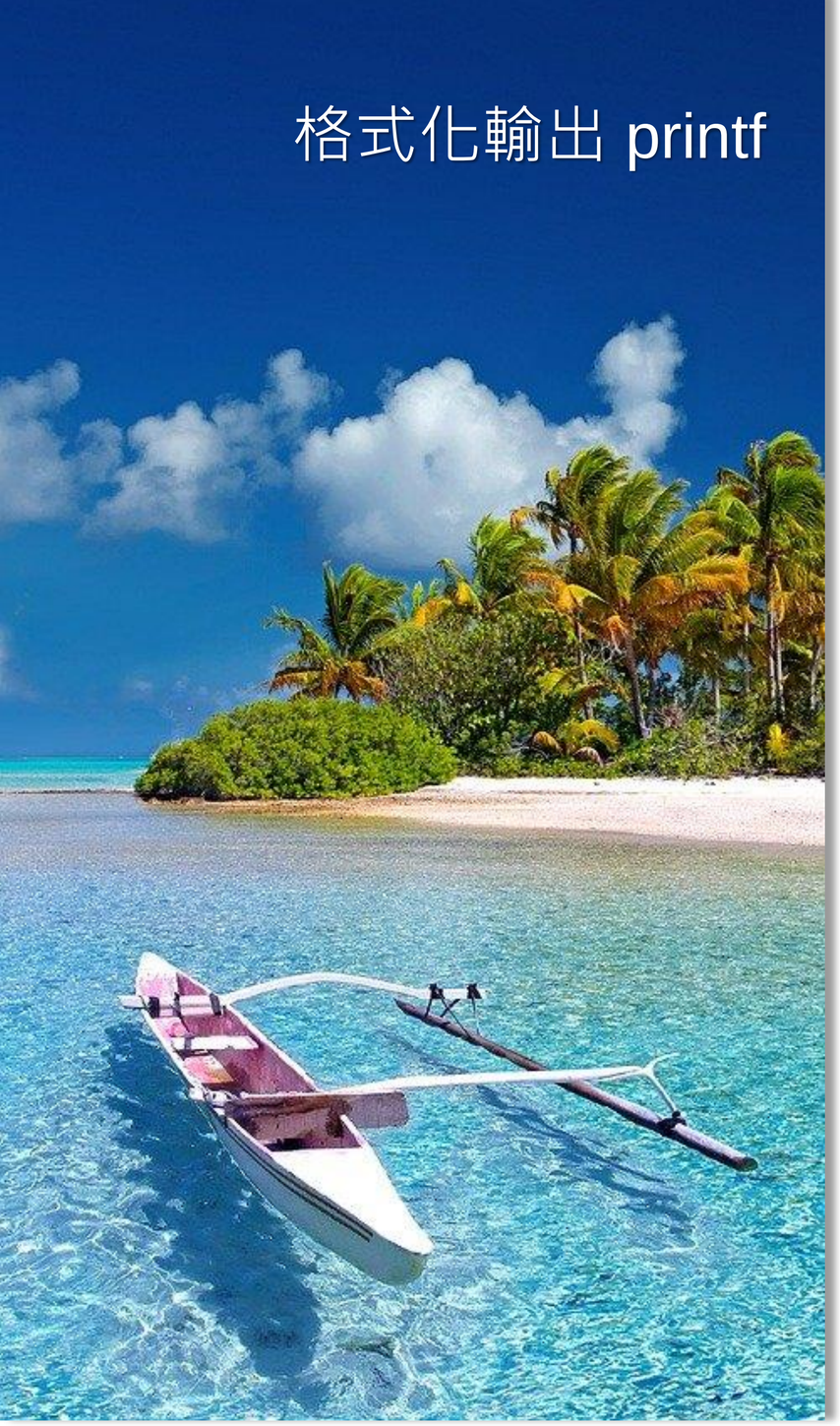
格式化輸出 printf



- 常用格式控制符號

符號	說明
%%	百分比符號
%d	以十進位整數格式輸出
%o	以八進位整數格式輸出
%x %X	以十六進位整數格式輸出
%f	以十進位浮點數格式輸出
%e %E	以科學記號浮點數格式輸出
%s %S	字串格式輸出
%c %C	以字元符號輸出
%b %B	輸出boolean值
%n	依平台而異的換行符號

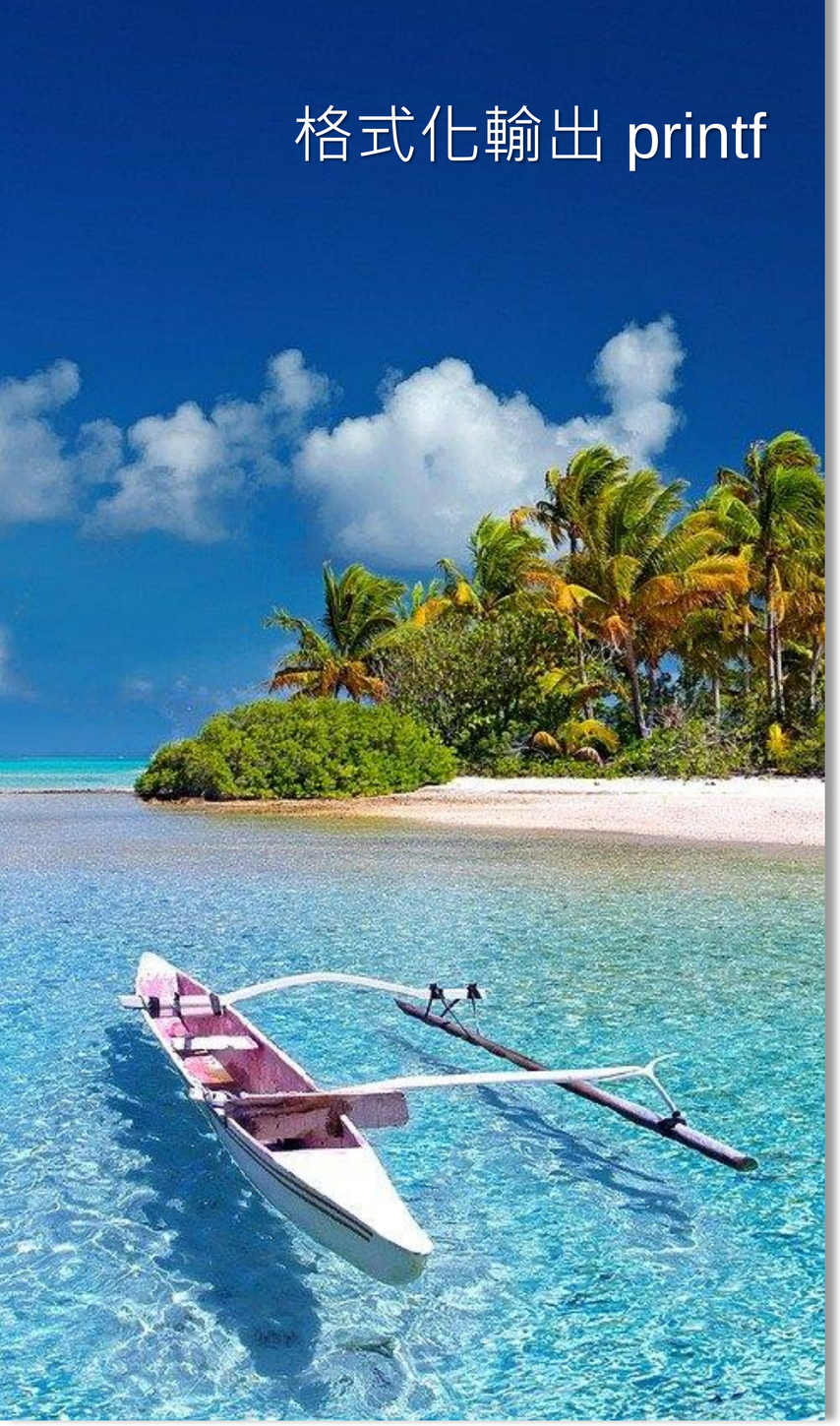
格式化輸出 printf



- 範例

```
String name = "Tom";
int age = 10;
double height = 123.45;

System.out.printf("我是%s，今年%d歲，身高%f公分%n",name,age,height);
```

我是Tom，今年10歲，身高123.450000公分



格式化輸出 printf

範例

```
String name = "Tom";
int age = 10;
double height = 123.45;

System.out.printf("我是%s，今年%d歲，身高%f公分%n",name,age,height);
```

我是Tom，今年10歲，身高123.450000公分

```
String type = "A";
System.out.printf("%s%04d%n", type,1);     //A0001
System.out.printf("%s%04d%n", type,24);    //A0024
System.out.printf("%s%04d%n", type,135);   //A0135
System.out.printf("%s%04d%n", type,2468);  //A2468
```

```
A0001
A0024
A0135
A2468
```

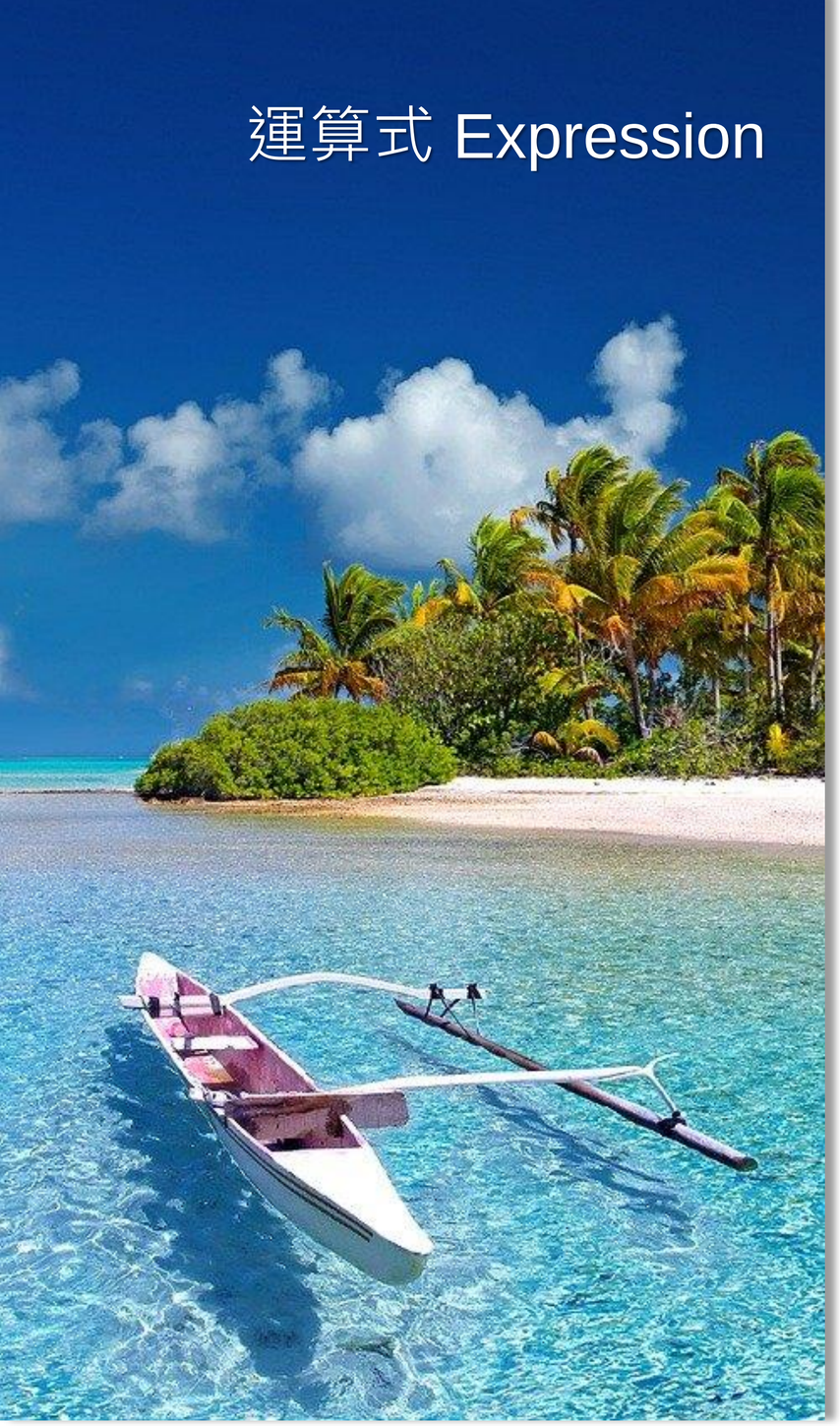
```
System.out.printf("%6.2f%n", 123.456789);
System.out.printf("%6.2f%n", 12.3456789);
System.out.printf("%6.2f%n", 1.23456789);
```

```
123.46
 12.35
  1.23
```

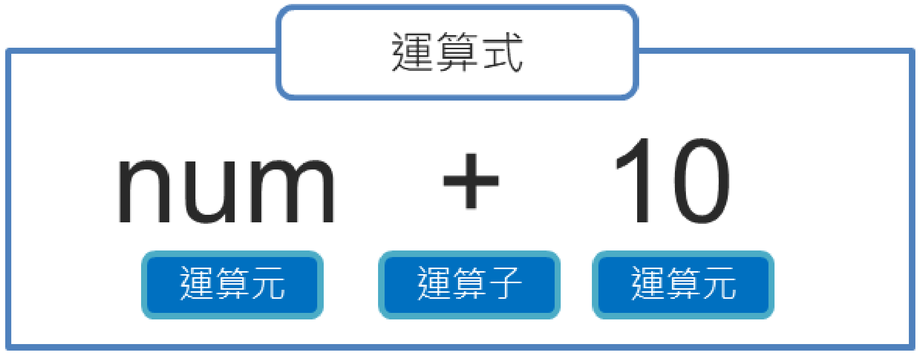
浮點數 %m.nf 其中m為總位數包含整數+小數點+小數 , n為小數位數

運算子 OPERATOR

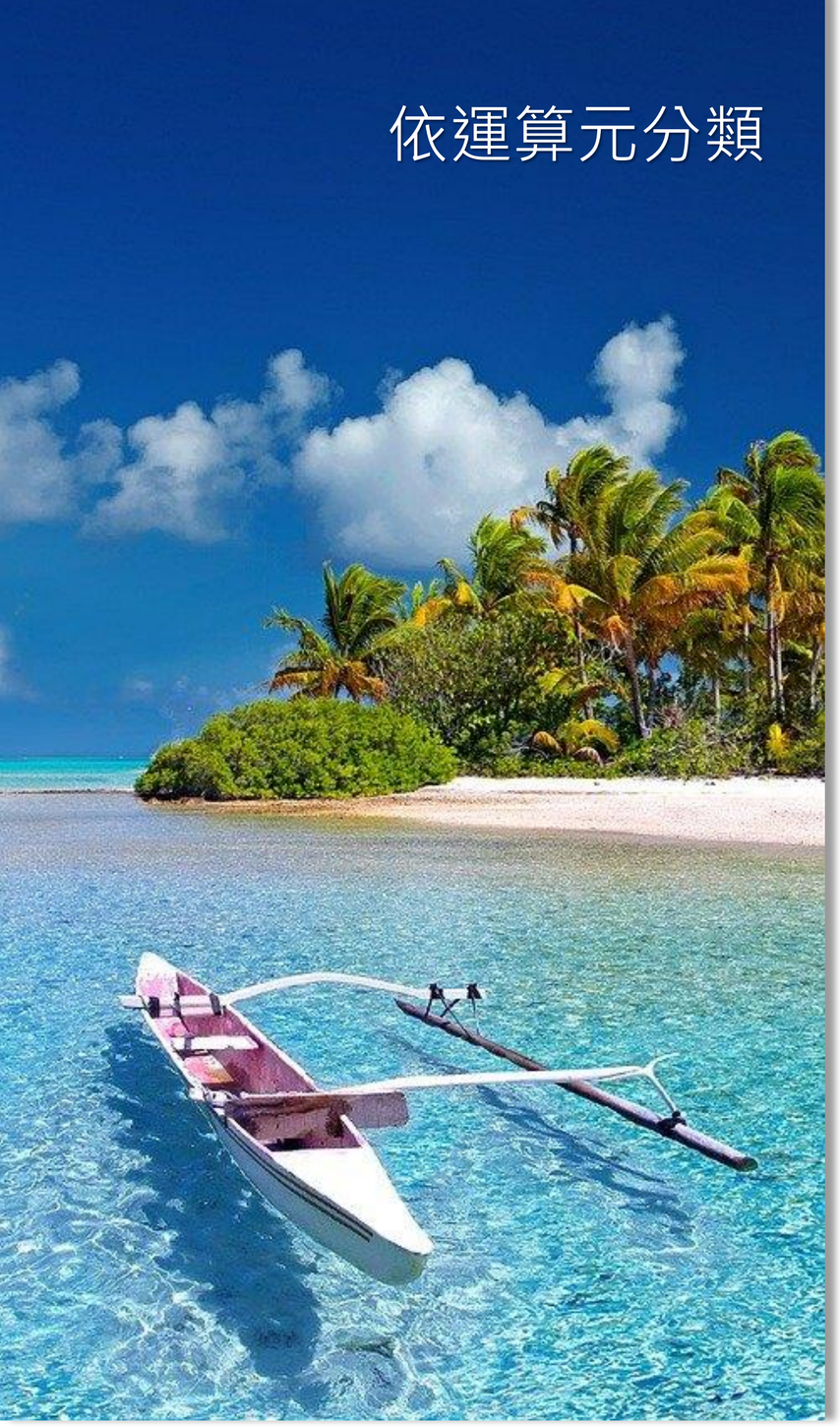
運算式 Expression



- 運算式是由運算子元與運算子所組成
- 運算元可以是變數或字面值
- 運算子是程式裡用於執行計算動作的符號
- 例如



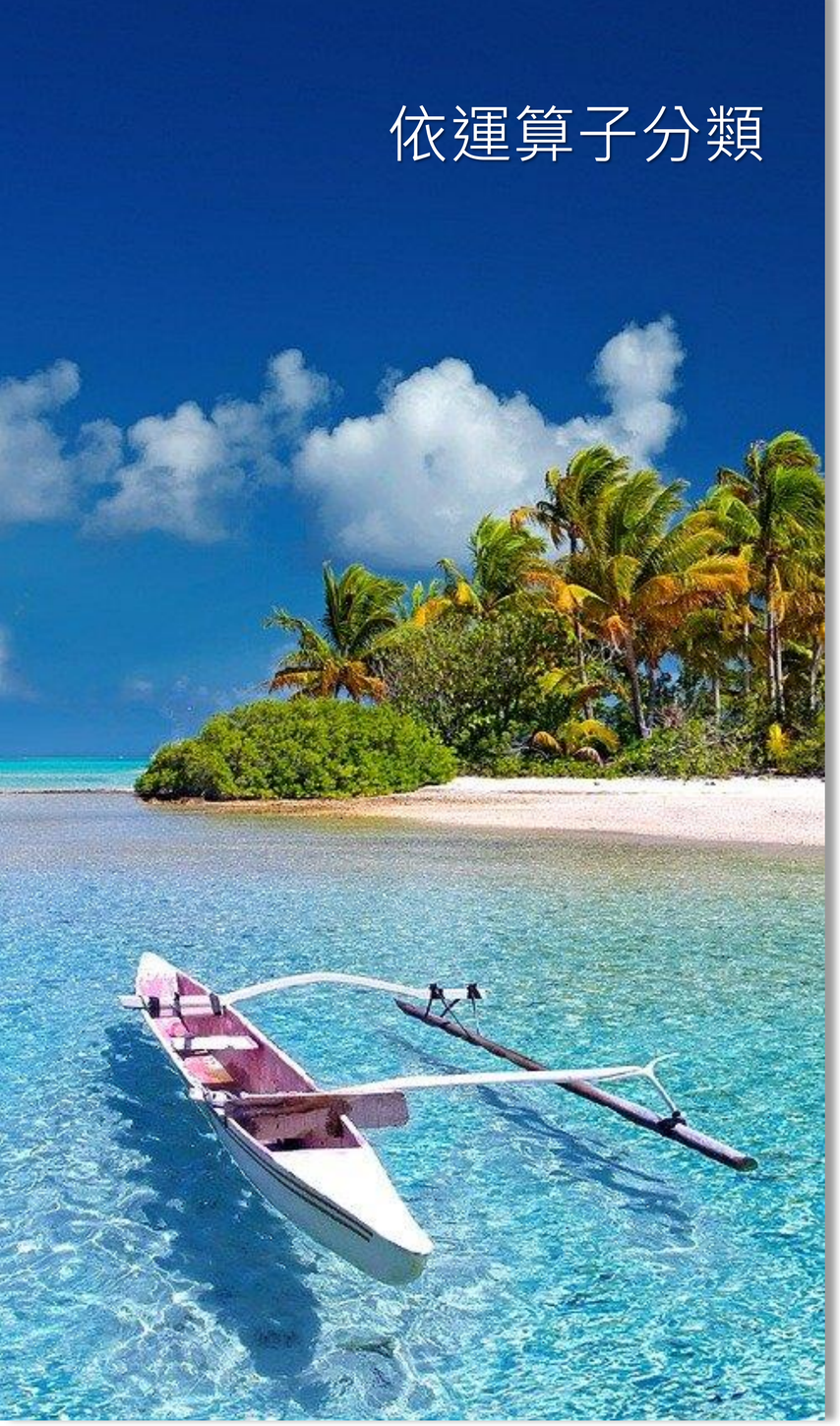
依運算元分類



- 運算子依運算元數目可分成
  - 一元運算子(uniry)(一般運算子的前後皆會有運算元，但一元運算子只有一個運算元)
    - -5 、 i++ 或 !booA
  - 二元運算子(binary)
    - x + y
  - 三元運算子(ternary)
    - max = a>b ? a : b

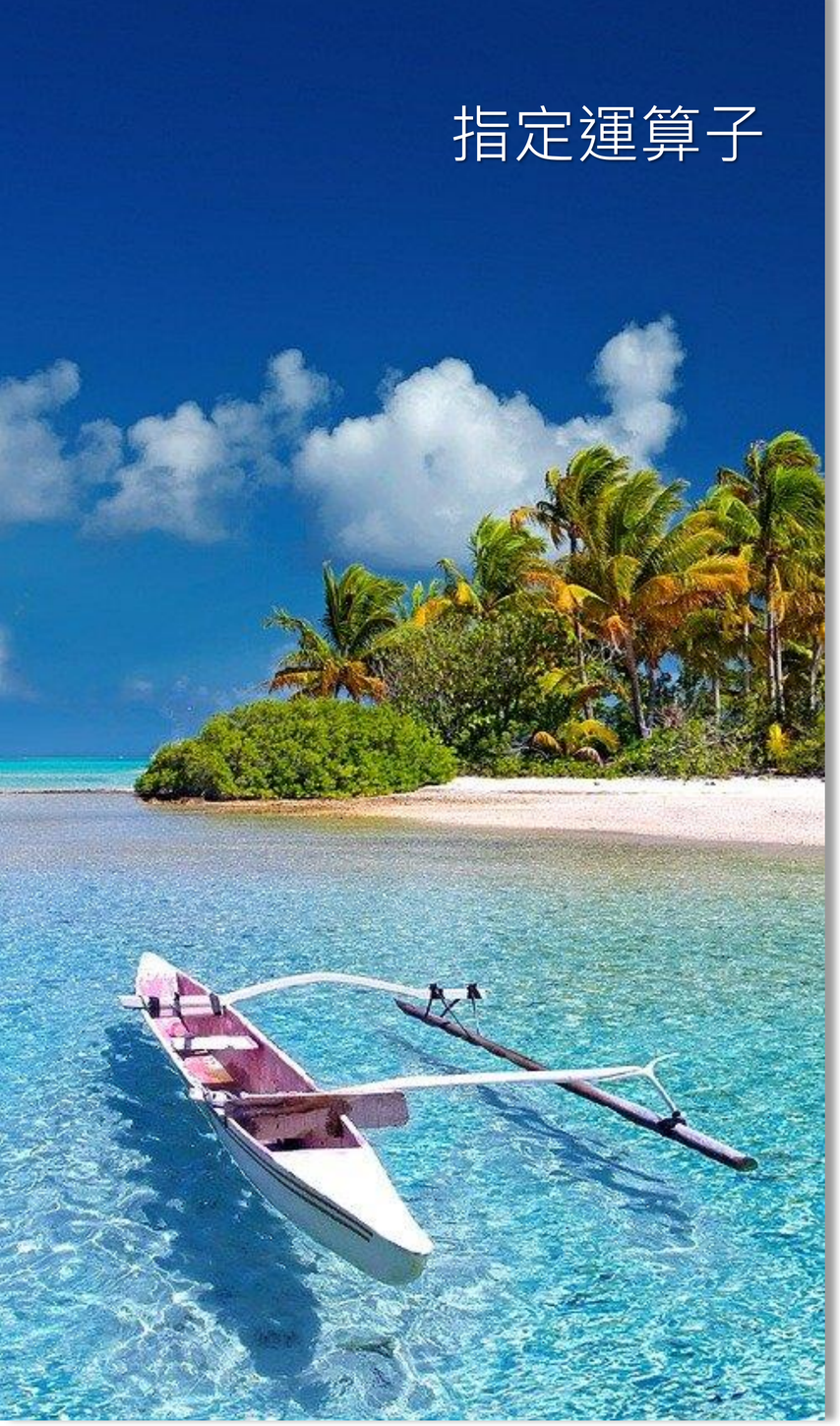
      //判斷 a 是否大於 b，若成立，將 a 指派給 max，若不成立，將 b 指派給 max

依運算子分類



- 運算子性質
  - 指定運算子 (=)
  - 算術運算子 (+ - * / %)
  - 遞增和遞減運算子 (++ --)
  - 關係運算子 (== != >= > <= <)
  - 邏輯運算子 (& | ! && ||)

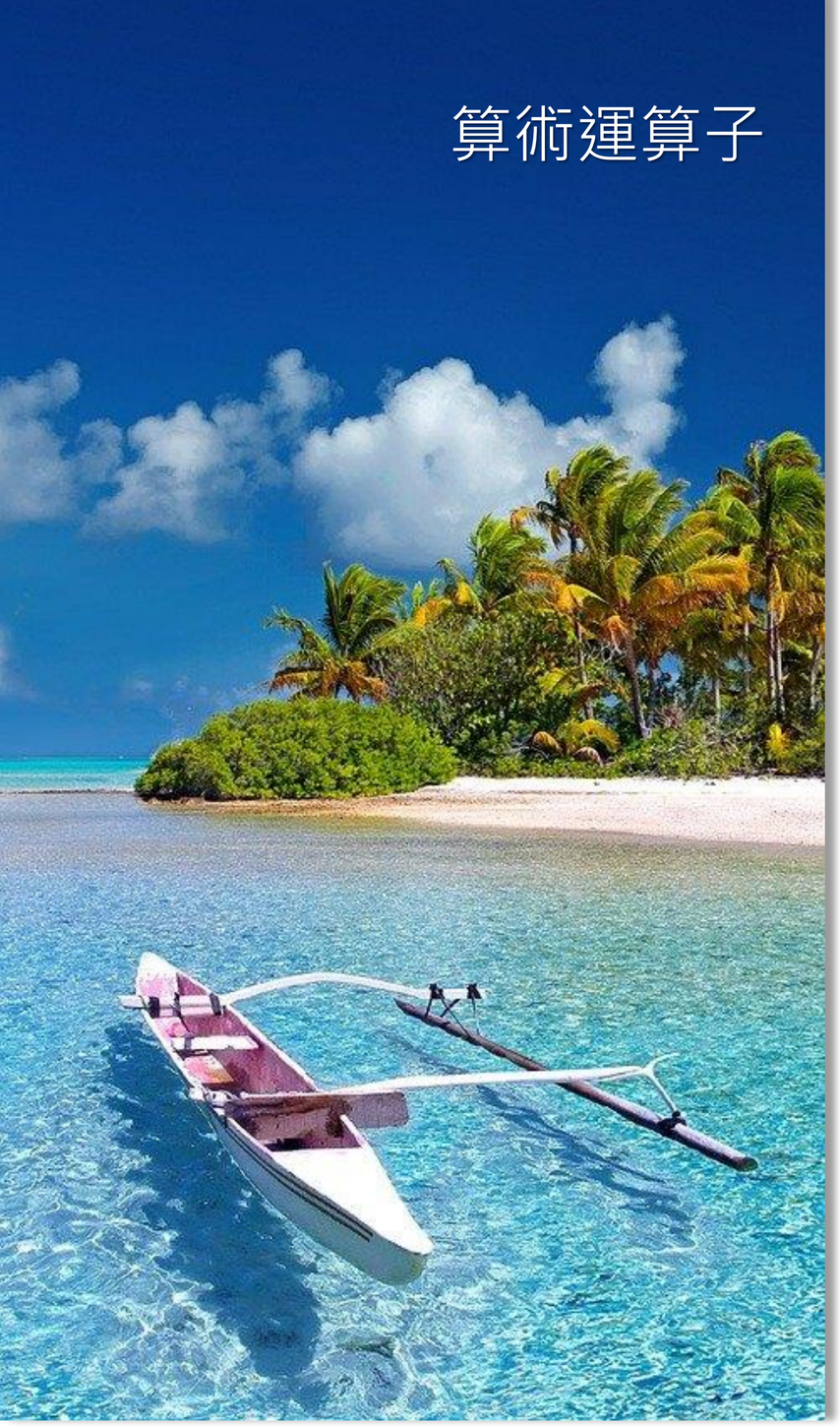
指定運算子



- 使用指定運算子『=』可將等號右邊的資料 (字面值、變數或運算式) 指定給等號左邊的變數
- 語法
  - 變數名稱 = 字面值 | 變數 | 運算式
- 例如

```
int a1;
a1 = 100;
a1 = a1 + 50;
System.out.println("a1="+a1); //顯示 150
```

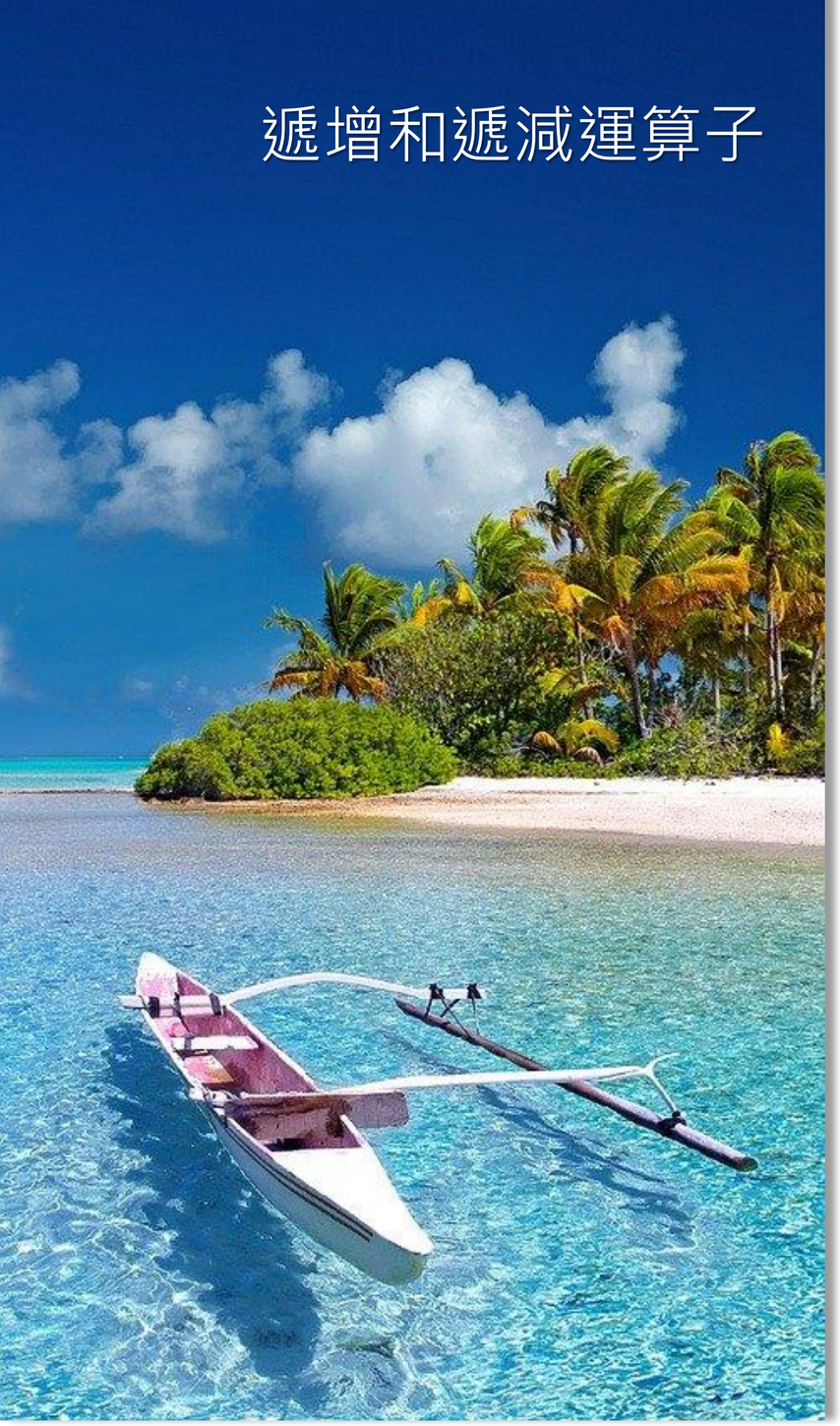
算術運算子



- 執行數學上常用到的加、減、乘、除與餘數
- 算數運算子有

運算子	名稱	語法範例
+	相加	a + b
-	相減	a - b
*	相乘	a * b
/	相除	a / b
%	取餘數	a % b

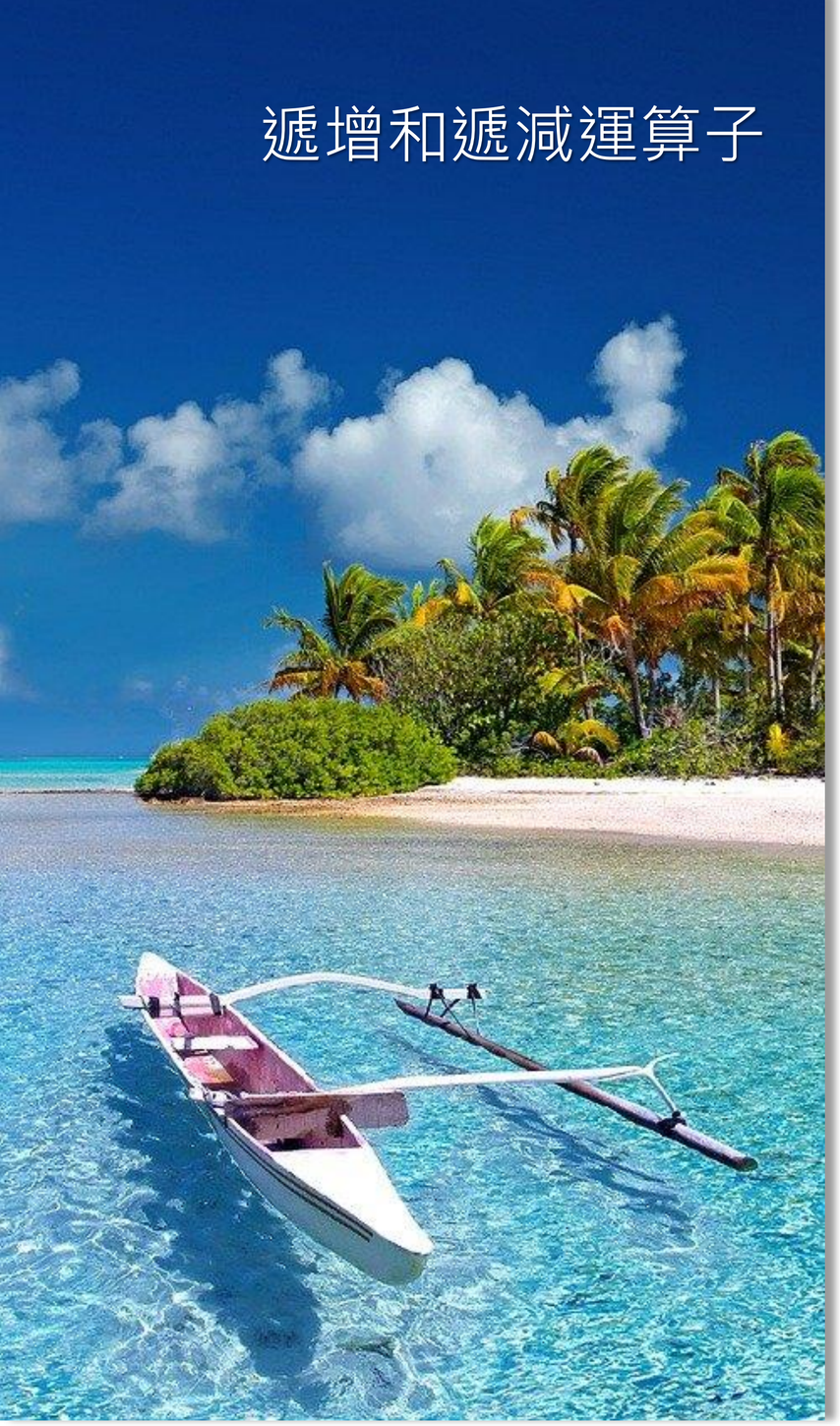
遞增和遞減運算子



- 將變數加 1 或減 1，如程式撰寫為 x=x+1 或 x=x-1 時，使用遞增或遞減運算子可以寫成 x++ 與 x--
- 遞增或遞減運算子

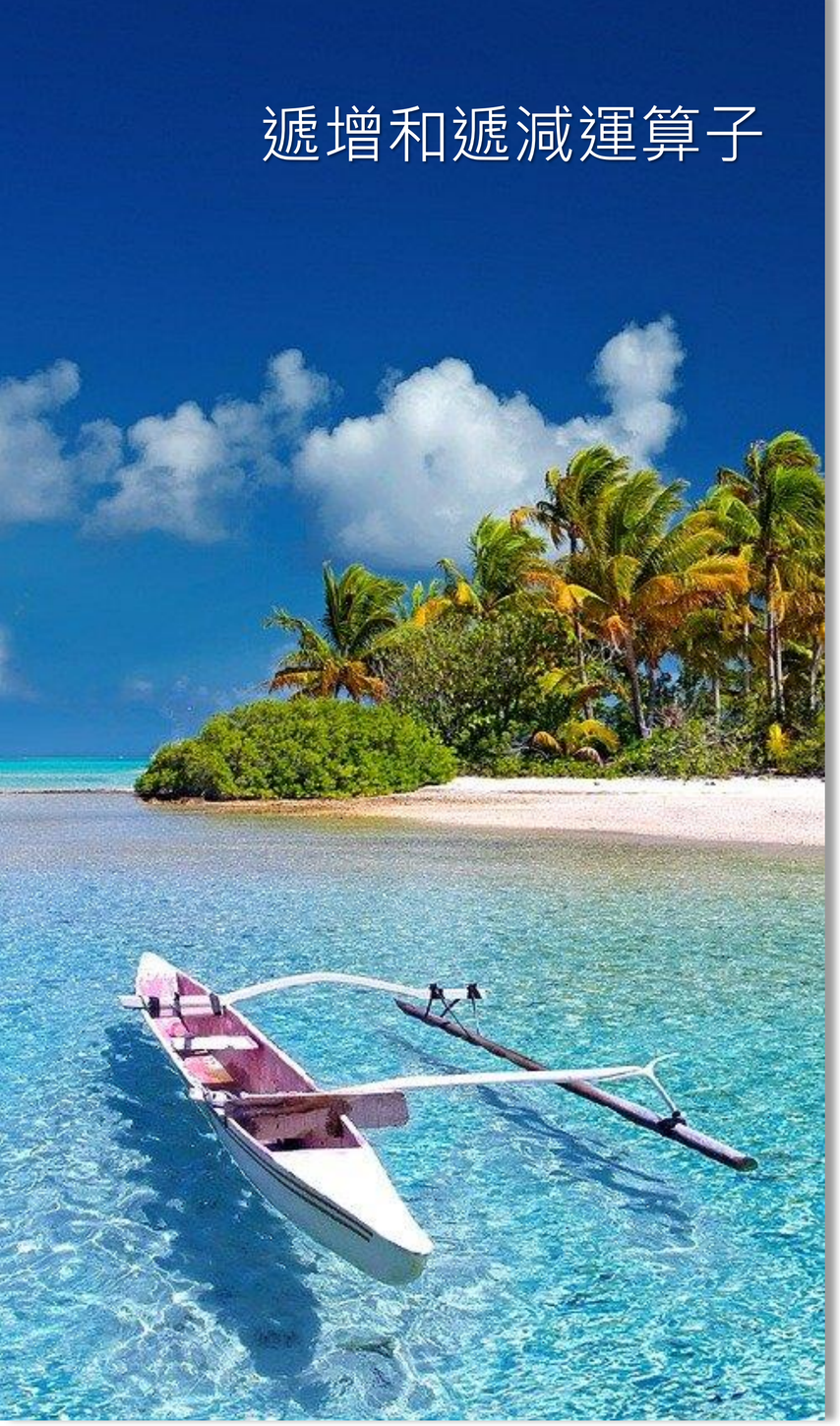
運算子	名稱	語法範例	功能等同
++	遞增	x++ 或 ++x	x = x + 1;
--	遞減	x-- 或 --x	x = x – 1;

遞增和遞減運算子



- 遞增與遞減運算子有分前置與後置，寫在變數前與寫在變數後是不一樣的
  - 前置 ++x 或 --x (x 先加 1 或 先減 1 再做運算)
  - 後置 x++ 或 x-- (x 先運算 再做加 1 或減 1)
- 例如

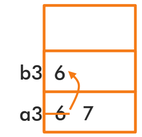
```
int a3 = 6;
int b3 = a3++; //後置遞增，a3先將資料指派給b3後，再將a3加1
System.out.printf("a3=%d ,b3=%d%n",a3,b3);

int c3 = 6;
int d3 = ++c3; //前置遞增，c3先加1後再將資料指派給d3
System.out.printf("c3=%d ,d3=%d%n", c3,d3);
```



遞增和遞減運算子

```
//後置遞增，a4先將資料指派給b4後，再將a4加1
int a3 = 6;
int b3 = a3++;
```

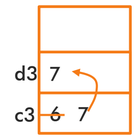
① 指派 ② 加1



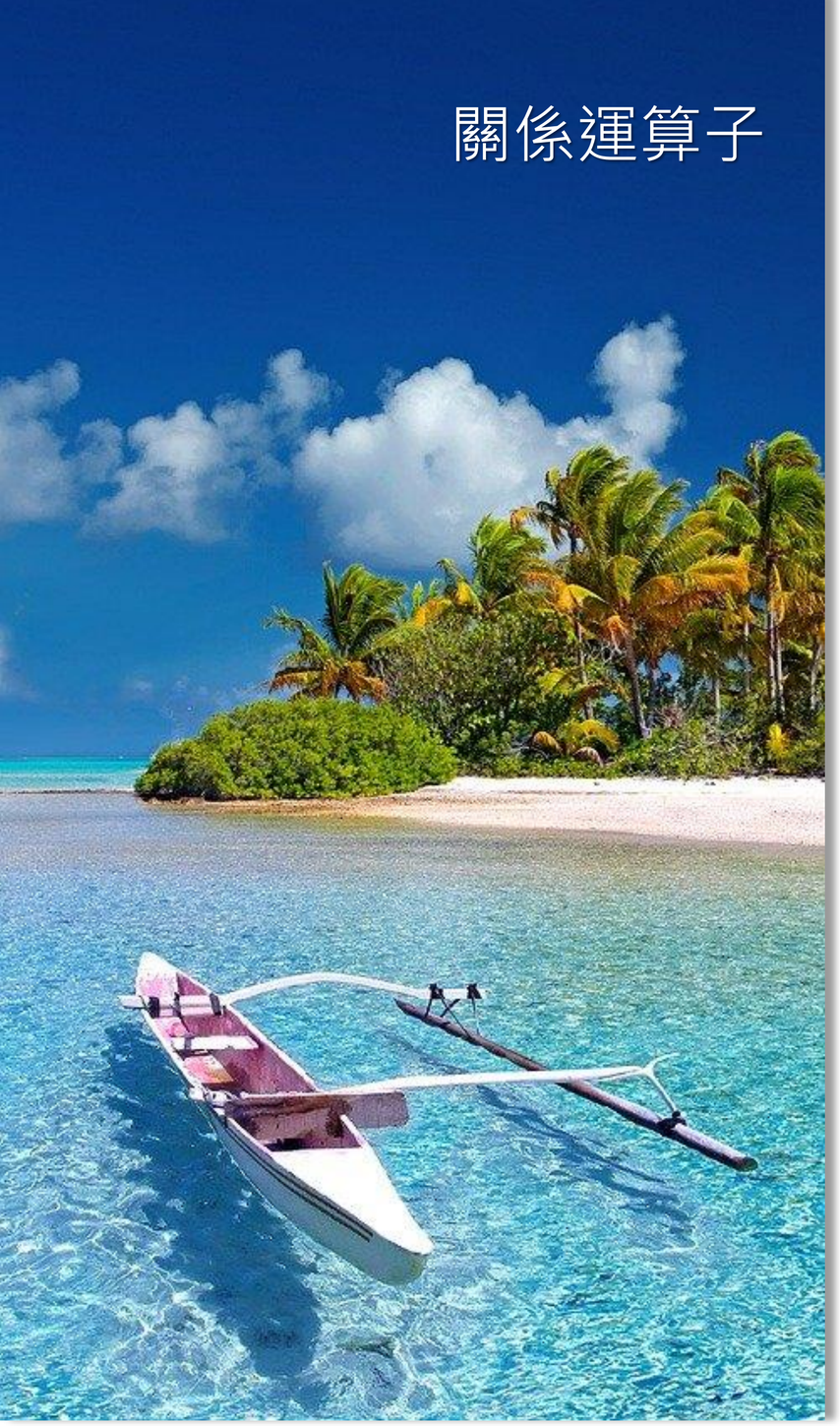

```
//前置遞增，c4先加1後再將資料指派給d4
int c3 = 6;
int d3 = ++c3;
```

① 加1

② 指派



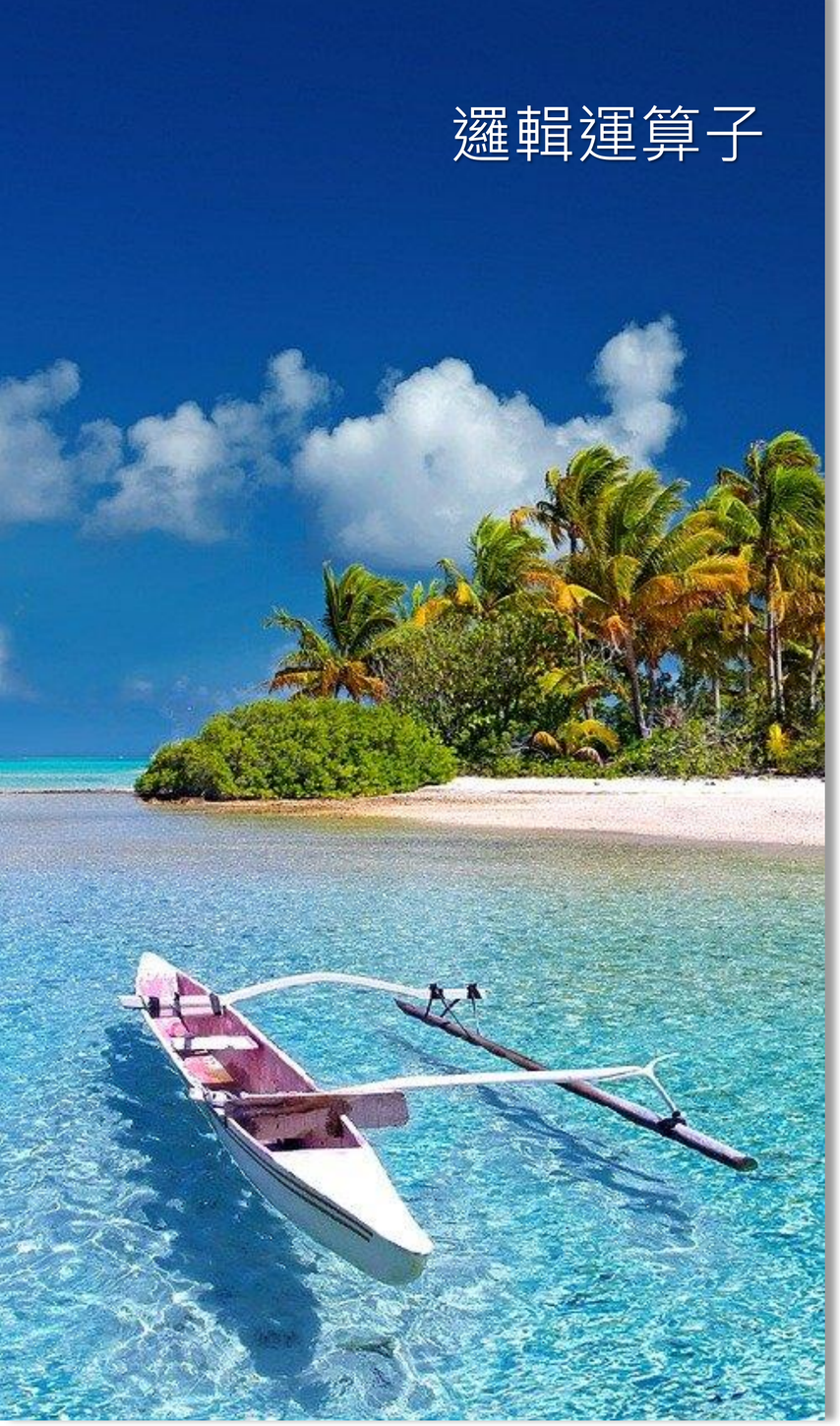
關係運算子



- 關係運算子(比較運算子) 用於比較兩數值間的大小，比較結果為 true 或 false
- 關係運算子有

運算子	名稱	語法範例
==	相等	a==b
>	大於	a>b
>=	大於等於	a>=b
<	小於	a<b
<=	小於等於	a<=b
!=	不等於	a!=b

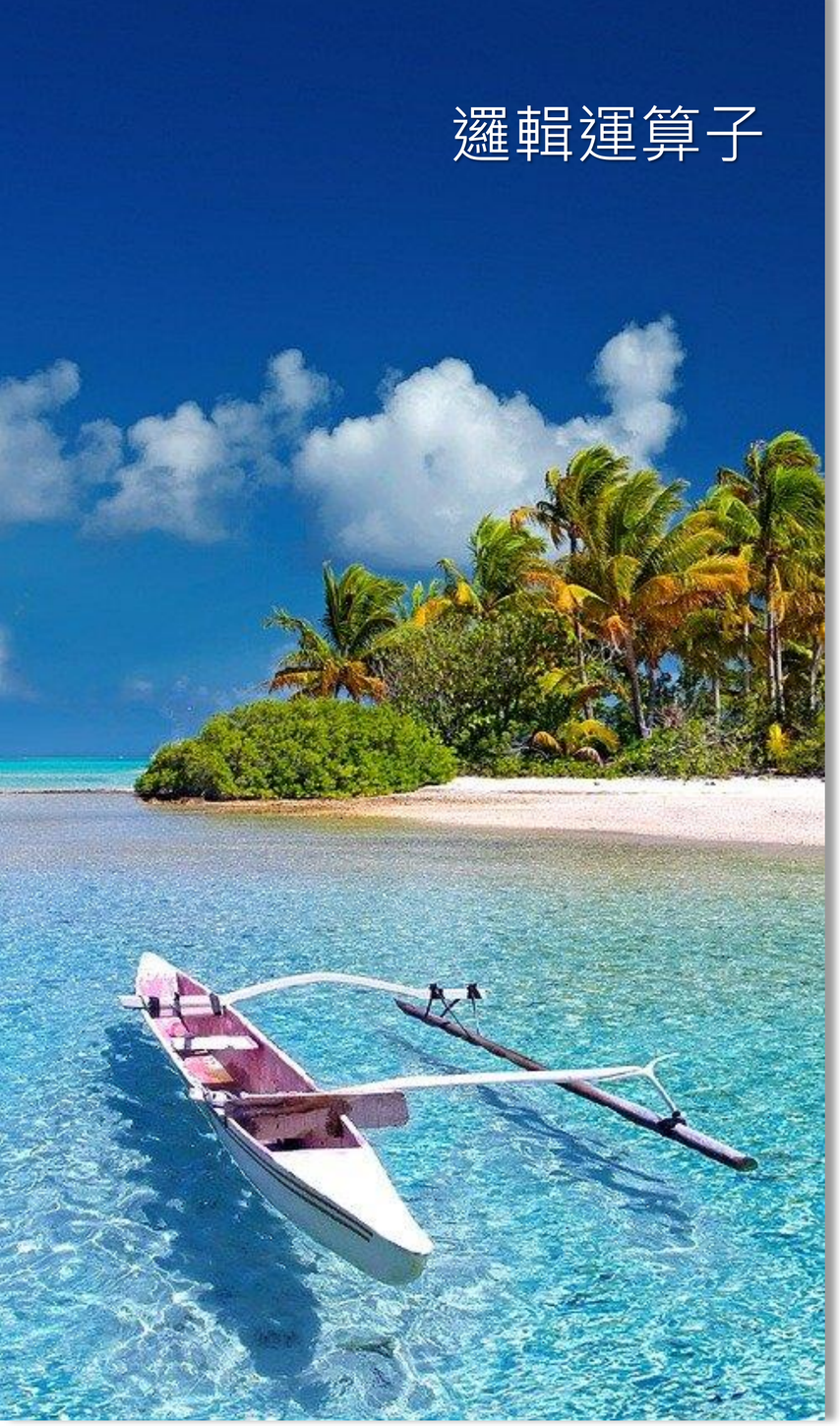
邏輯運算子



- 邏輯運算子可以用來連接兩個或多個關係運算式進行條件判斷，判斷結果為 true 或 false
- 邏輯運算子有

運算子	名稱	語法範例
& &&	AND(且)，連接的2個關係運算式皆為true，運算結果為 true	a & b a && b
| ||	OR(或)，連接的2個關係運算式任一個為true，運算結果為 true	a | b a || b
!	NOT(反)，運算結果為 true成false，false成true	! a

邏輯運算子



AND 與 OR 運算方式

運算式A	運算子	運算式B	結果
true	&	true	true
	&	false	false
false	&	true	false
	&	false	false

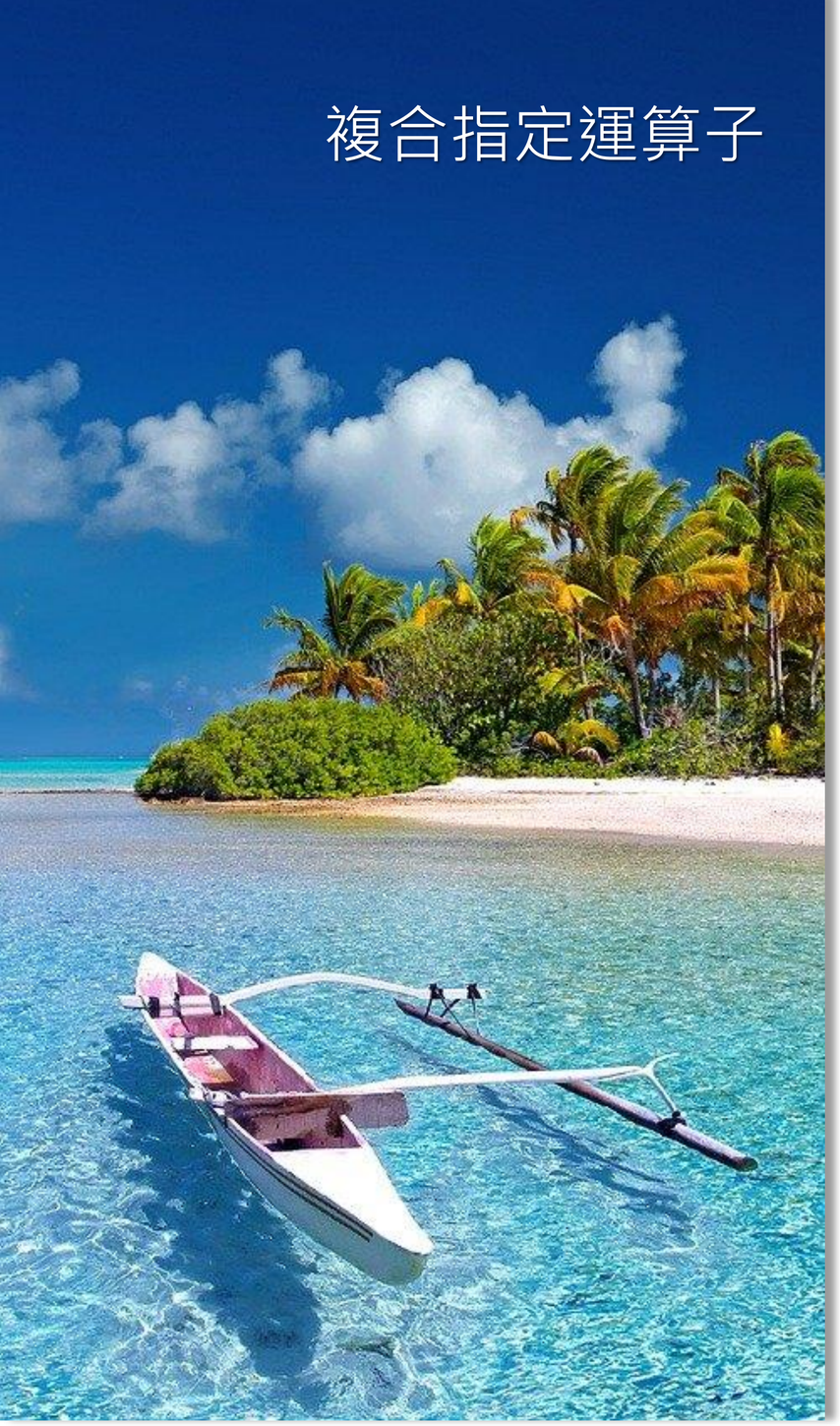
運算式A	運算子	運算式B	結果
true	|	true	true
	|	false	true
false	|	true	true
	|	false	false

捷徑運算

運算式A	運算子	運算式B	結果
true	&&	true	true
	&&	false	false
false	&&	X	false

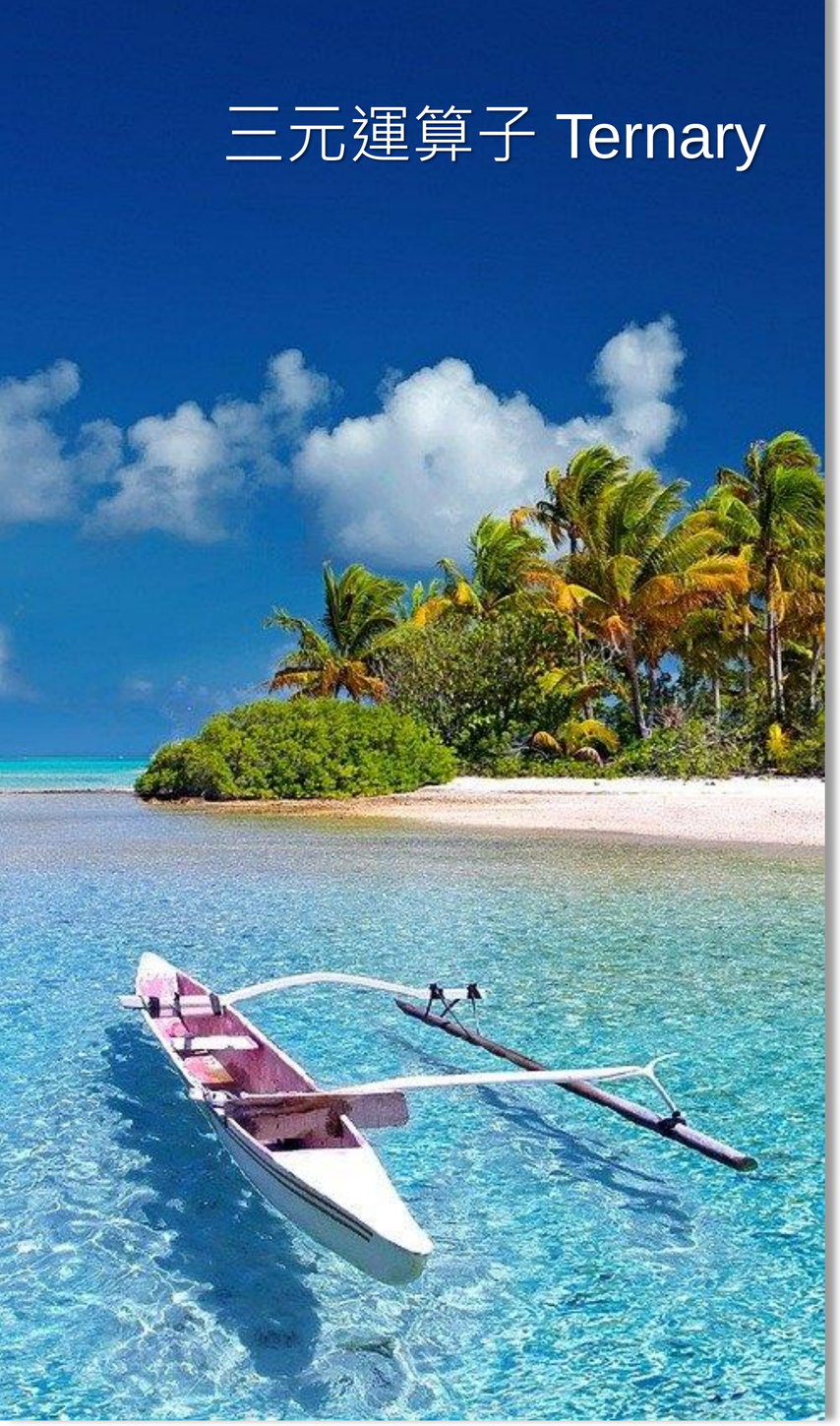
運算式A	運算子	運算式B	結果
true	||	X	true
false	||	true	true
	||	false	false

複合指定運算子



運算子	語法範例	功能等同
=	a =b	無
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a%= b	a = a % b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b

三元運算子 Ternary



- 三元運算子和一般 if-else 條件敘述差不多，其表示方法如下：
  - 判斷式 ? 判斷式成立執行運算式 A : 判斷式不成立執行運算式 B
  - 其運算式是表示當判斷式的回傳值為 true 的時候，會進行冒號 “：” 左邊的敘述，反之則進行冒號右邊的敘述

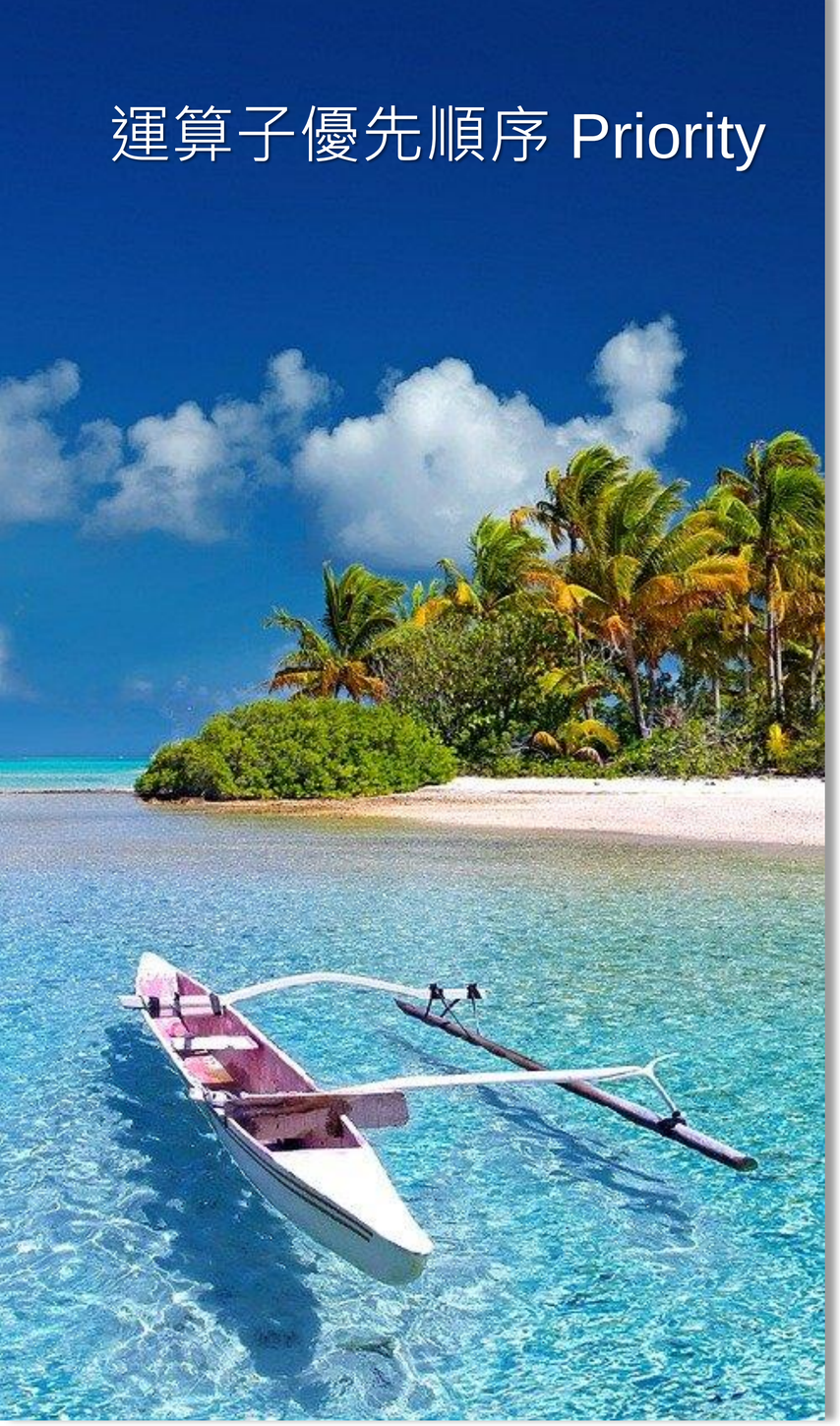
```
int a7 = 6,b7 = 9;

int max = a7 > b7 ? a7 : b7;

System.out.printf("max = %d%n",max);
```

false

true

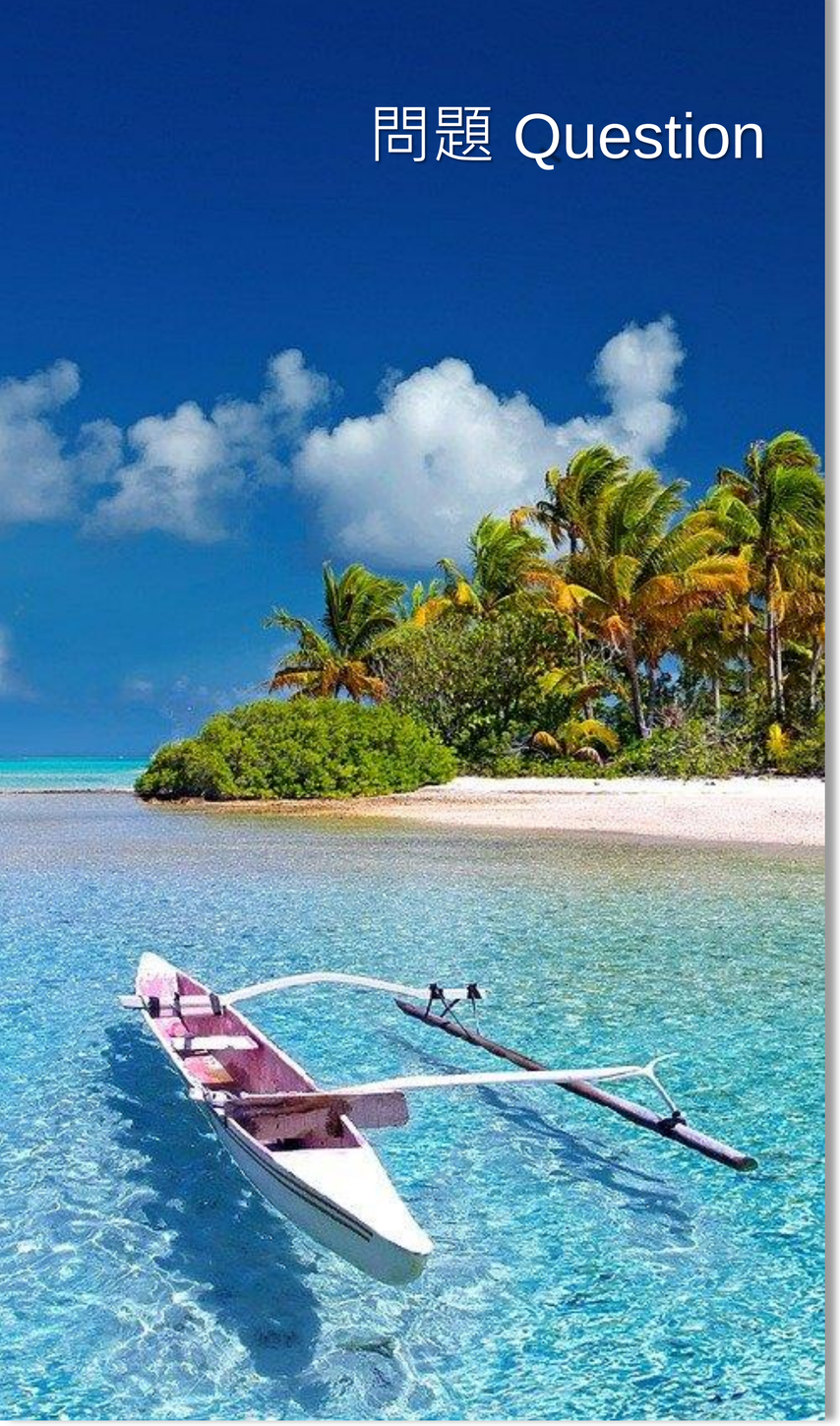
max = 9

運算子優先順序 Priority



優先權	運算子
1	( ) 括號
2	+ 正號 - 負號 ++ --
3	* / %
4	+ 加號 - 減號
5	>> << >>>
6	< > <= >= != ==
7	&
8	^
9	|
10	&&
11	||
12	?:
13	= += -= *= /= <<= >>= >>>=

問題 Question



```
boolean booA;
booA = 5 * 3 < 20 & 3 + 7 > 9 - 1 || 20 >= 20 - 30 || false;
System.out.println("booA = "+booA);
```

解析：

booA = 5 * 3 < 20 & 3 + 7 > 9 – 1 || 20 >= 20 – 30 || false

先算 5*3、3+7、9-1、20-30 的結果

15 < 20 & 10 > 8 || 20 >= -10 || false

再算15 < 20、10 > 8、20 >= -10 的結果

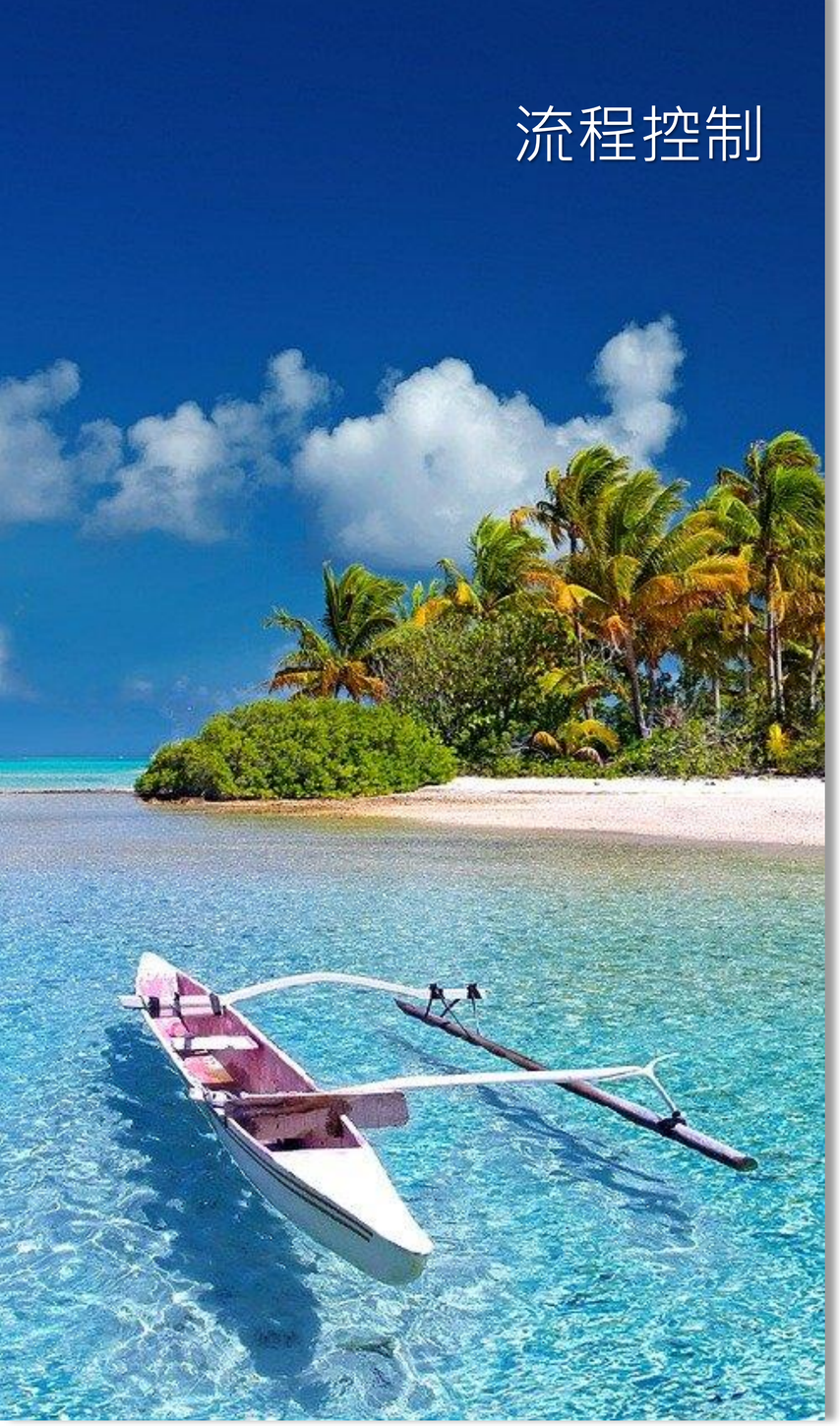
true & true || true || false

再算 true & true

true || true || false

booA = true

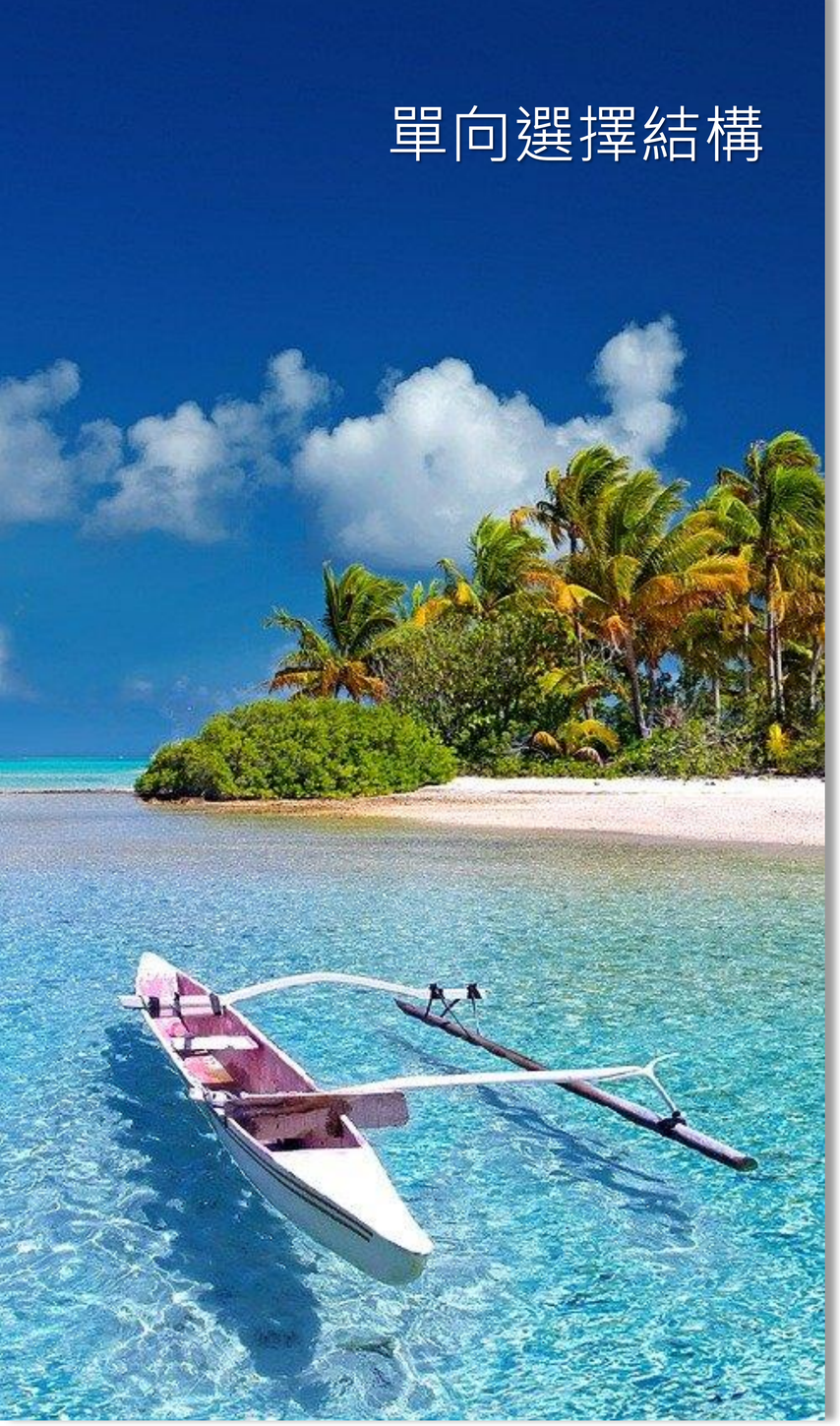
流程控制 CONTROL FLOW



流程控制

- 判斷式：根據關係運算或邏輯運算的條件來判斷程式執行的流程
  - if-else
  - switch-case
- 迴圈：根據關係運算或邏輯運算條件式的結果來判斷，以決定是否重複執行指定的程式
  - for
  - while
  - do-while

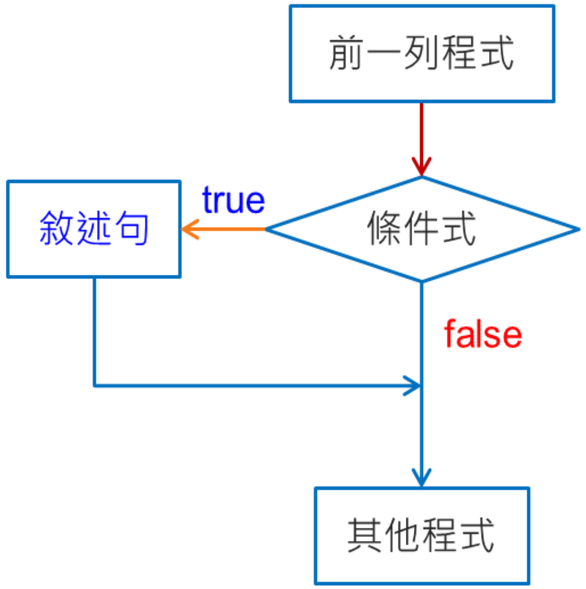
單向選擇結構



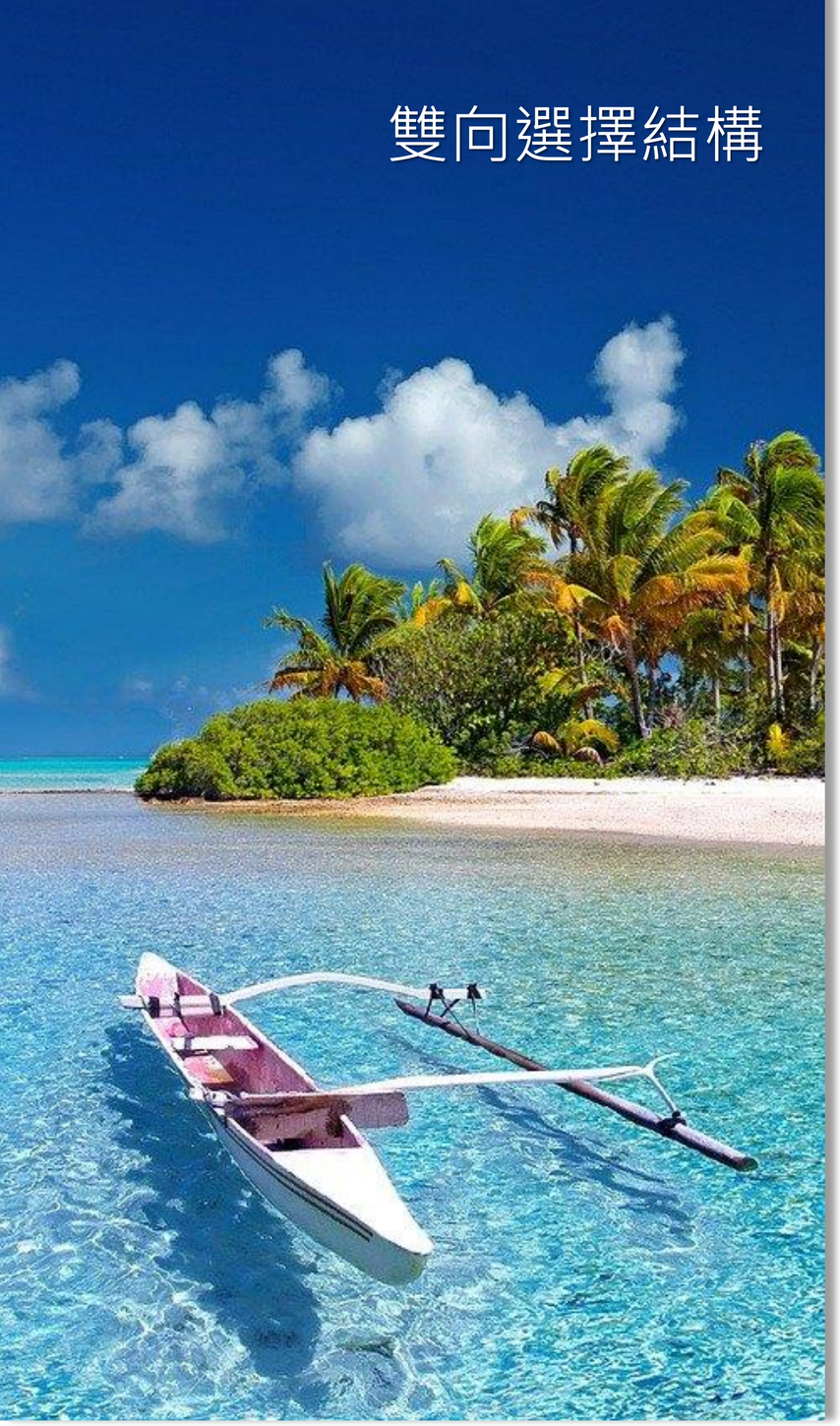
- 語法

```
if(條件式){
    敘述句;
}
```

```
if(條件式)
    敘述句;
```



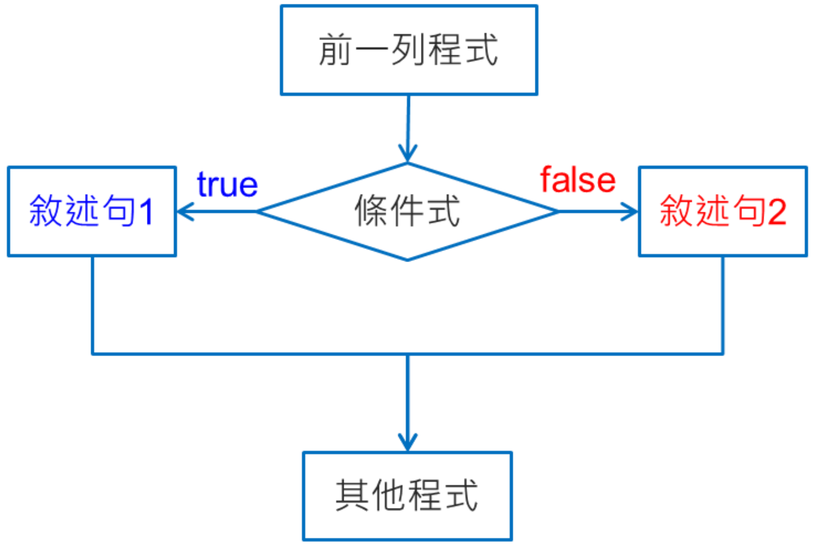
雙向選擇結構

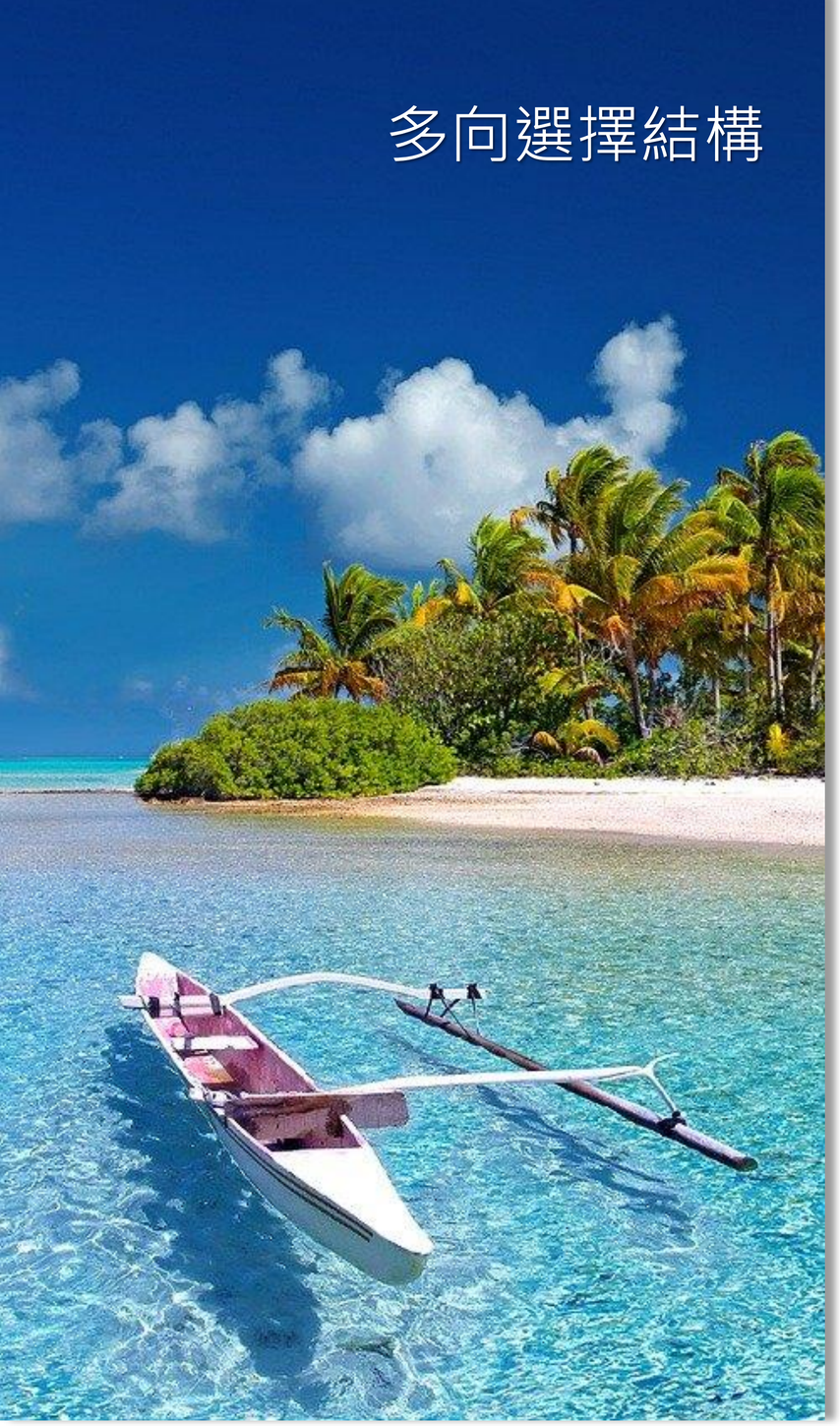


- 語法

```
if(條件式){
    敘述句1;
}else{
    敘述句2;
}
```

```
if(條件式)
    敘述句1;
else
    敘述句2;
```

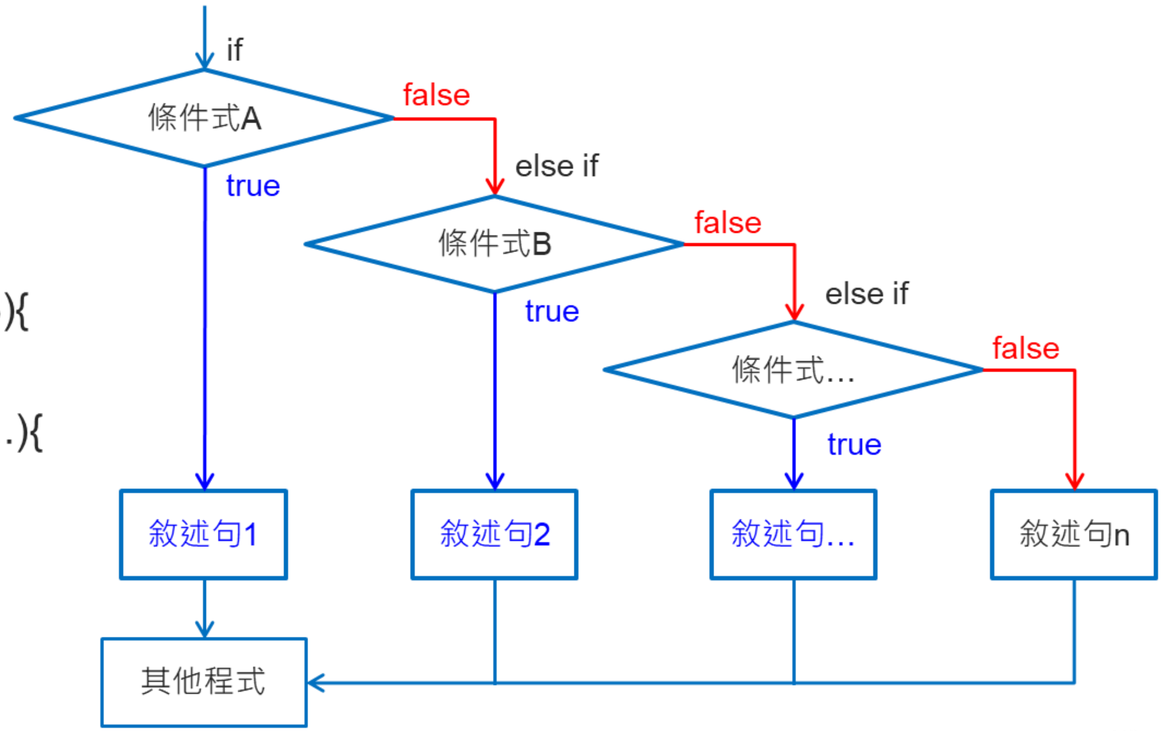




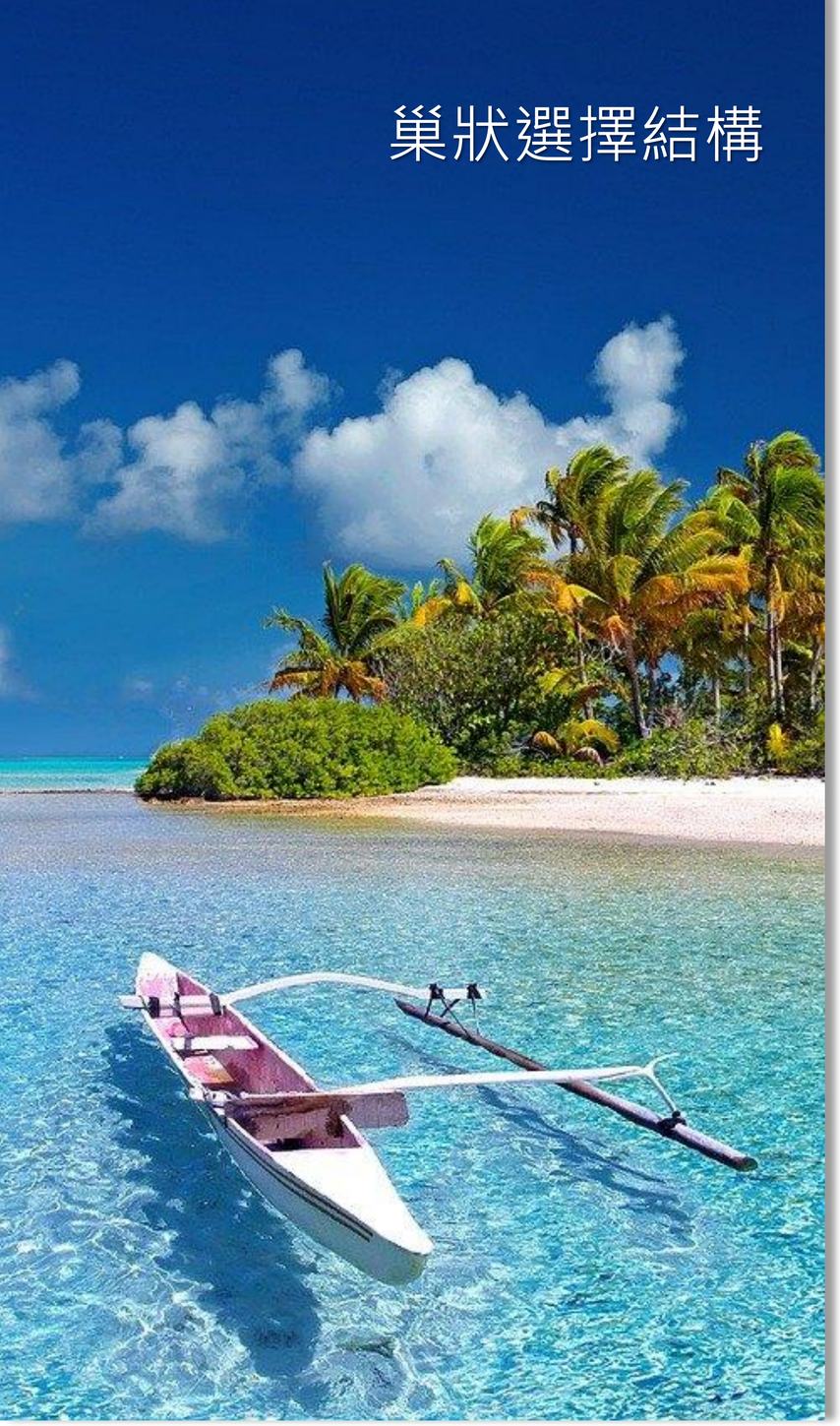
多向選擇結構

- 語法

```
if(條件式A){
   //敘述句1
} else if(條件式B){
   //敘述句2
} else if(條件式…){
   //敘述句…
} else{
  //敘述句n
}
```



巢狀選擇結構

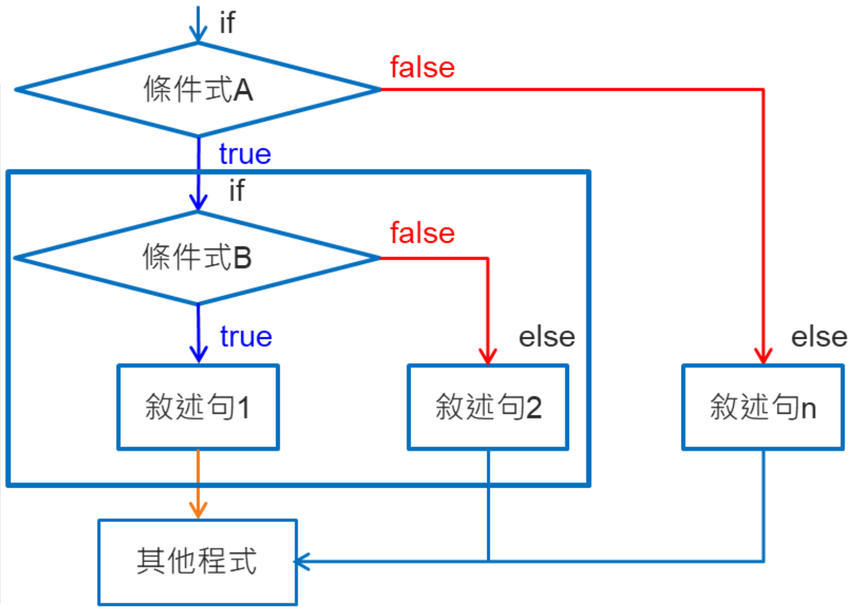


語法

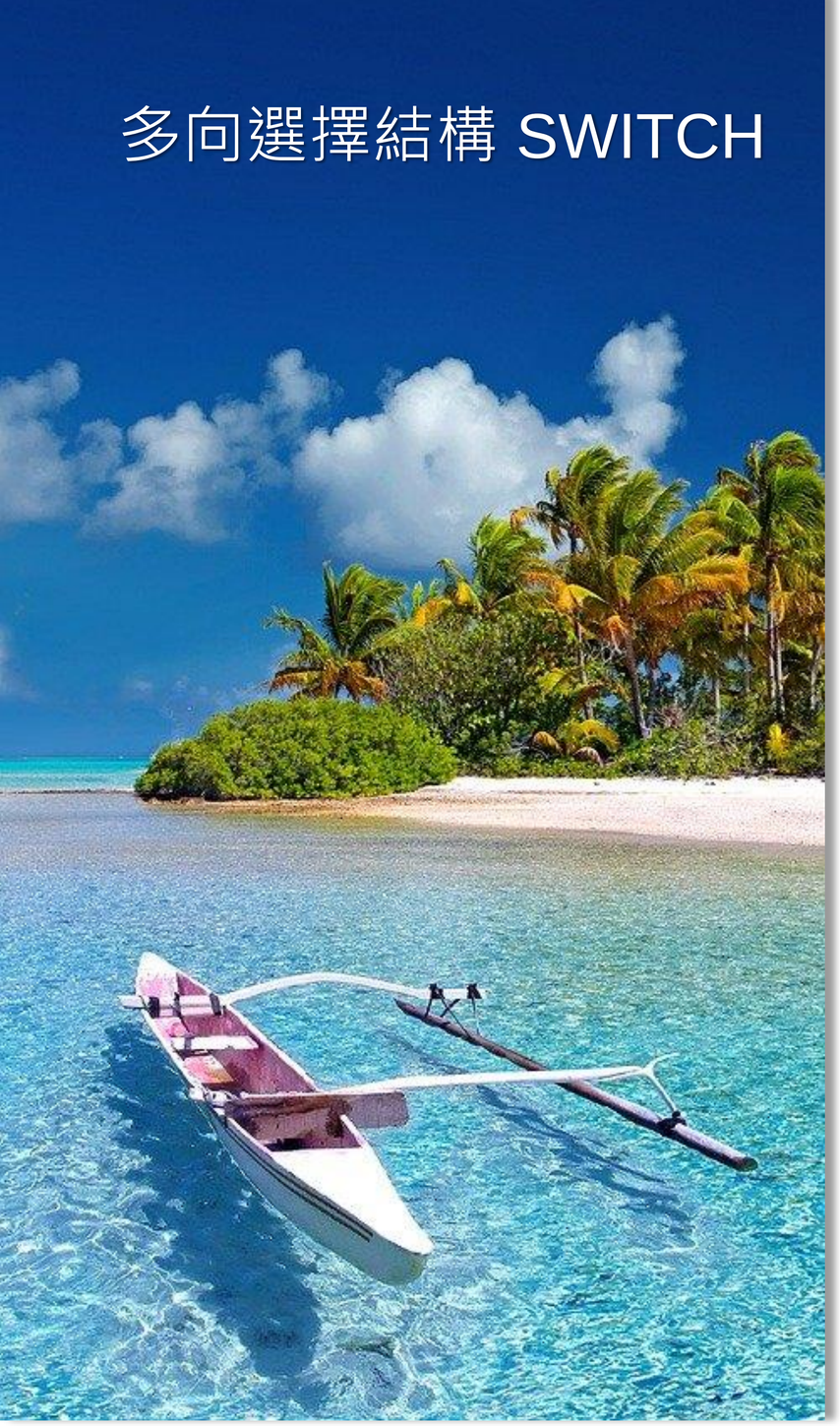
外層 if 結構

```
if(條件式A){
    if(條件式B){
        敘述句1;
    } else{
        敘述句2;
    }
} else{
敘述句n
}
```

內層 if 結構

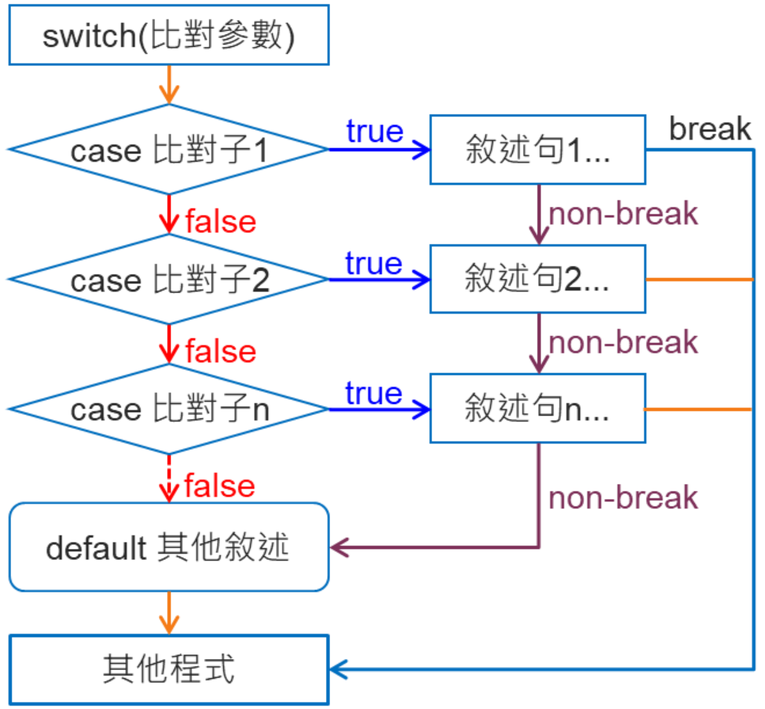


多向選擇結構 SWITCH

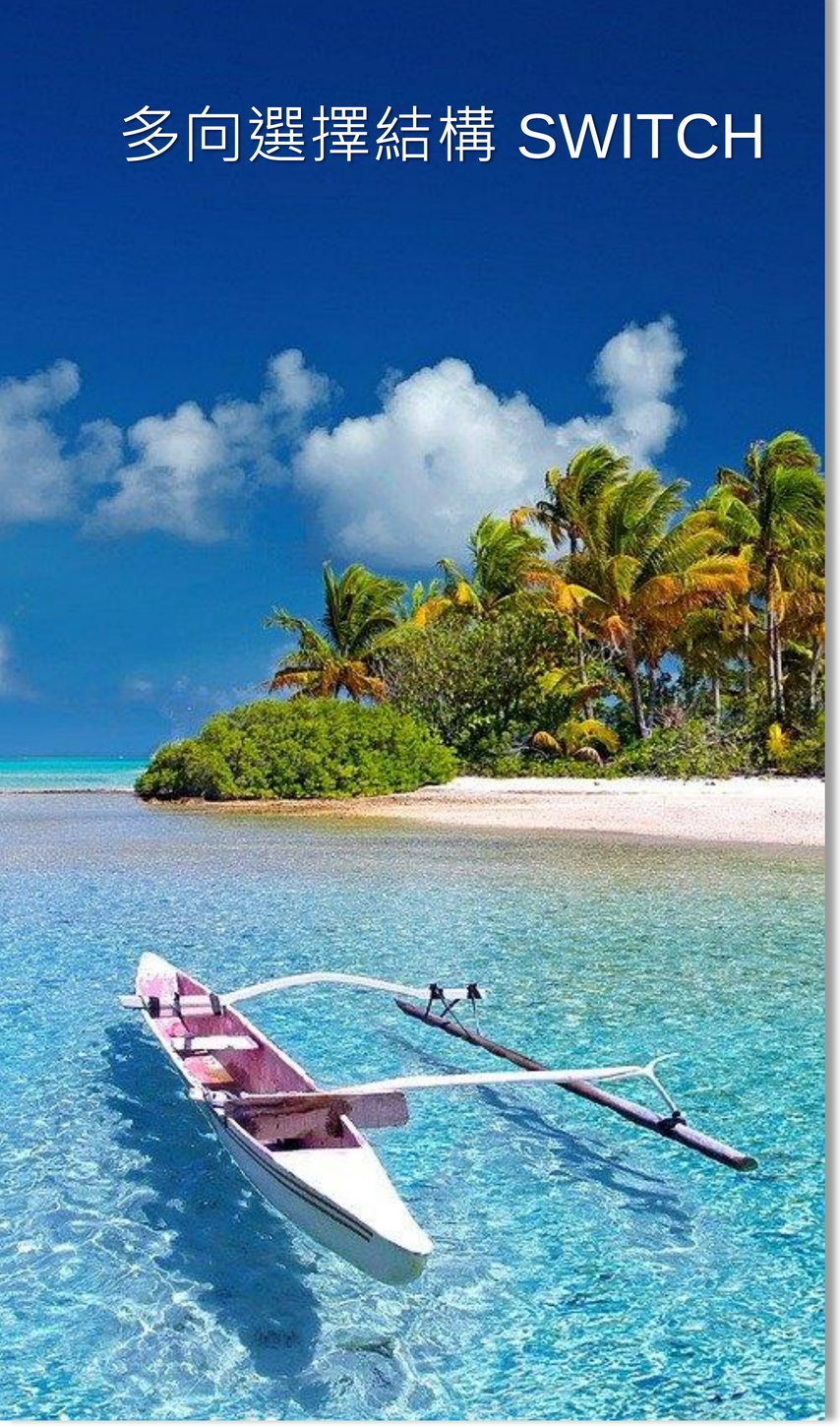


- 語法

```
switch(比對參數){
   case 比對子1:
      程式敘述1;
      [break;]
   case 比對子2:
      程式敘述2;
      [break;]

   …
   [default:]
      …
}
```

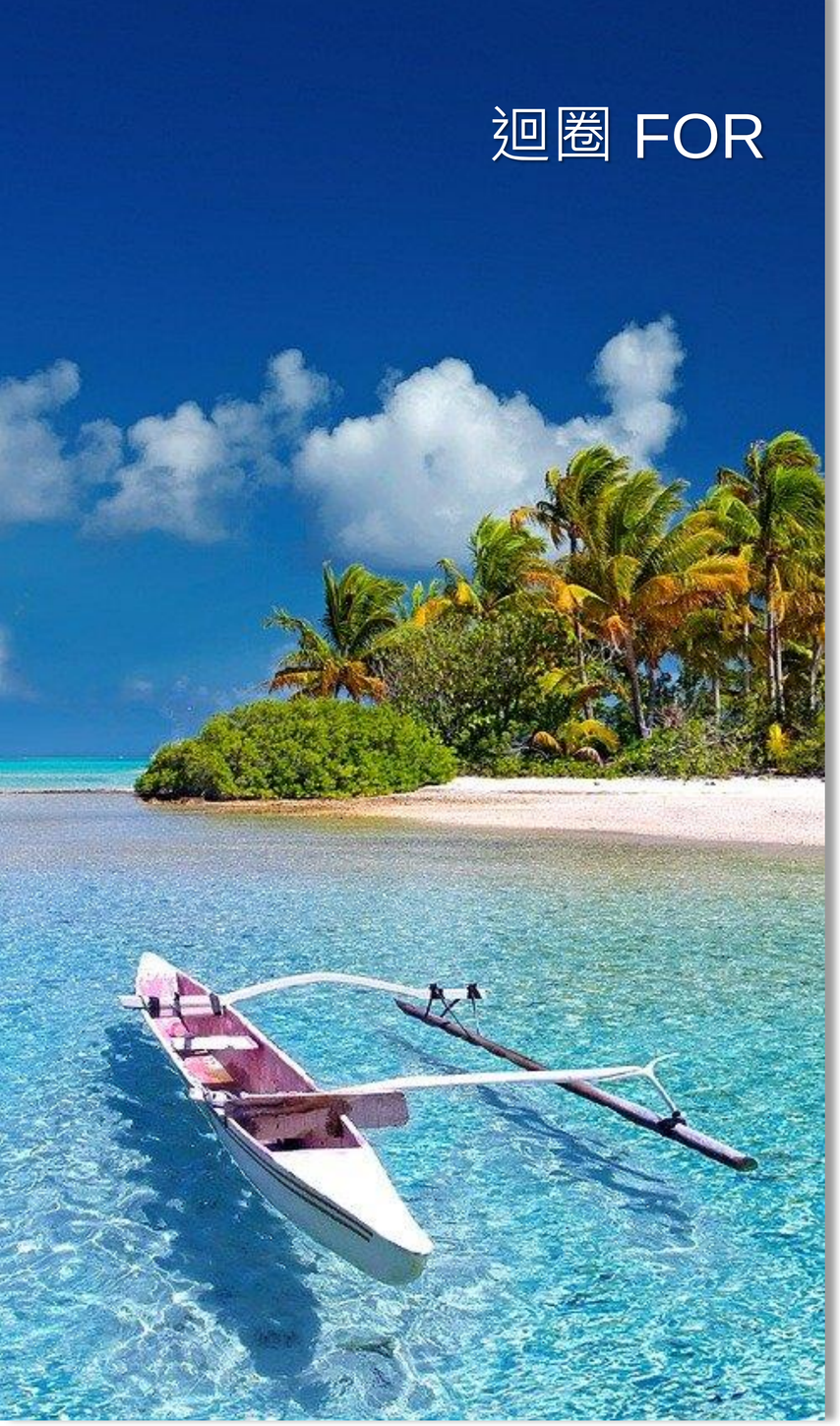


多向選擇結構 SWITCH



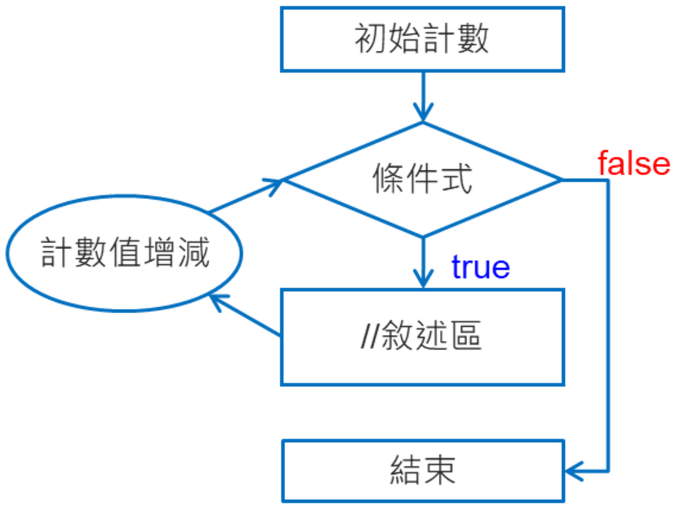
- switch-case 選擇性控制敘述
- 語法中主要成員：
  - 比對參數
    - char、byte、short、int、wrapper class(Character Byte Short Integer) 、String、enum(列舉)
  - case：比對子,比對子必需為字面值或是常數
  - break：控制 switch-case 流程
- 比對時，比對參數與比對子會自動轉換成 int 來進行比對
- default 敘述，default 敘述不一定要放在 switch 述式的最後，如果不放在最後的話，則需要加上 break 敘述
- break 敘述省略，程式執行會將下一個比對內容一併執行

迴圈 FOR

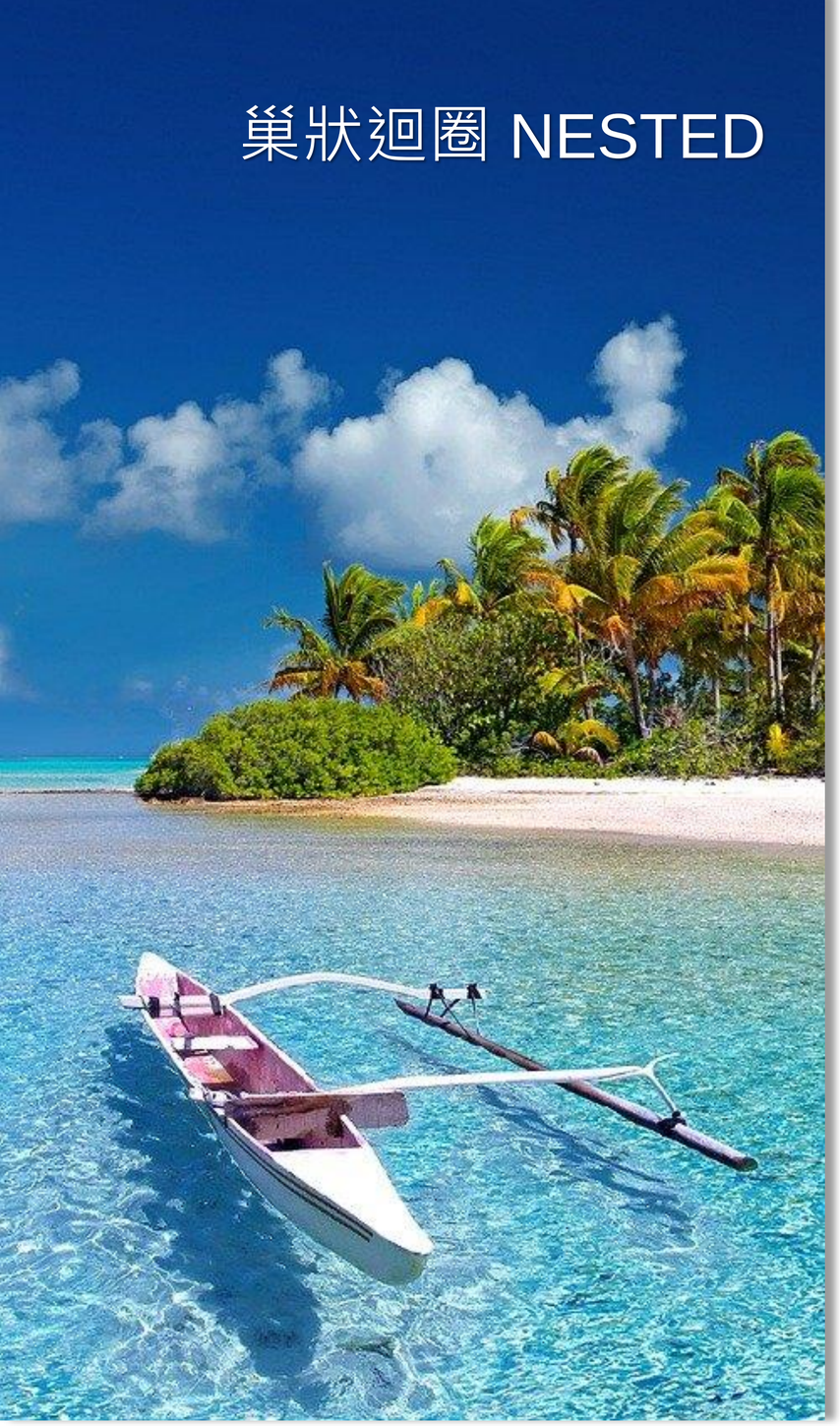


宣告或指派運算式　條件運算式　算術運算式

```
for ( 初始計數值 ; 條件式 ; 增減值 ) {
        敘述區;
}
```



巢狀迴圈 NESTED



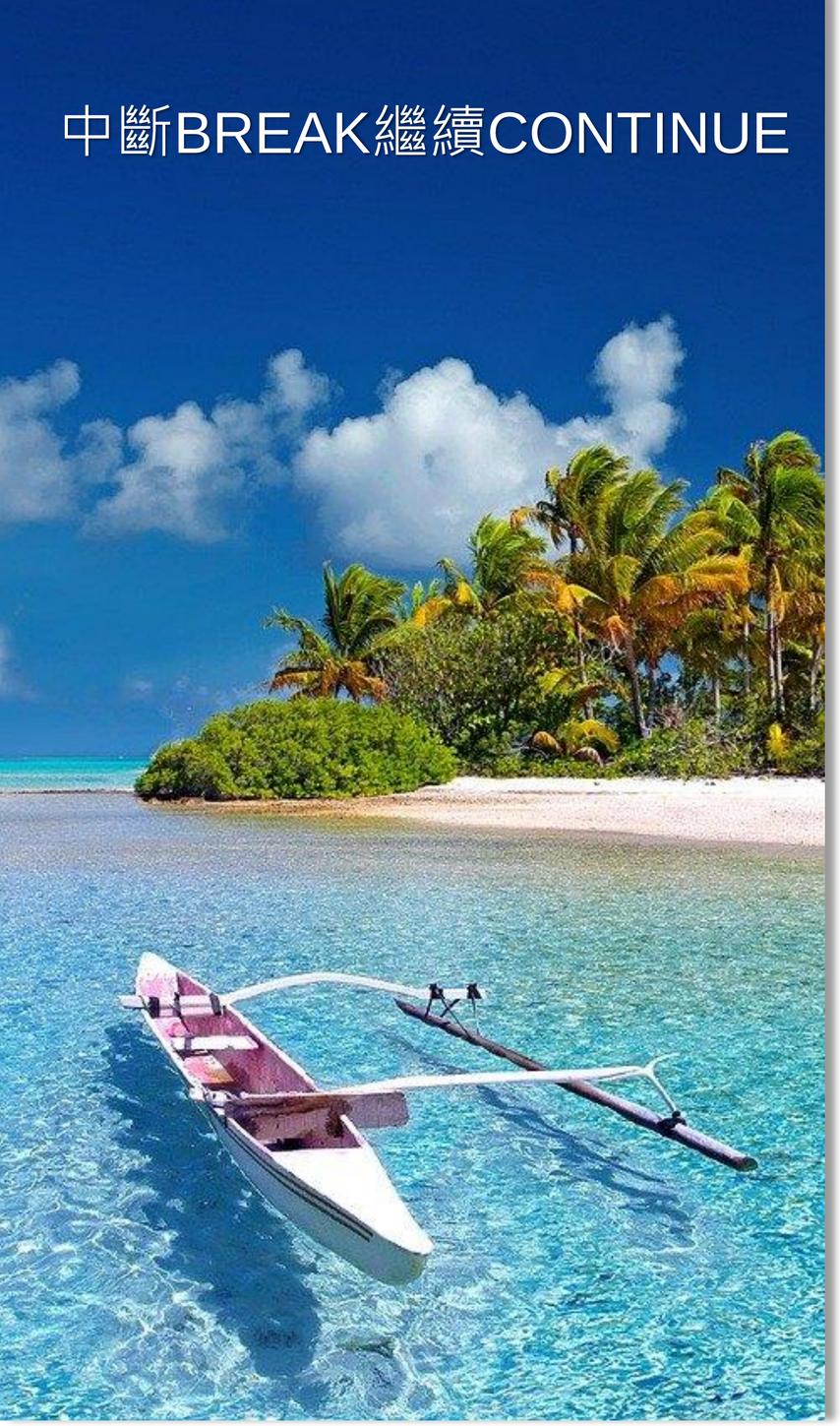
語法

外迴圈

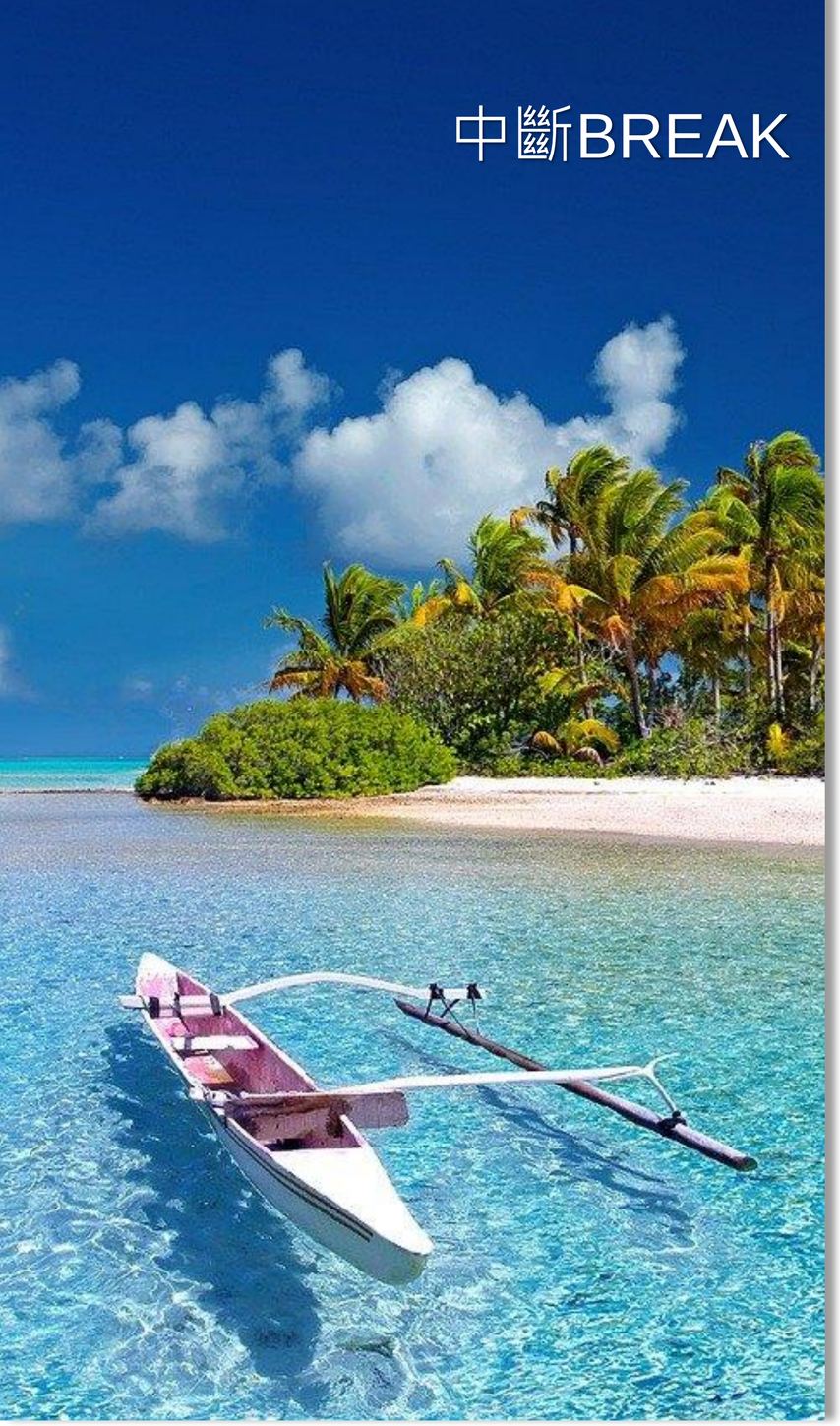
```
for ( 初始計數值1 ; 布林運算式1 ; 增減值1 ) {

    for ( 初始計數值2 ; 布林運算式2 ; 增減值2 ) {
      敘述區;
    }

}
```

內迴圈

中斷BREAK繼續CONTINUE



- break、continue 與 lable 標籤設定可以讓流程控制的程式更具彈性
- Java 不支援 goto 敘述，提供 break 和 continue 及標籤，藉以達到類似 goto 效果
- break：中斷迴圈執行
- continue：迴圈執行時可以略過 continue 後面程式碼的執行，回到迴圈開頭繼續迴圈的執行
- lable：可以配合 break 或 continue 運作



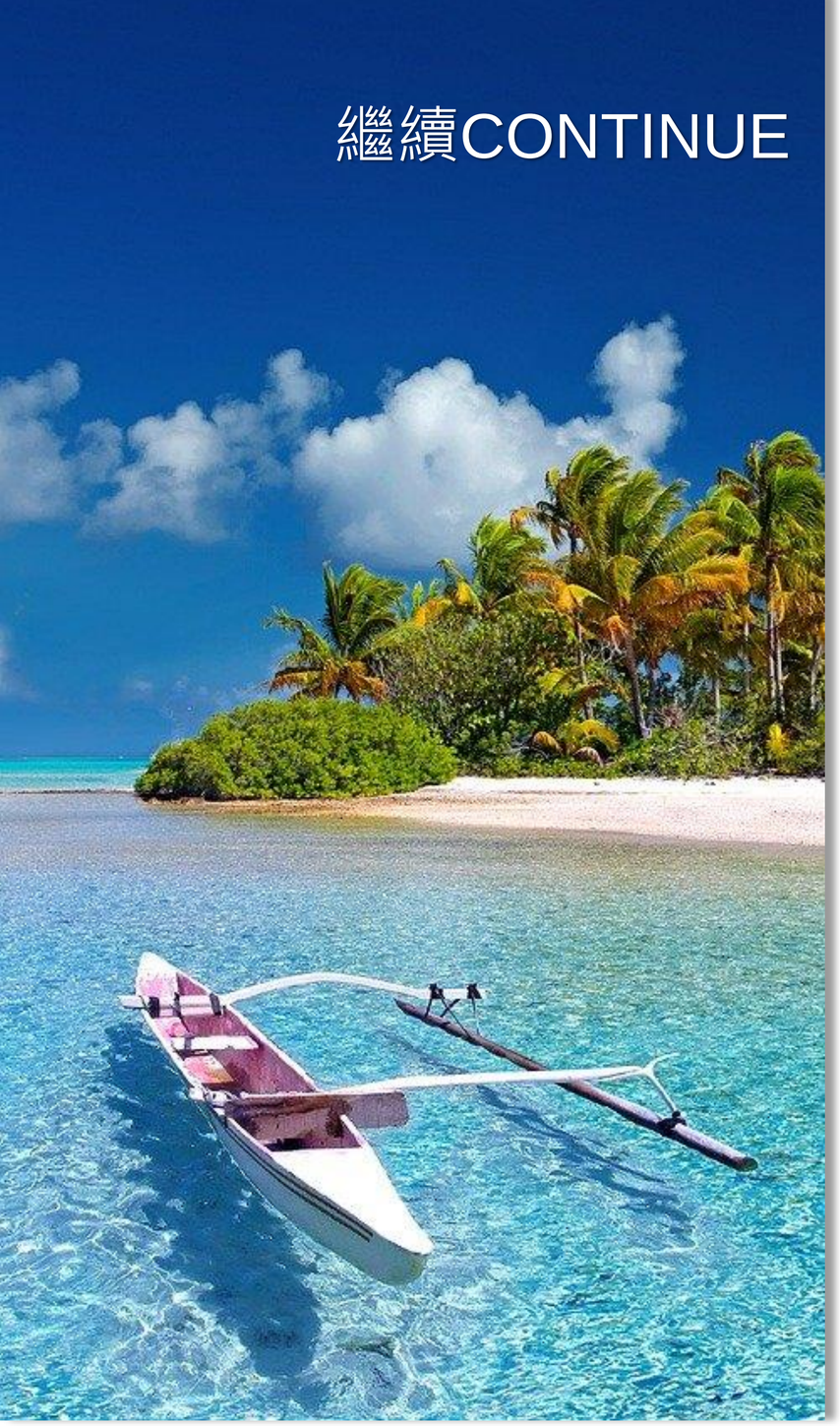
中斷BREAK

- break 中斷迴圈執行
- 只能中斷內迴圈,外迴圈需使用label
- 語法

```
for ( 初始計數值1 ; 布林運算式1 ; 增減值1 ){
    …
    if(條件式)
            break;
    …
}
```

條件成立時，中斷迴圈執行

繼續CONTINUE

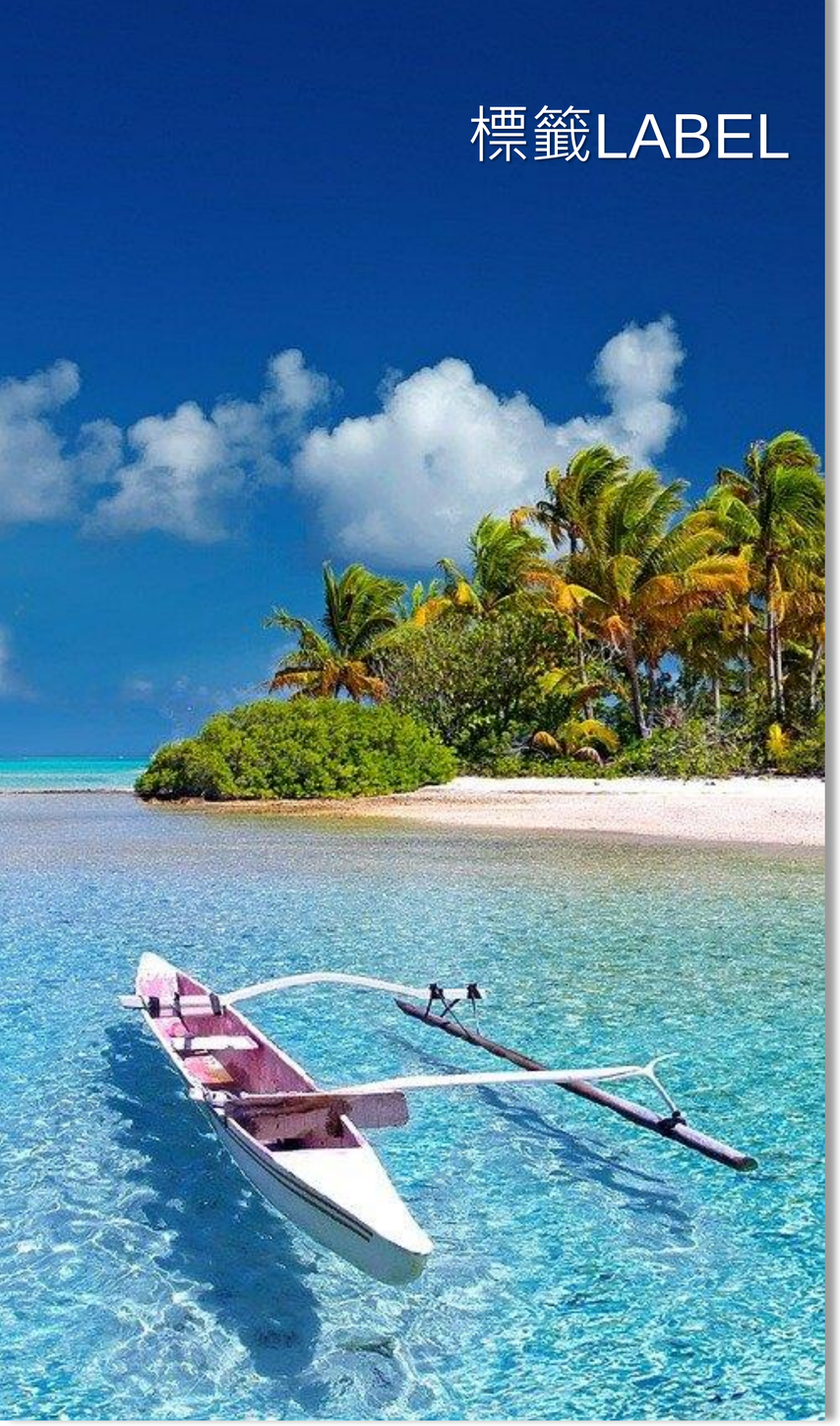


- continue 迴圈執行時可以略過 continue 後面程式碼的執行，回到迴圈開頭繼續迴圈的執行
- 語法

```
for ( 初始計數值1 ; 布林運算式1 ; 增減值1 ){
    …
    if(條件式)
            continue;
    …
}
```

條件成立時，回到迴圈繼續下一個執行

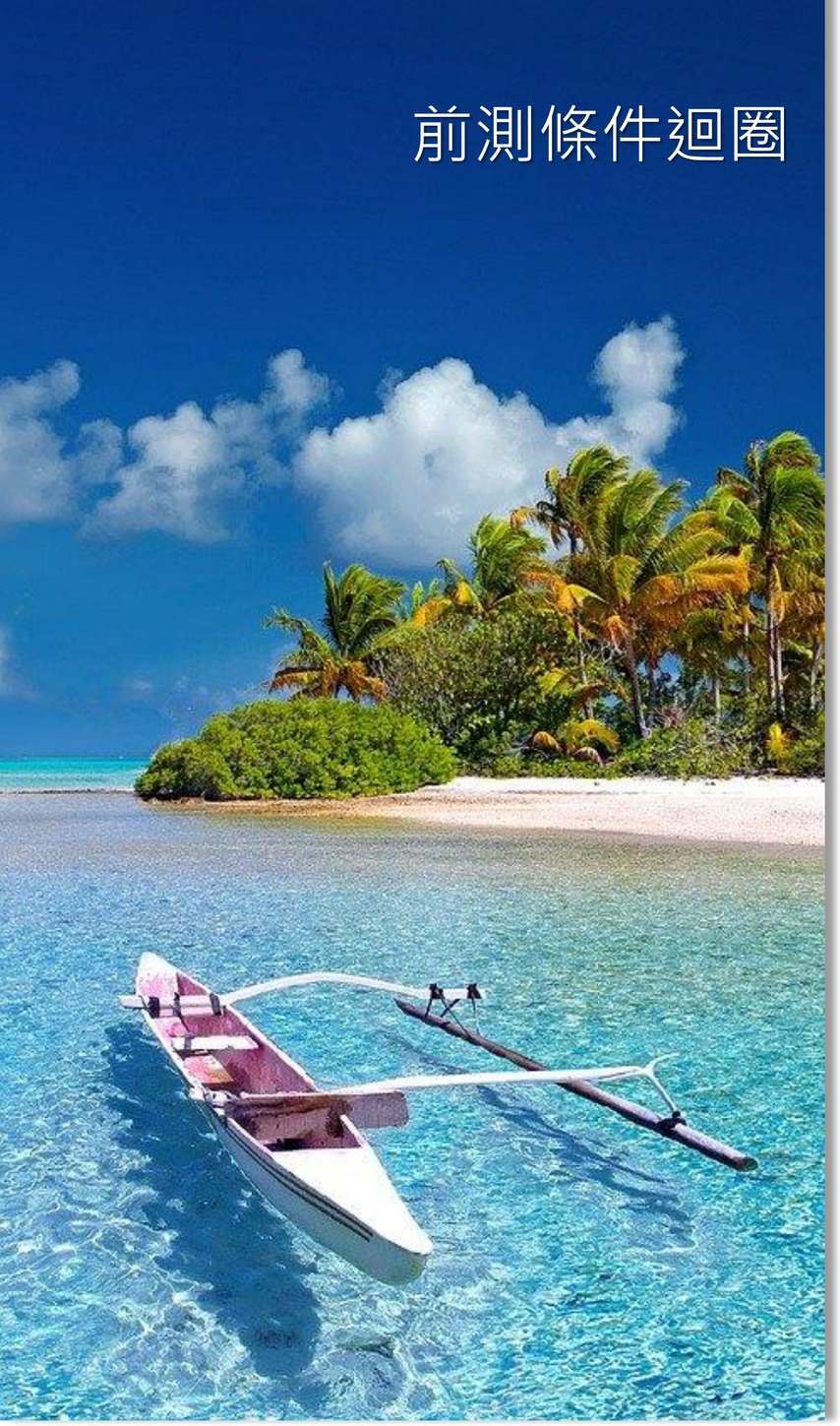
標籤LABEL



- 標籤代表迴圈的標記，使用時可以配合 break 或 continue
- 標示迴圈名稱方式
  - Lable_name :

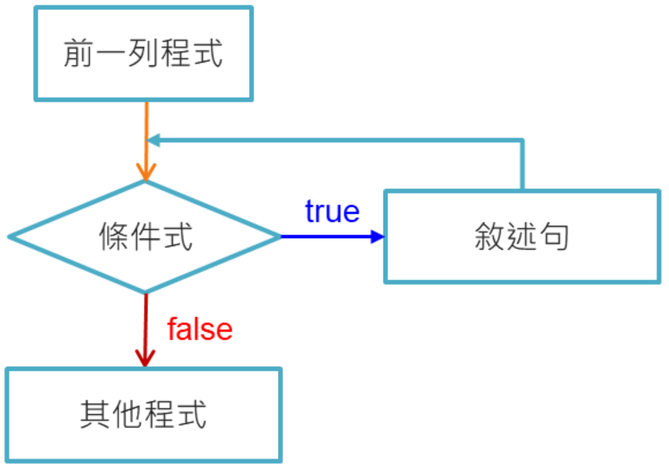
    迴圈敘述...
  - 名稱後方接【：】(冒號)
  - 需除註解以外，緊接迴圈敘述

前測條件迴圈

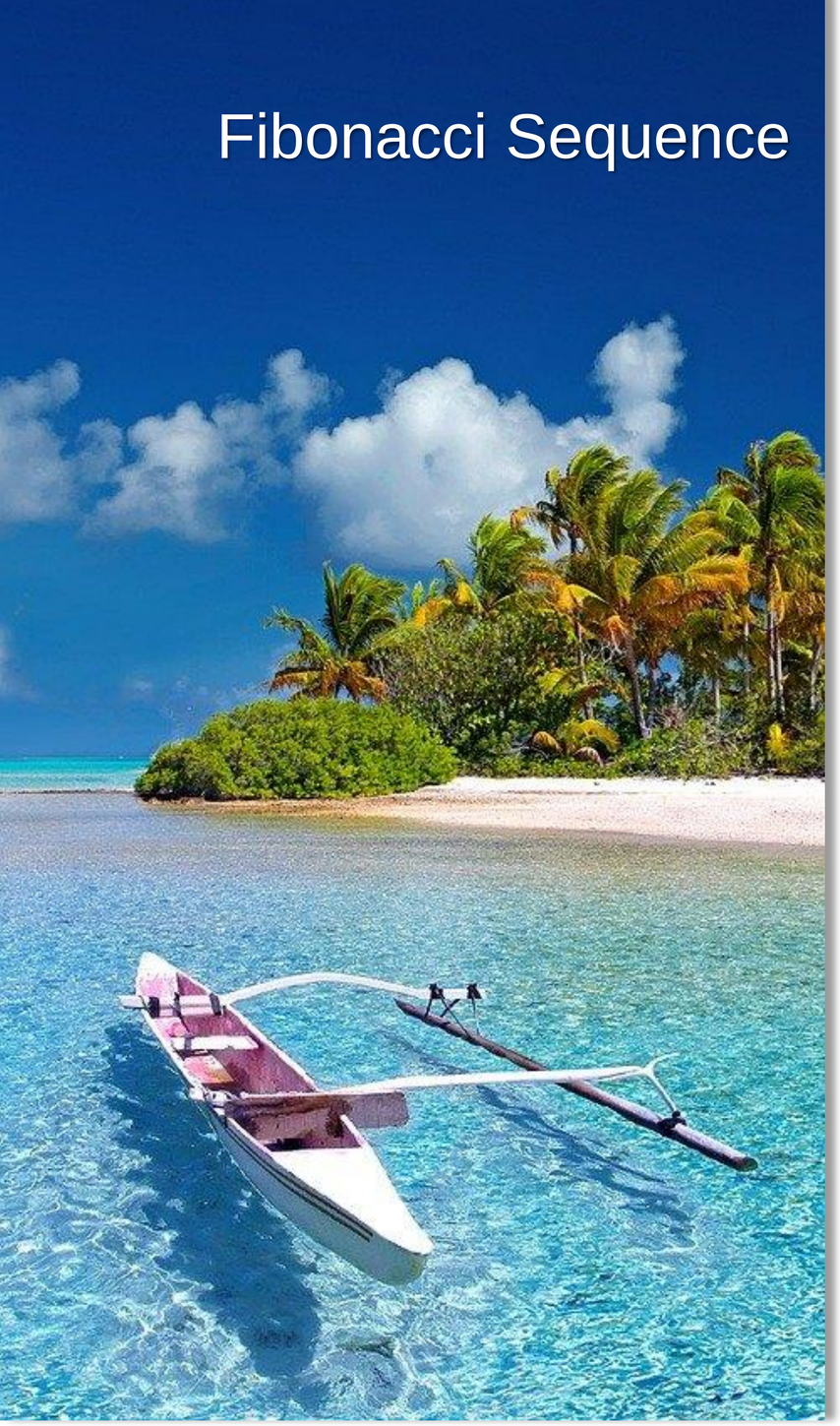


- while 迴圈是屬於前測型的迴圈，在不可預測執行的次數情況下使用
- 語法

```
while ( 條件式 )
{
    敘述句;
}
```



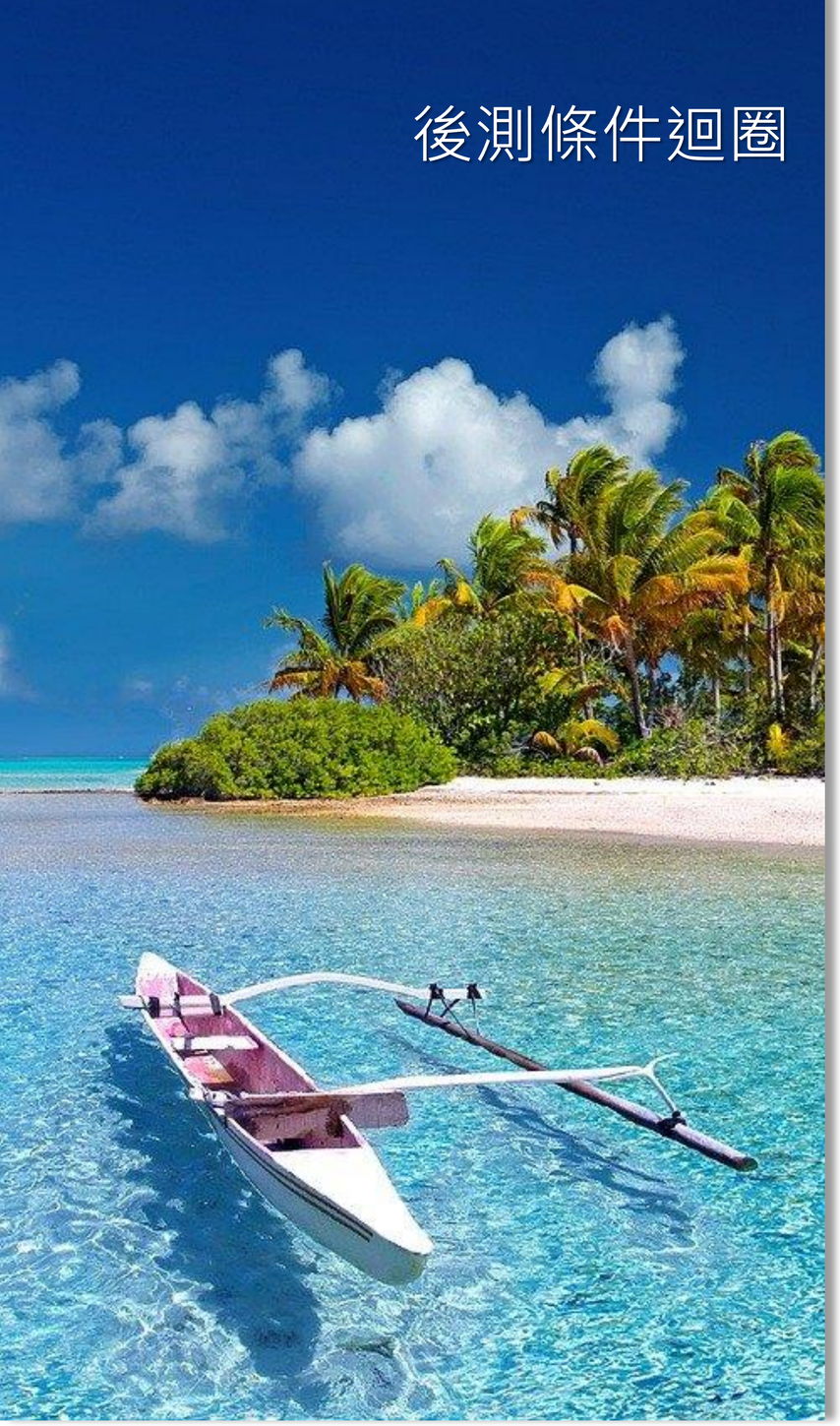
Fibonacci Sequence



操作重點

1. 寫一個程式，輸出小於指定的數字的費伯那西數列 (Fibonacci Sequence)
   - 取的要列印的最大範圍

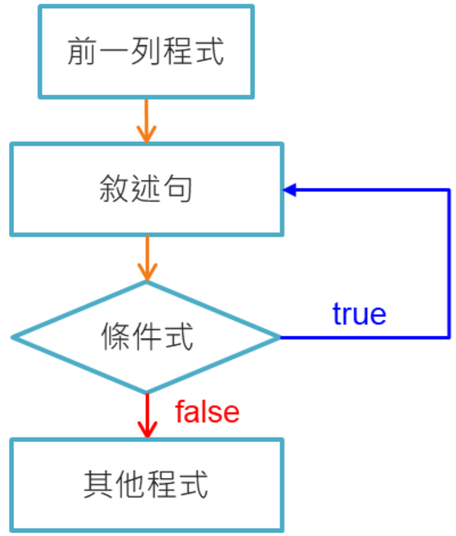
   費伯那西數列 (Fibonacci Sequence)
   - 1、1、2、3、5、8、13、21、34、55、89、144 ……
   - 每個數字都是前兩個數字的和



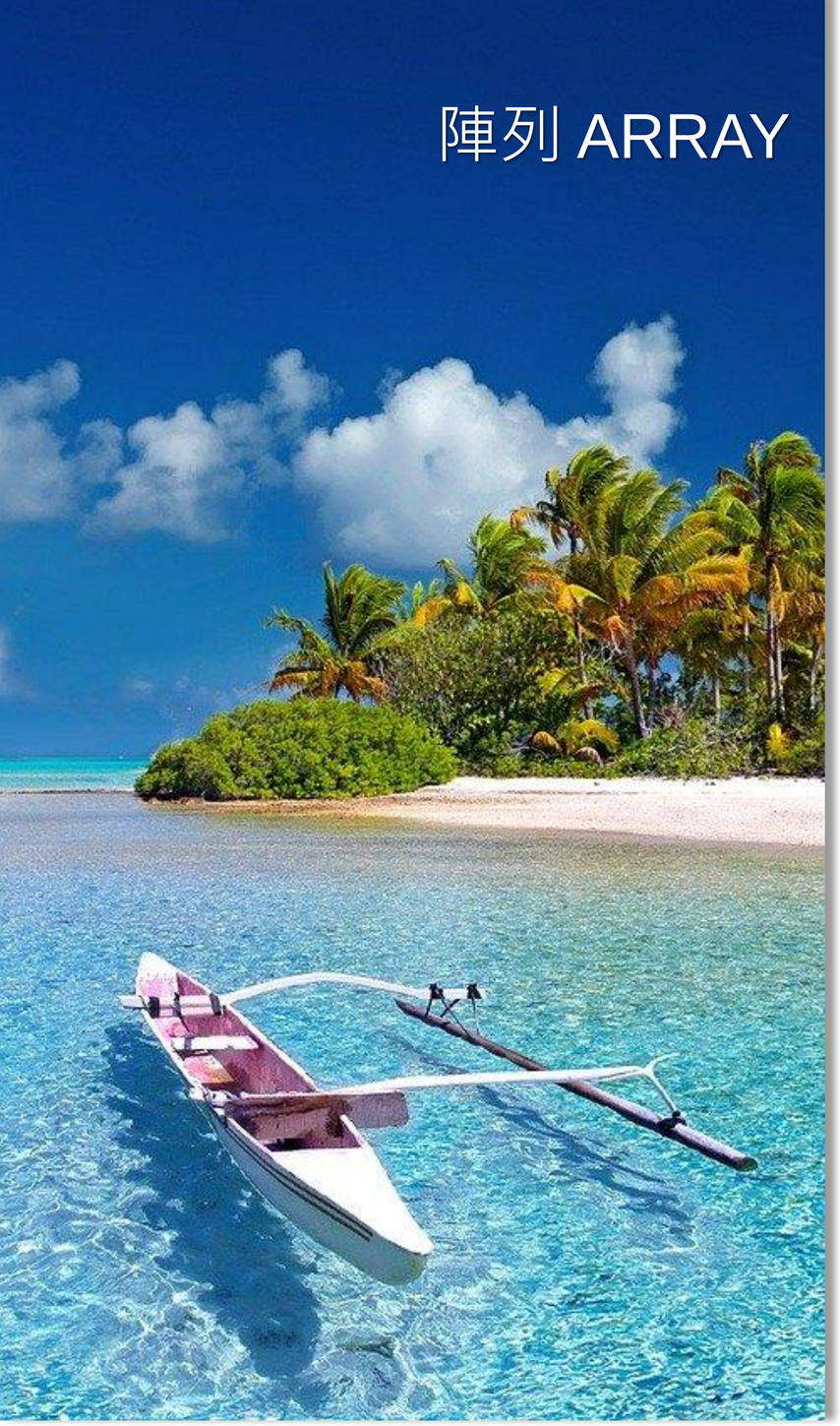
後測條件迴圈

- do-while 迴圈是屬於後測型的迴圈，也是在不可預測執行的次數情況下使用，但程式定會執行一次
- 語法

```
do
{
    敘述句;
} while (條件式) ;
```

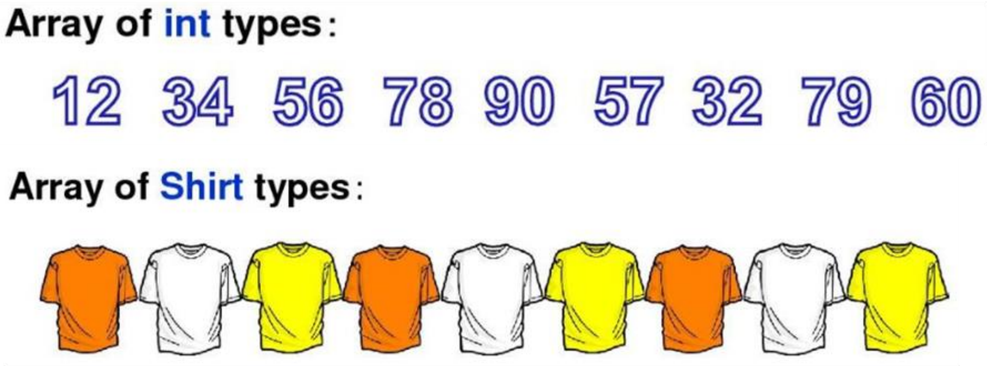


陣列 ARRAY

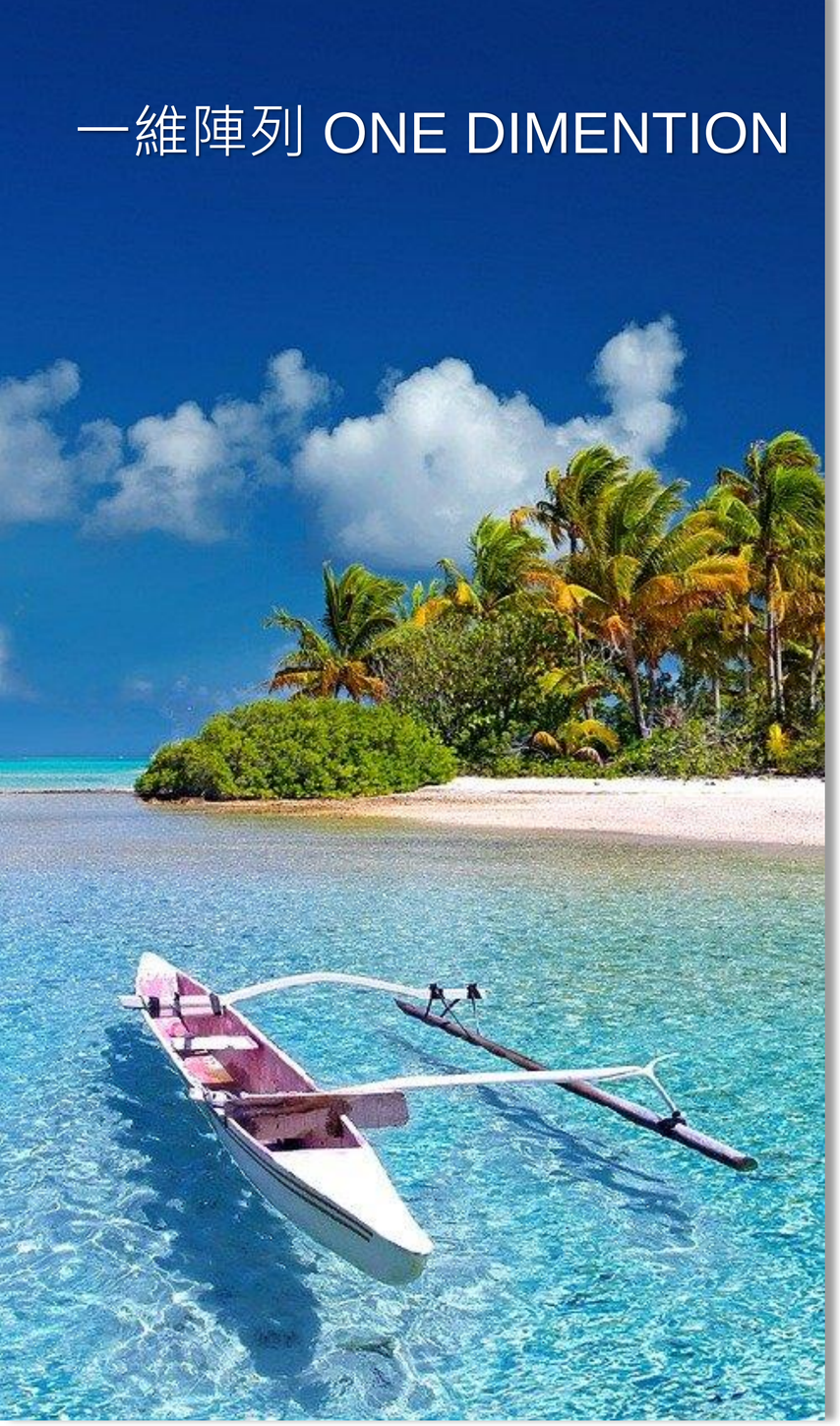


陣列 ARRAY

- 陣列
  - 將同樣型別的東西聚集在一起，用相同名稱來參考一群物件
  - 參考型別物件
- 使用步驟
  - 宣告
  - 實體化
  - 初始化



一維陣列 ONE DIMENTION



宣告 DECLARATION

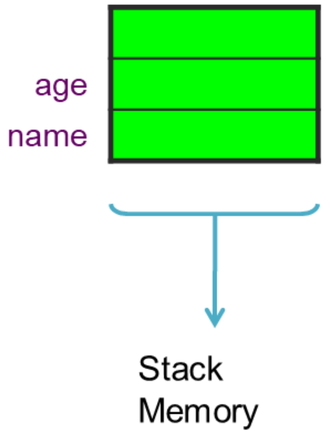
- 語法

type [ ] array_name;

type array_name [ ];

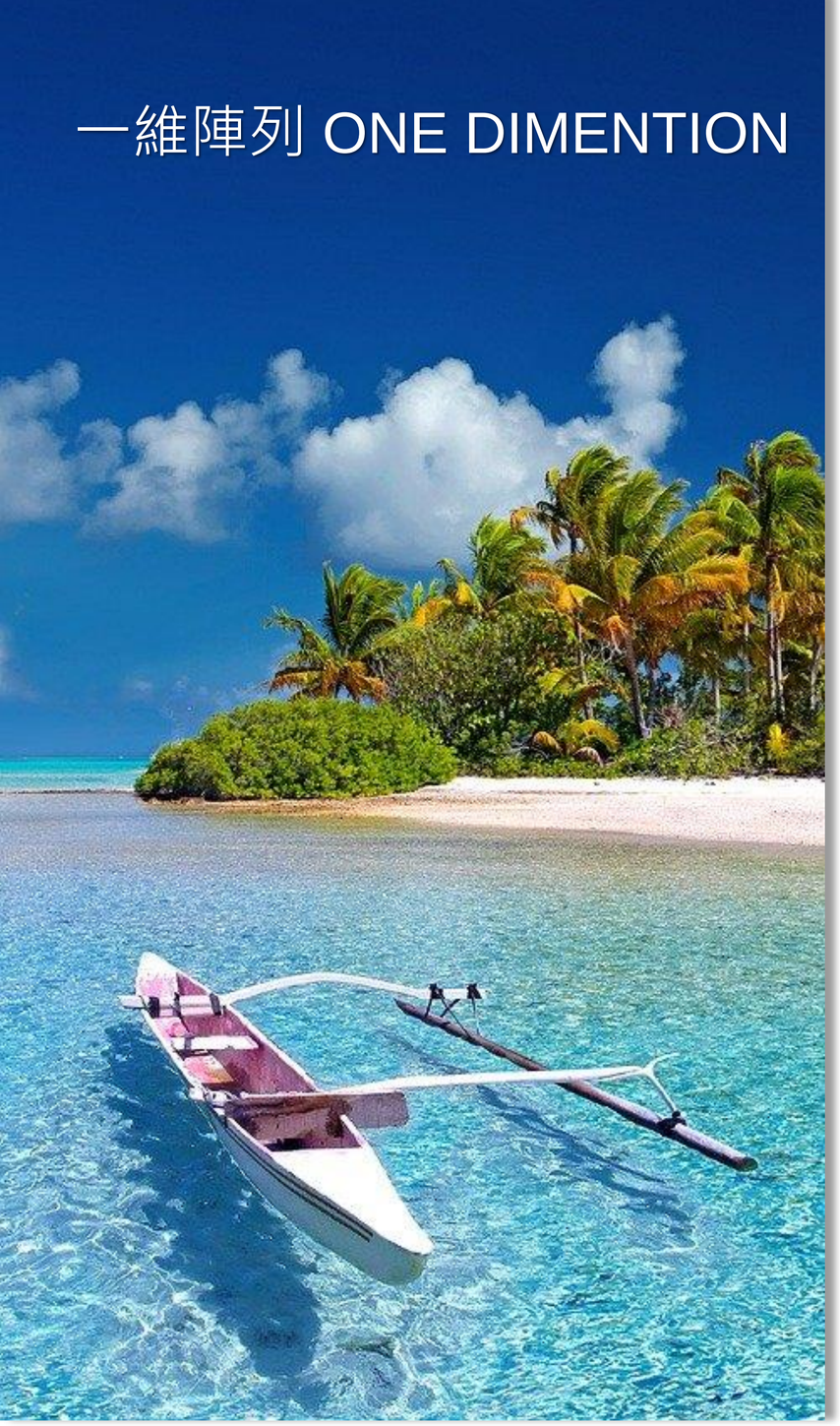
Ex: int [ ] age;

String name[ ];



Stack
Memory

一維陣列 ONE DIMENTION



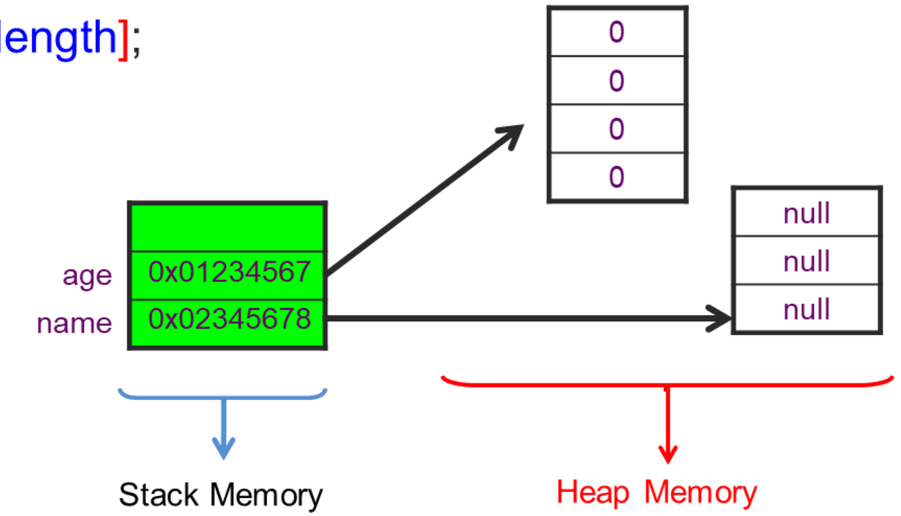
實體化 INSTANTIATED

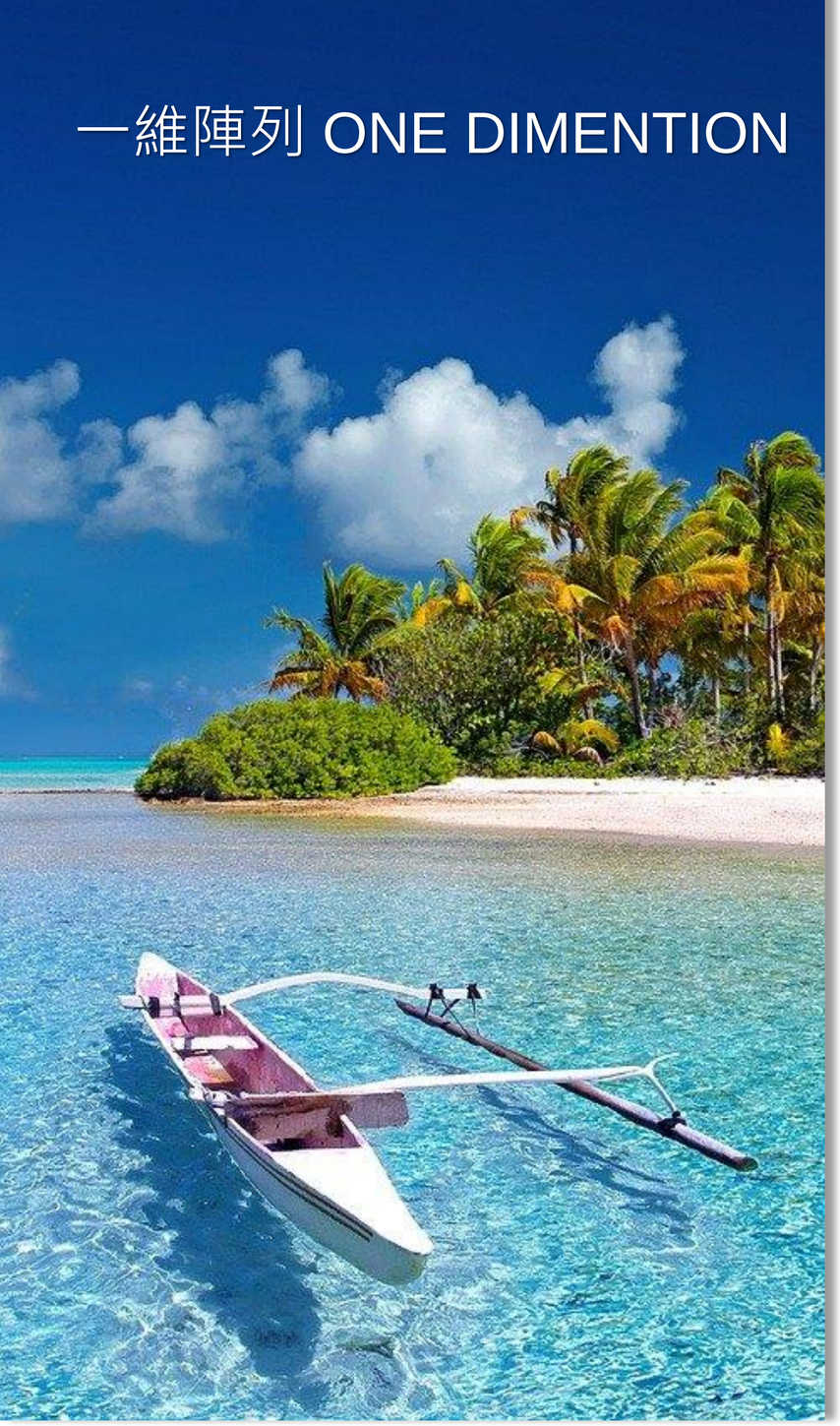
◆ 語法：

```
array_name = new type[length];
```

Ex: age = new int [4];

name = new String [3];




一維陣列 ONE DIMENTION

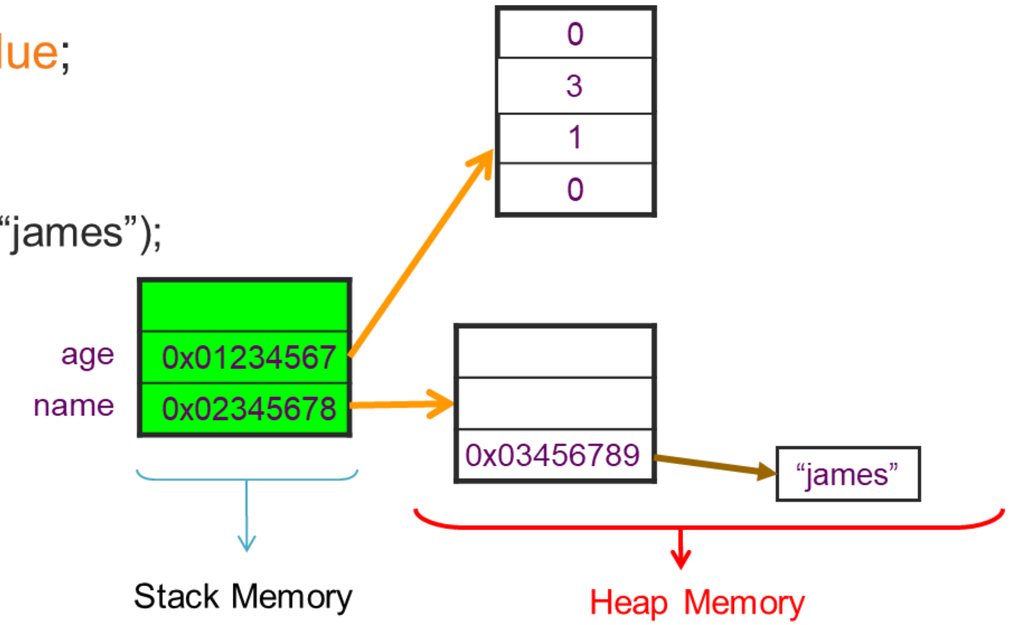
初始化 INITIALIZATION

◆ 語法：

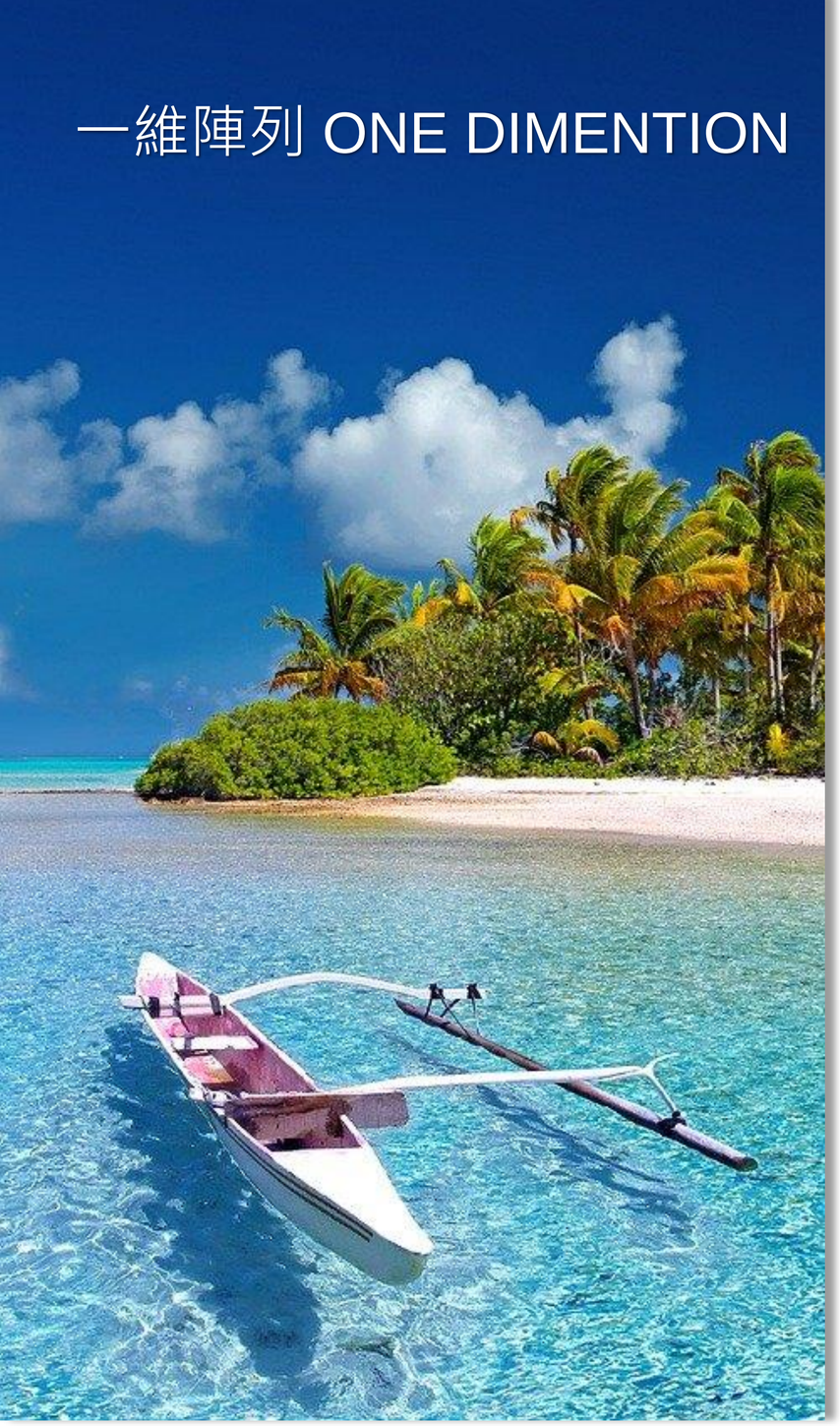
array_name [index] = value;

Ex: age [2] = 3;

name [0] = new String(“james”);



一維陣列 ONE DIMENTION

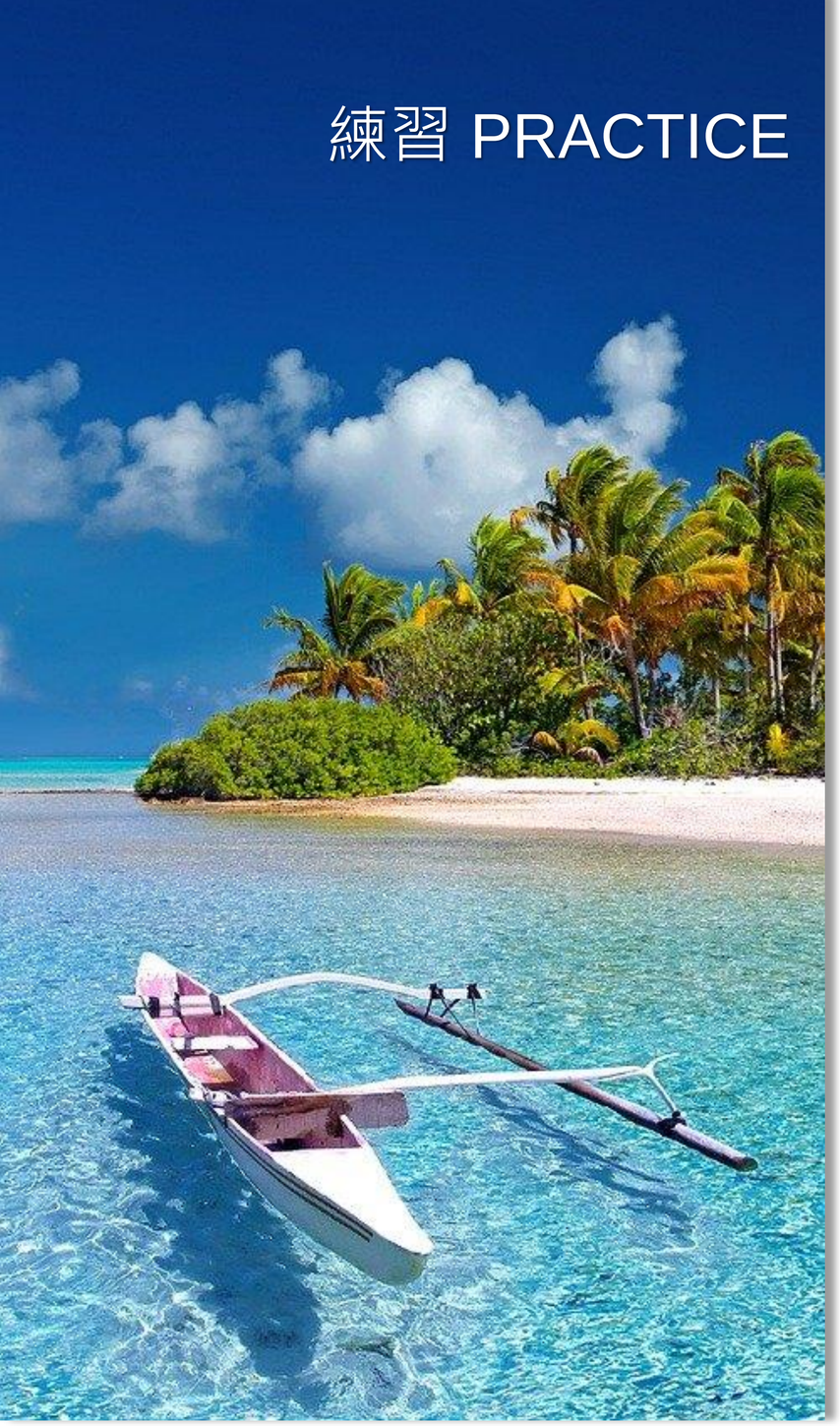


建構方式

宣告	int[] i;	int[] i = new int[3];	int[] i;	int[] i = new int[ ] {1, 3, 5}; int[] i = {1, 3, 5};
實體化	i = new int[3];		i = new int[ ] {1, 3, 5};	

初始化	i[0] = 1;
	i[1] = 3;
	i[2] = 5;
		i[0] = 1;
		i[1] = 3;
		i[2] = 5;		

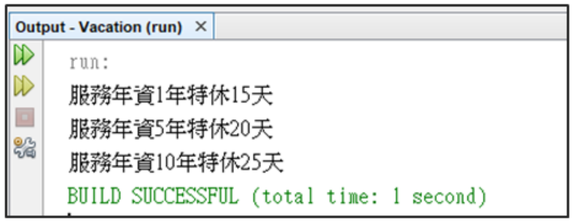
練習 PRACTICE



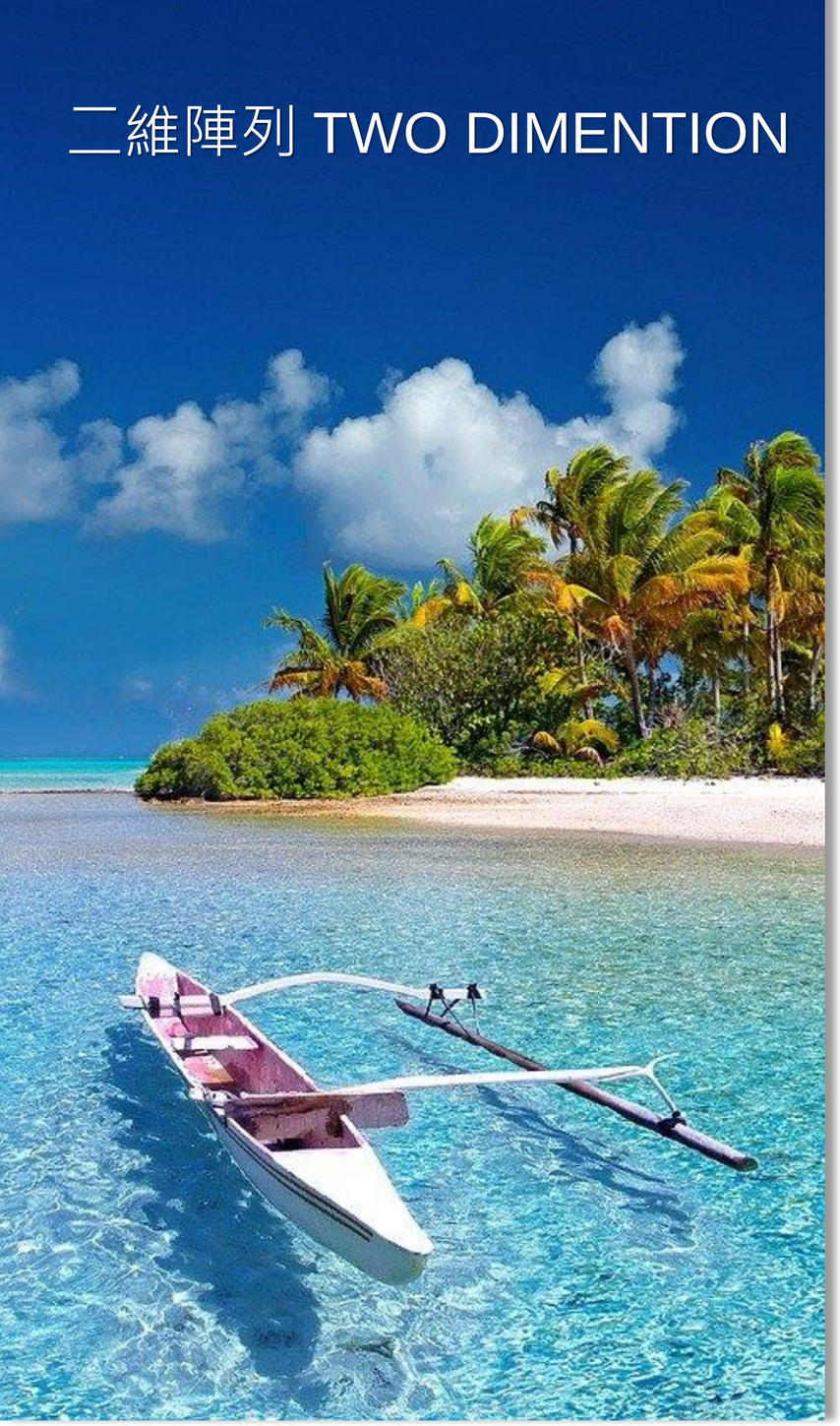
操作重點

1. 撰寫 VacationScale.java 類別
   - 某公司服務年資及特休的規則如右：
   - yearOfService 屬性代表服務年資
   - int[ ] vacationDays 屬性儲存年資與特休對應關係
   - 建構子中設定 vacationDays 的值
   - displayVacationDays( ) 方法特定服務年資對應的特休天數
2. 撰寫 VacationScaleTest.java 類別
   - 建立 VacationScale 物件
   - 變更物件服務年資為 1 年、5 年、 10 年
   - 顯示特休天數

服務年資 yearOfService	特休天數
未滿一年	10
1年~3年	15
4年~5年	20
6年以上	25

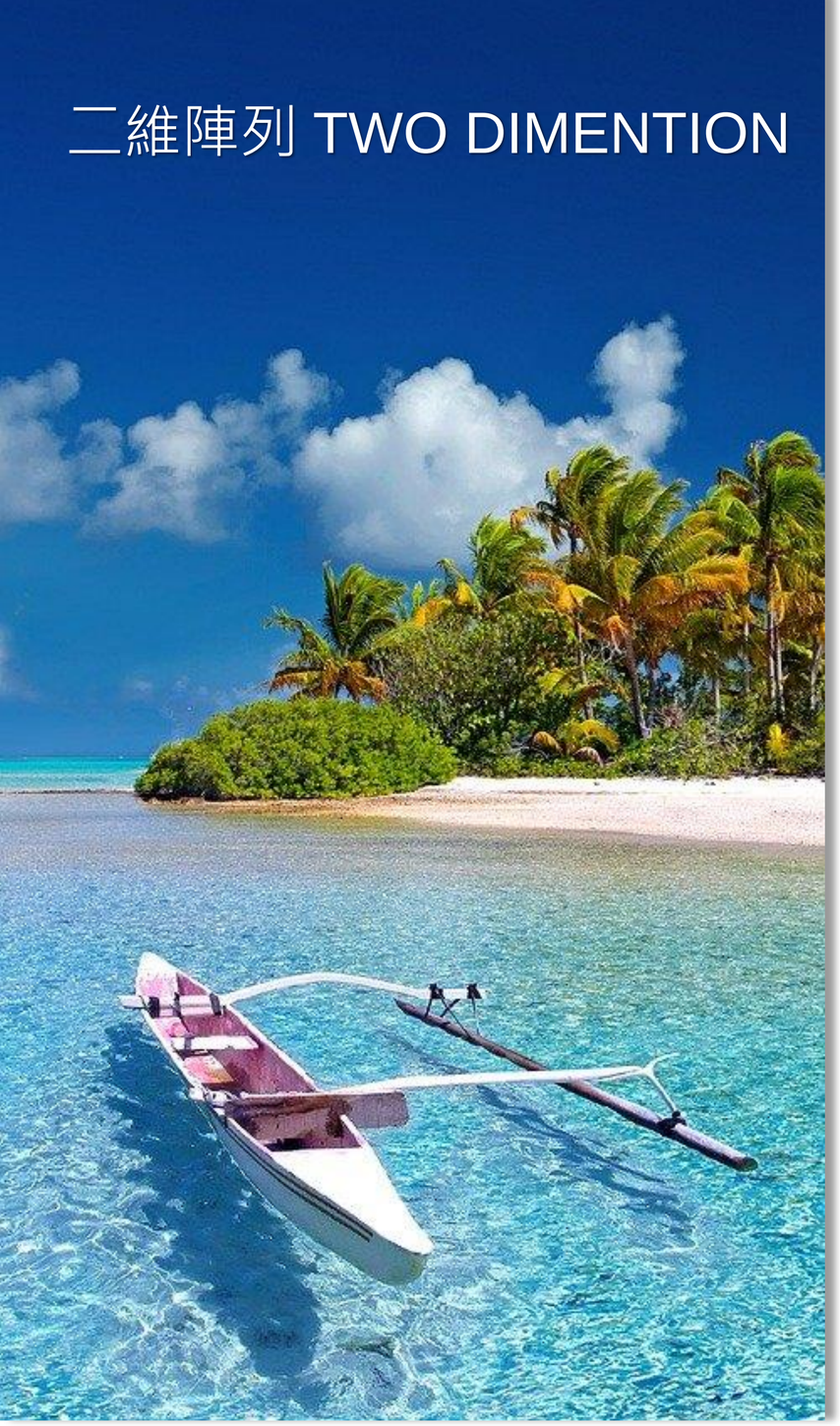


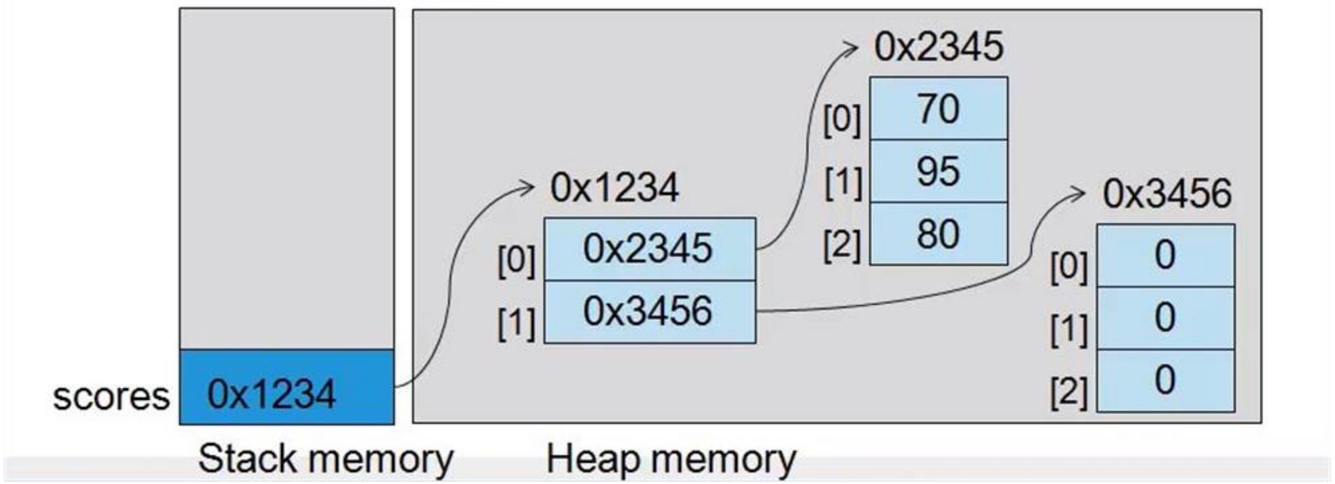
二維陣列 TWO DIMENTION



	Quarter1	Quarter2	Quarter3	Quarter4
Year1	sales[0][0]	sales[0][1]	sales[0][2]	sales[0][3]
Year2	sales[1][0]	sales[1][1]	sales[1][2]	sales[1][3]
Year3	sales[2][0]	sales[2][1]	sales[2][2]	sales[2][3]
Year4	sales[3][0]	sales[3][1]	sales[3][2]	sales[3][3]
Year5	sales[4][0]	sales[4][1]	sales[4][2]	sales[4][3]

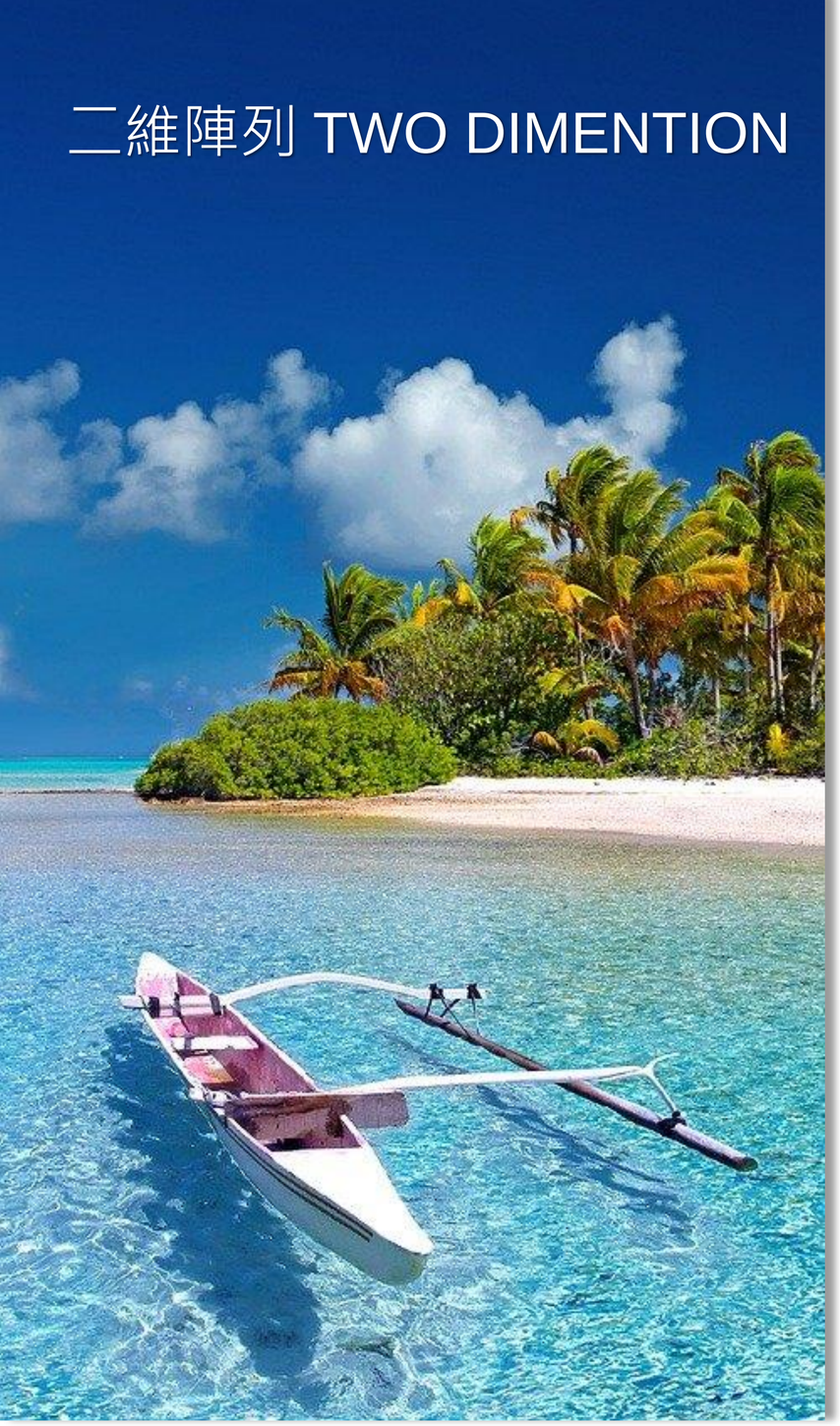
二維陣列 TWO DIMENTION






```
int scores[ ][ ]=new int[ ][ ]{{70,95,80},{0,0,0}};
```

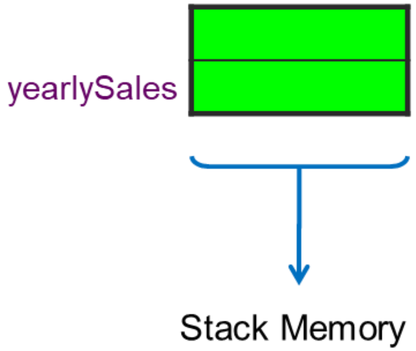
二維陣列 TWO DIMENTION



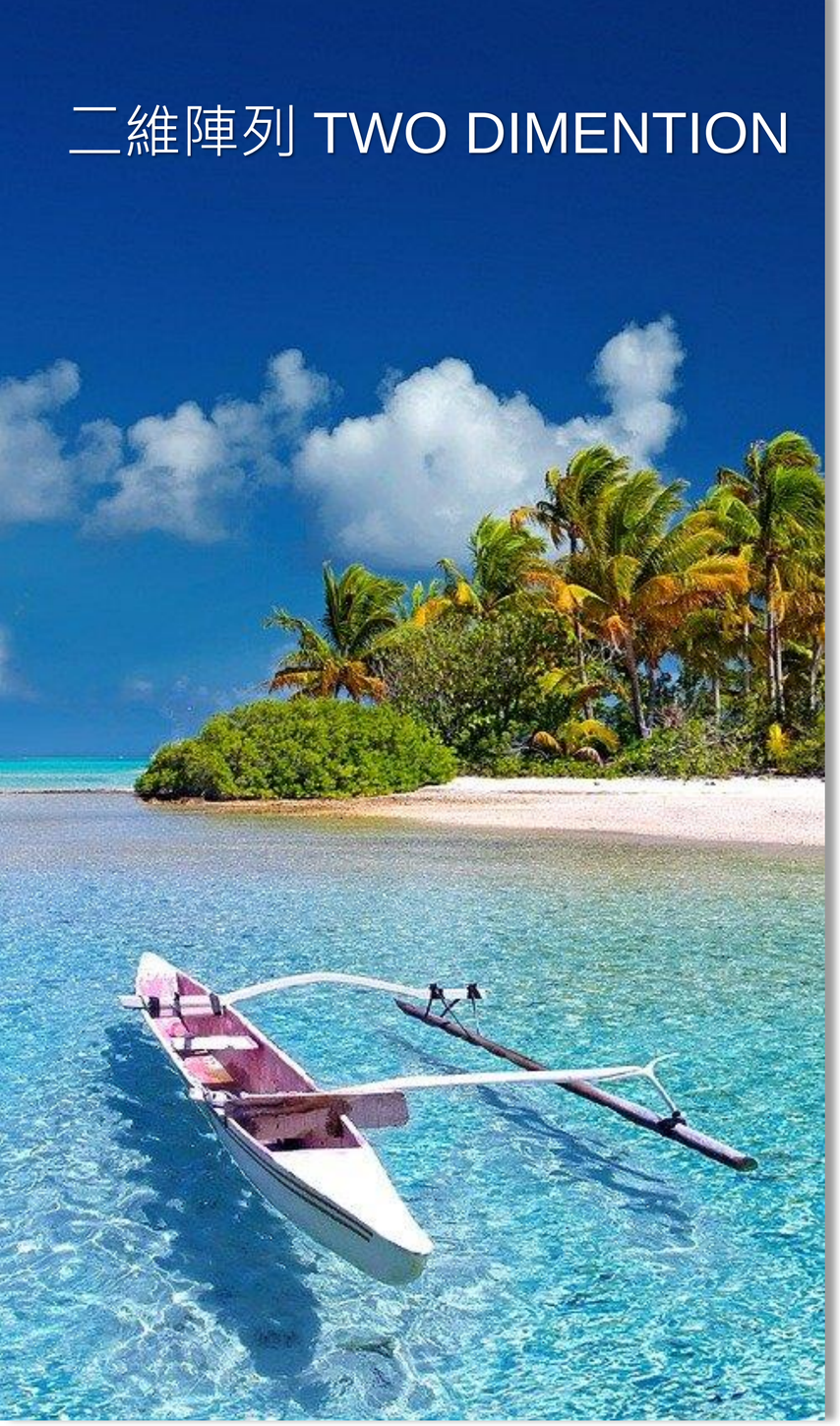
宣告 DECLARATION

- 語法

```
type [ ][ ] array_name;
type array_name [ ][ ];
Ex:
    int [][] yearlySales;
```



二維陣列 TWO DIMENTION



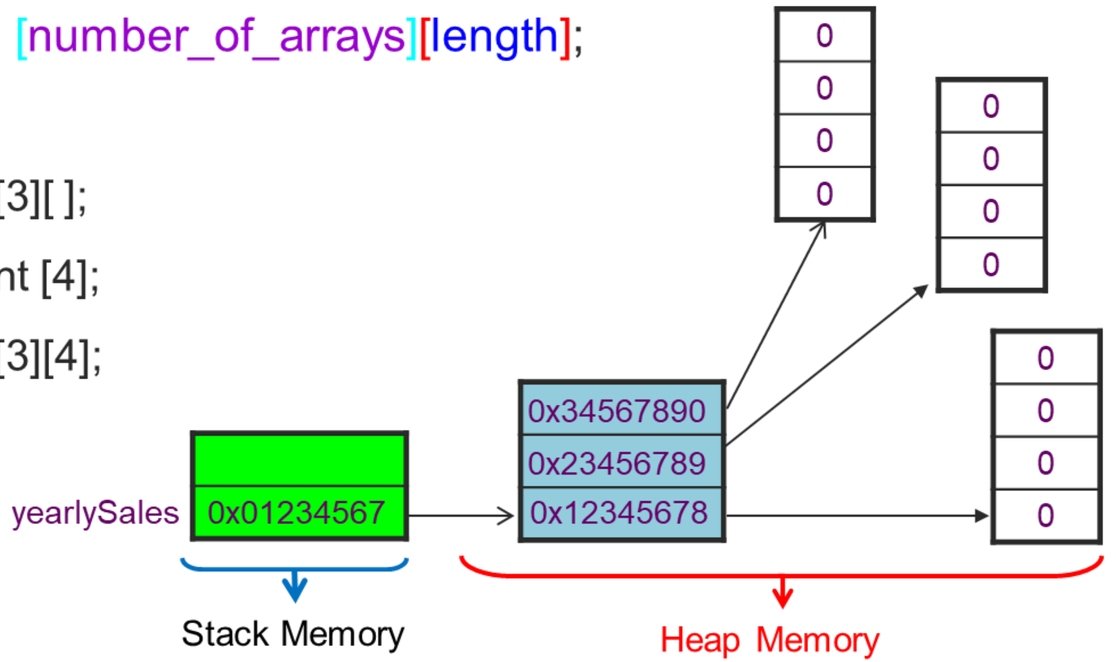
實體化 INSTANTIATED

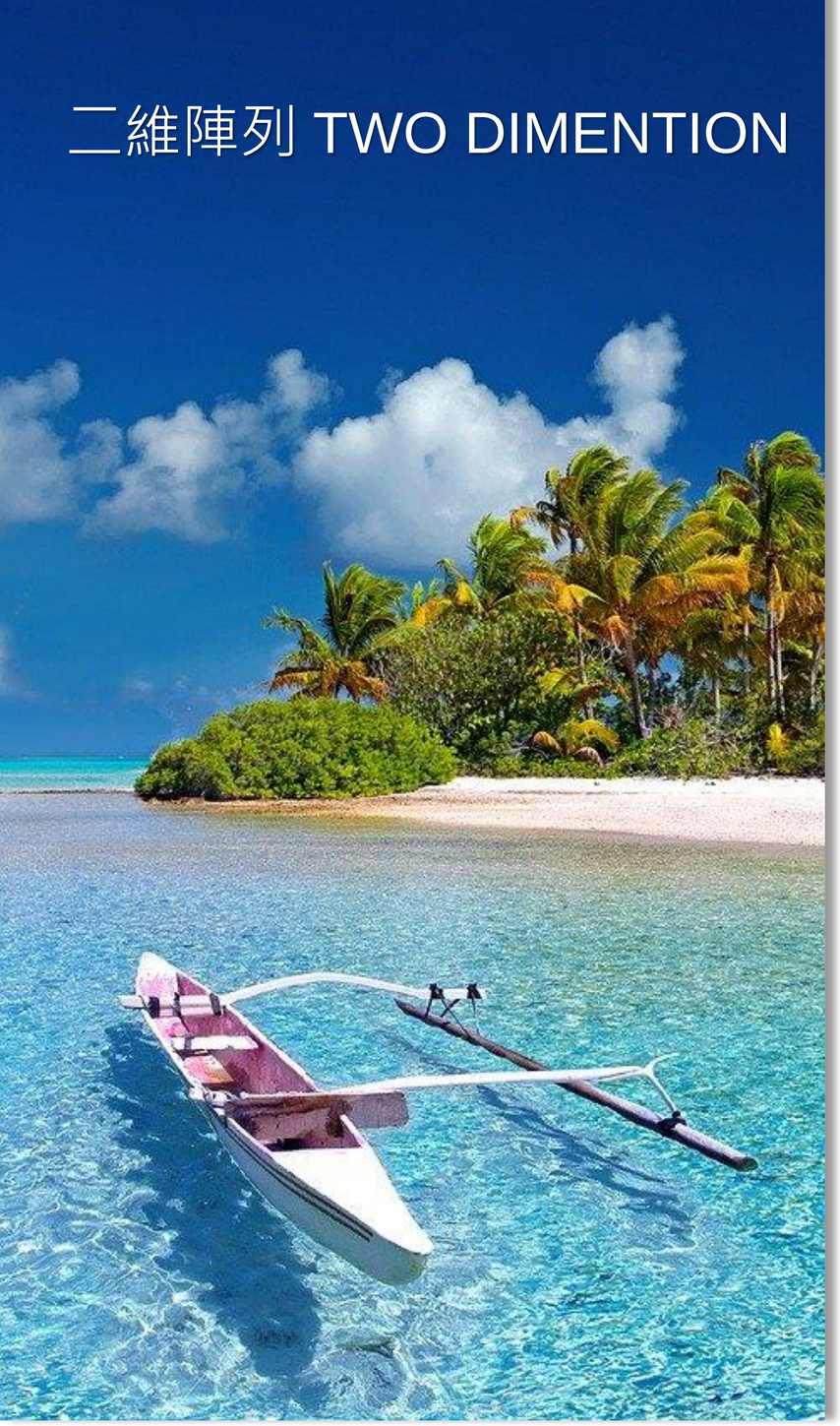
- 語法：

```
array_name = new type [number_of_arrays][length];
```

Ex:

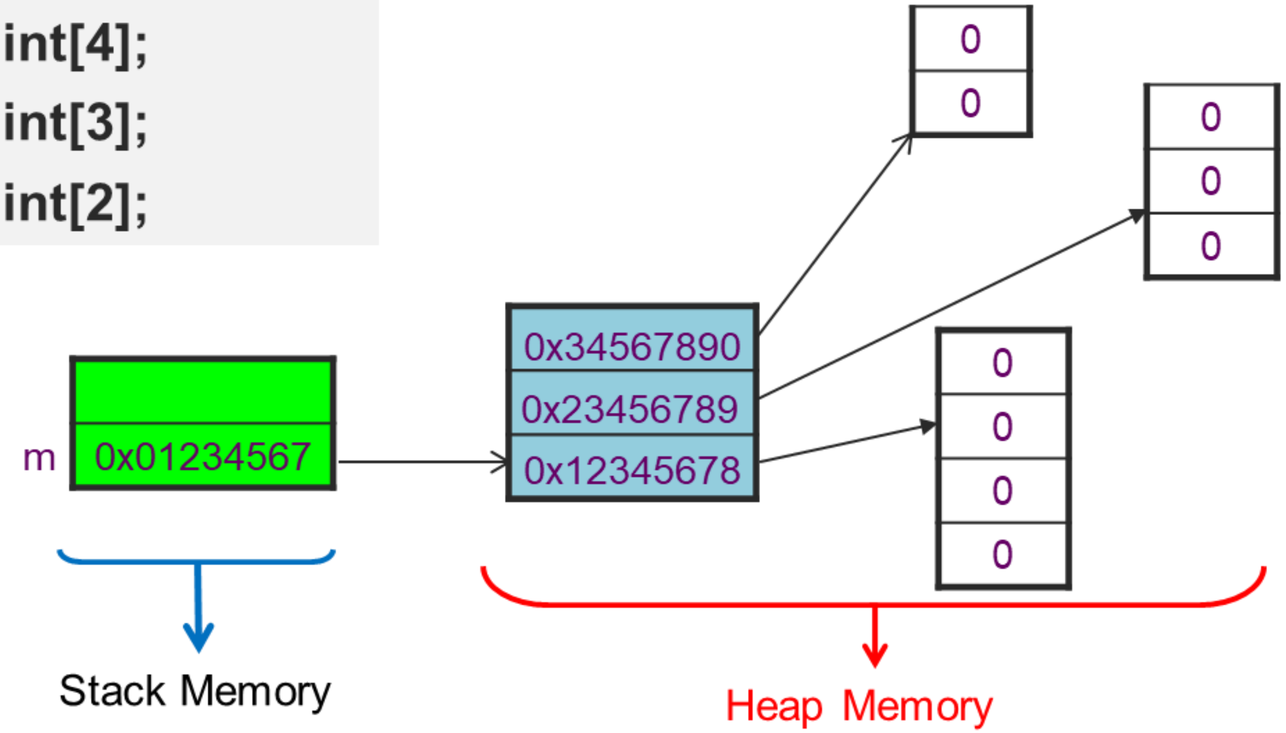
```
yearlySales = new int [3][ ];
yearlySales[0] = new int [4];
yearlySales = new int [3][4];
```



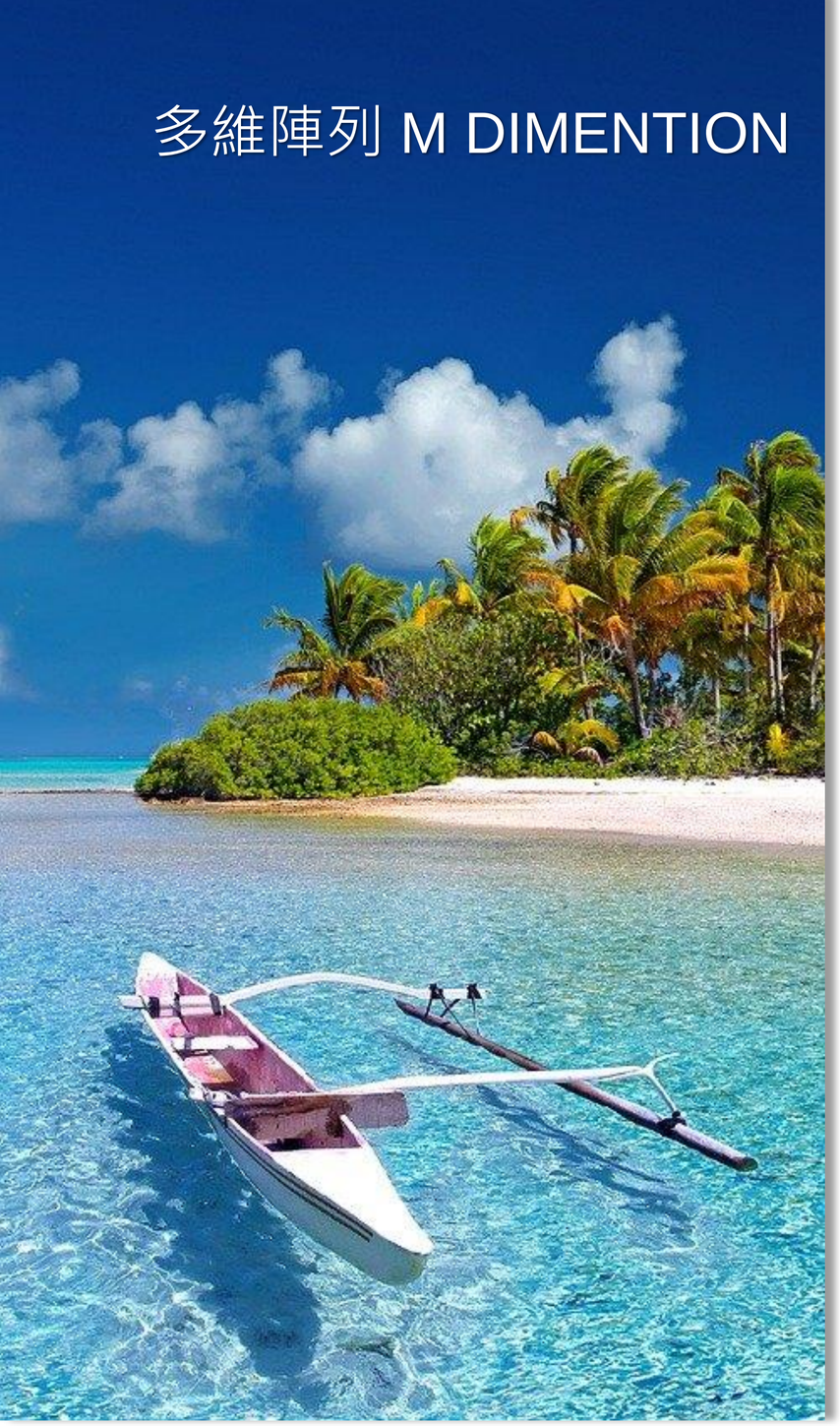


非對稱型陣列 Non-Rectangular

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



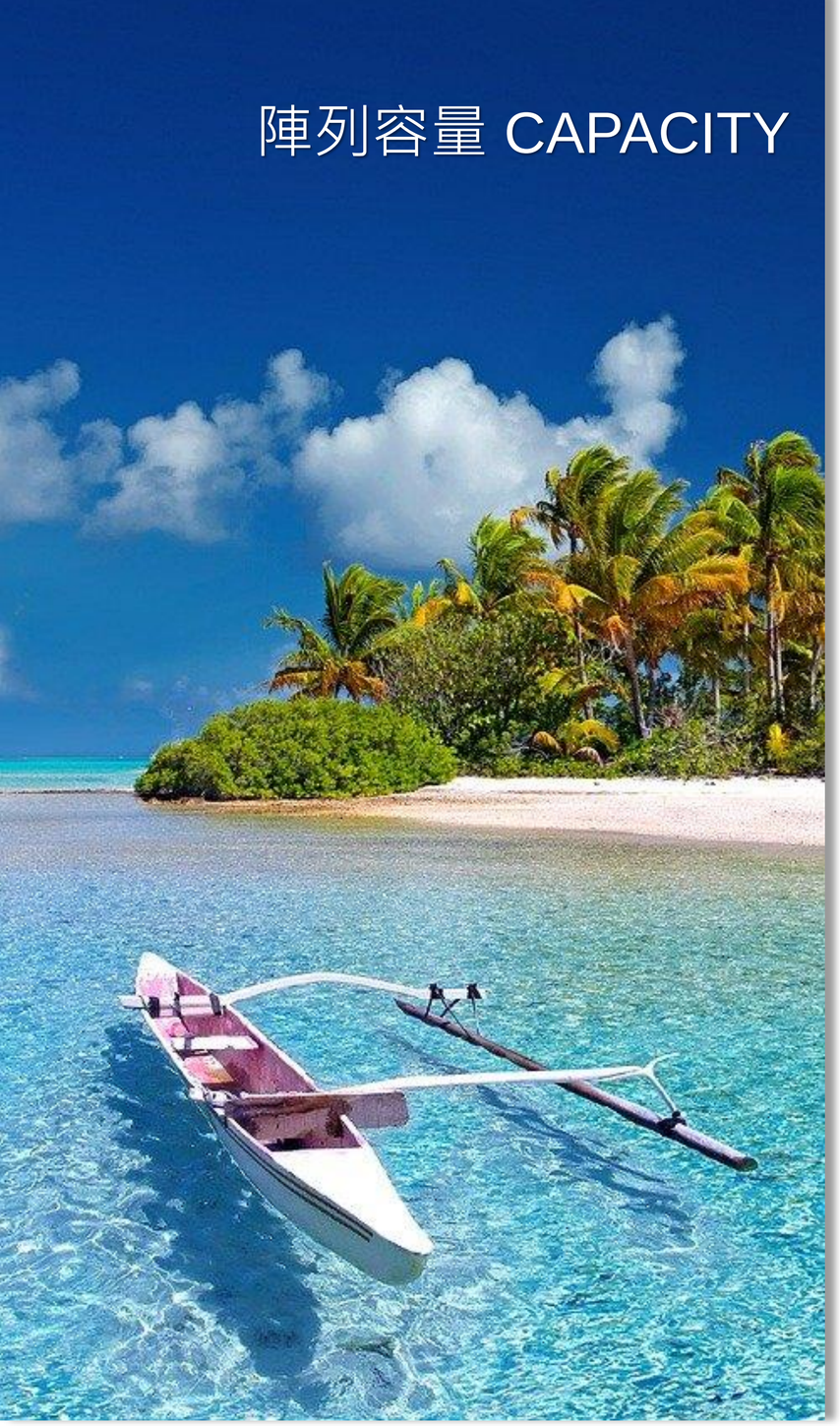
多維陣列 M DIMENTION



- 多維陣列的初始化內容宣告：
  - int m[ ][ ] = { {1,2}, {1,2,3} };
  - 分別以括號（{ }）來表示不同階層的陣列

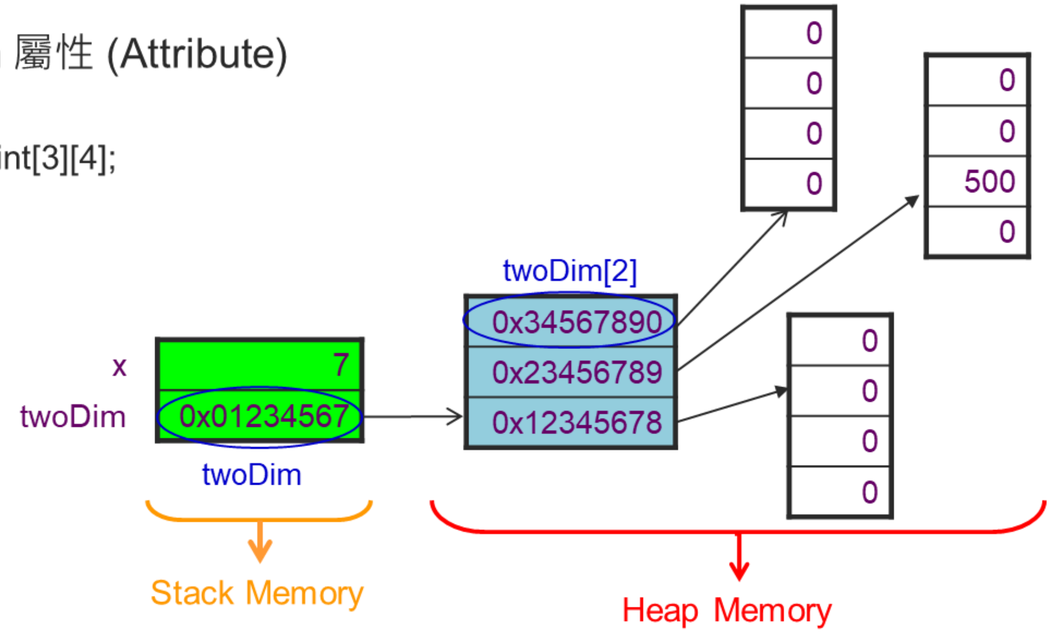
- 陣列的宣告是從左至右的方式來初始陣列大小：
  - int m[ ][ ] = new int[3][ ]; 正確
  - int m[ ][ ] = new int[ ][3];
    此為錯誤的宣告方式，會產生編譯錯誤 (Compile error)

陣列容量 CAPACITY

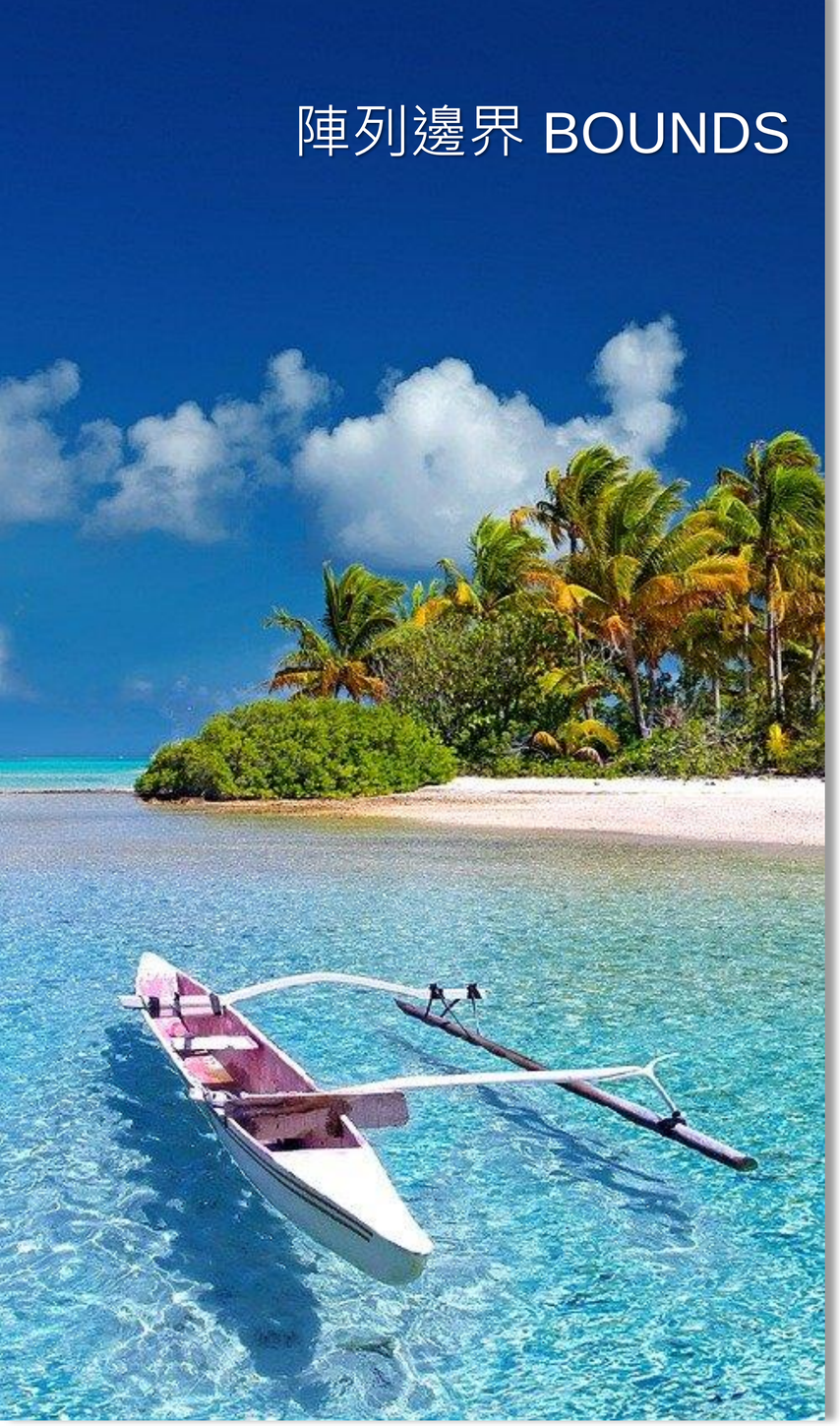


- 取得陣列的容量
  - 陣列物件的 length 屬性 (Attribute)

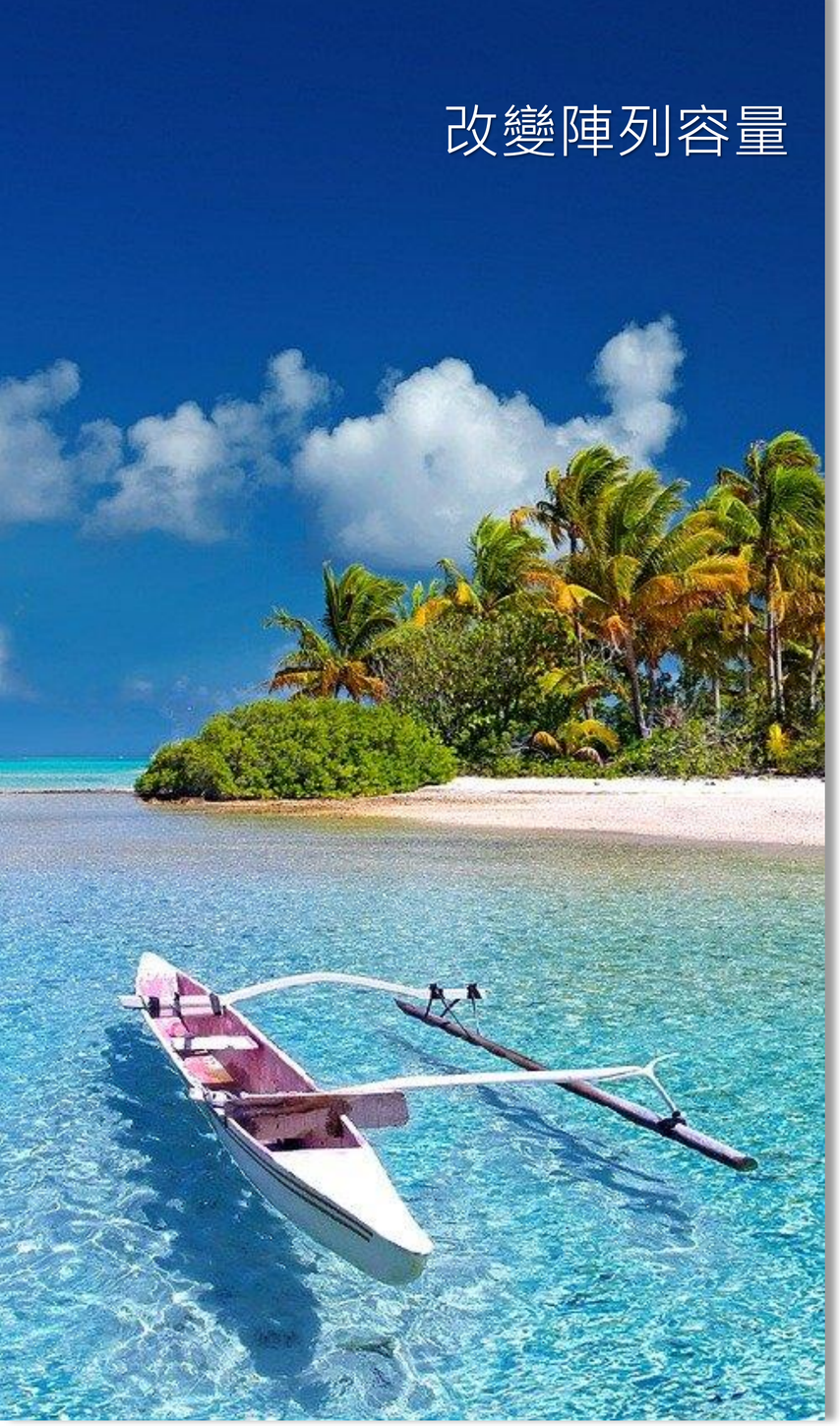
```
int [ ][ ] twoDim = new int[3][4];
twoDim.length;      → 3
twoDim[2].length    → 4
```



陣列邊界 BOUNDS



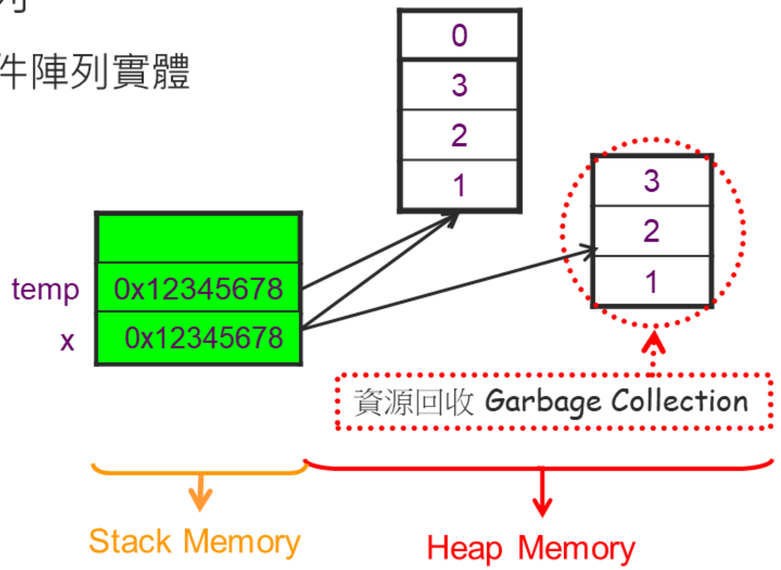
- 陣列索引值
  - 由 0 開始到 length-1
  - 存取陣列的索引值超過邊界範圍，會有執行時期錯誤：ArrayIndexOutOfBoundsException
  - 依序存取陣列元素



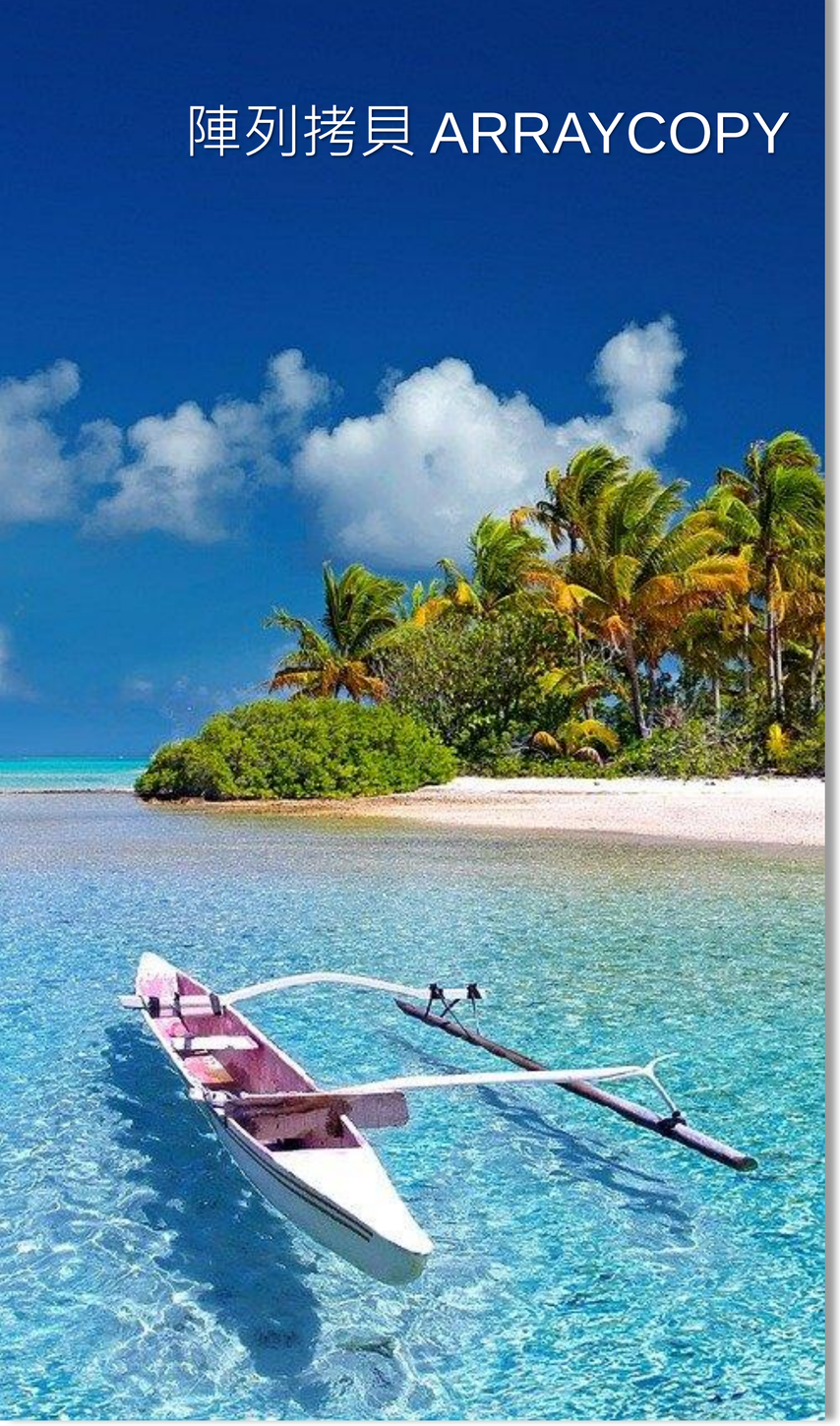
改變陣列容量

- 在 Java 中，陣列不被允許重新定義大小 (Resize)
- 要改變陣列大小必須重新建立陣列
  - 物件參考值會指向新建立出來的物件陣列實體

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



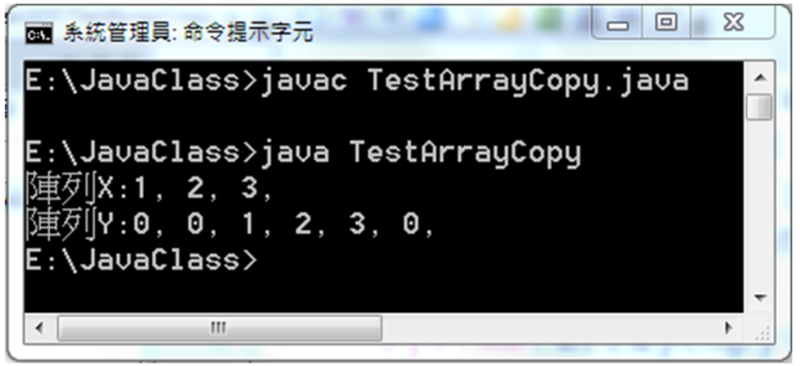
陣列拷貝 ARRAYCOPY



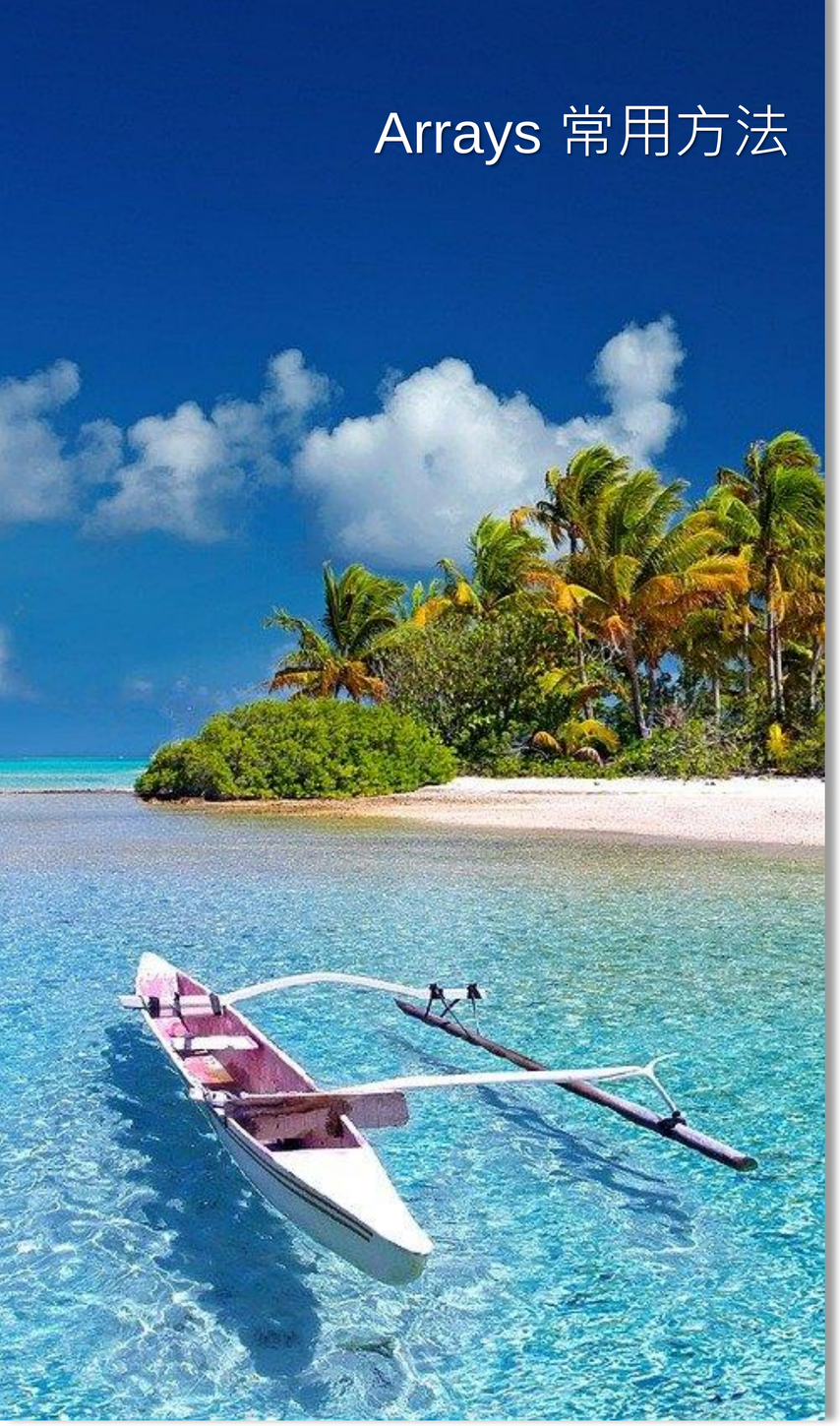
System 類別提供陣列複製的方法

方法名稱	修飾字/傳回值	說明
arraycopy(Object src, int srcPos, Object dest, int destPos, int length)	static void	src:來源陣列, srcPos:複製來源陣列的起始位置 dest:目標陣列, destPos:複製目標陣列的起始位置, length: 複製陣列元素個數

```
01 public class TestArrayCopy {
02     public static void main(String[] args) {
03         int[] x = {1,2,3};
04         int[] y = new int[6];
05         System.arraycopy(x, 0, y, 2, x.length);
06         System.out.print("陣列X:");
07         for(int i=0; i<x.length; i++){
08             System.out.print(x[i]+", ");
09         }
10         System.out.println( );
11         System.out.print("陣列Y:");
12         for(int a : y){
13             System.out.print(a+", ");
14         }
15     }
16 }
17 }
```

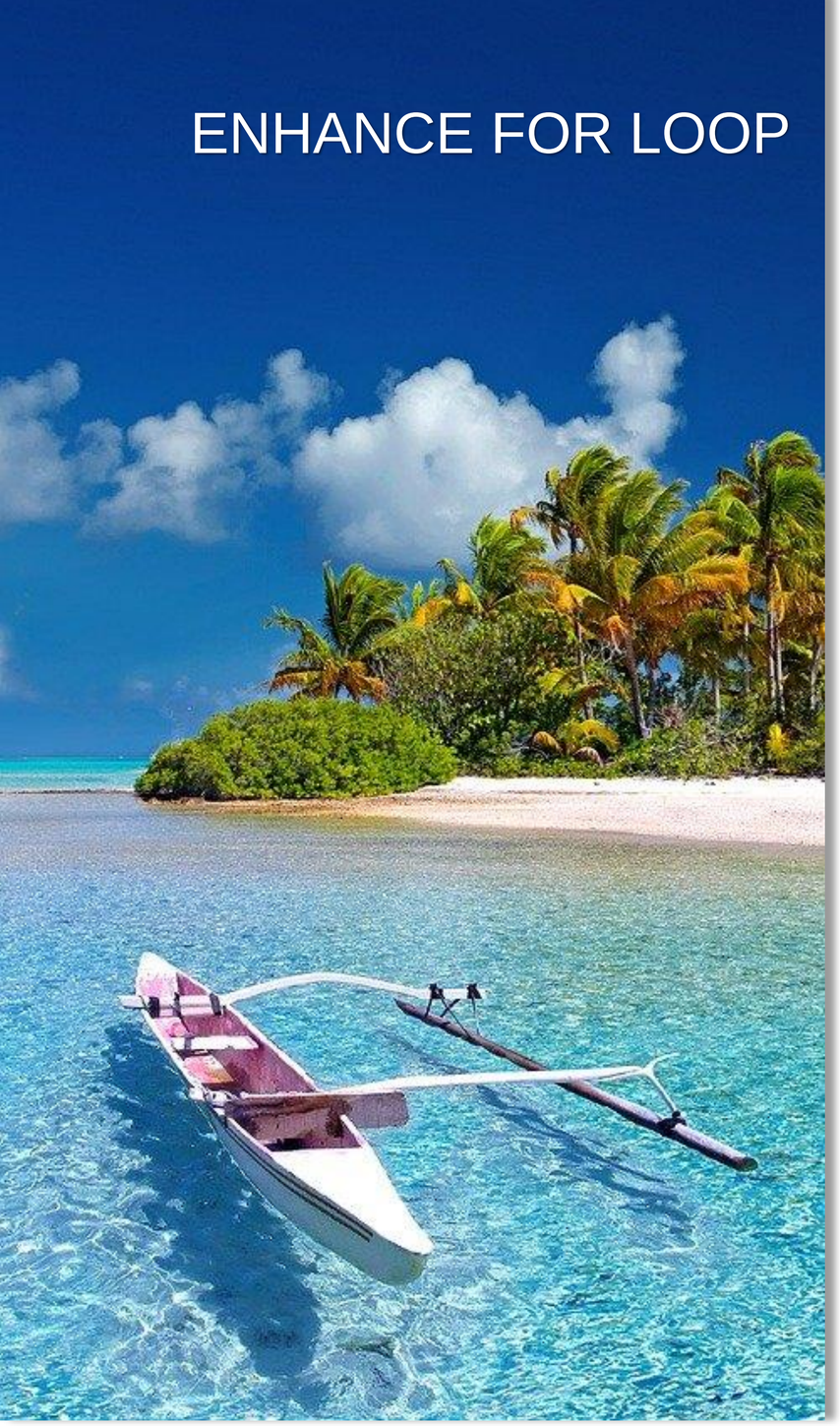


Arrays 常用方法



- toString( )將陣列轉成字串
- asList( )將陣列轉成固定長度的List
- stream( )將陣列轉成對應資料型態之資料流
- compare( )比較二陣列是否相同回傳 0 -1 1
- mismatch( )比較二陣列是否不同,回傳第一個不同之索引值
- sort( )陣列之排序

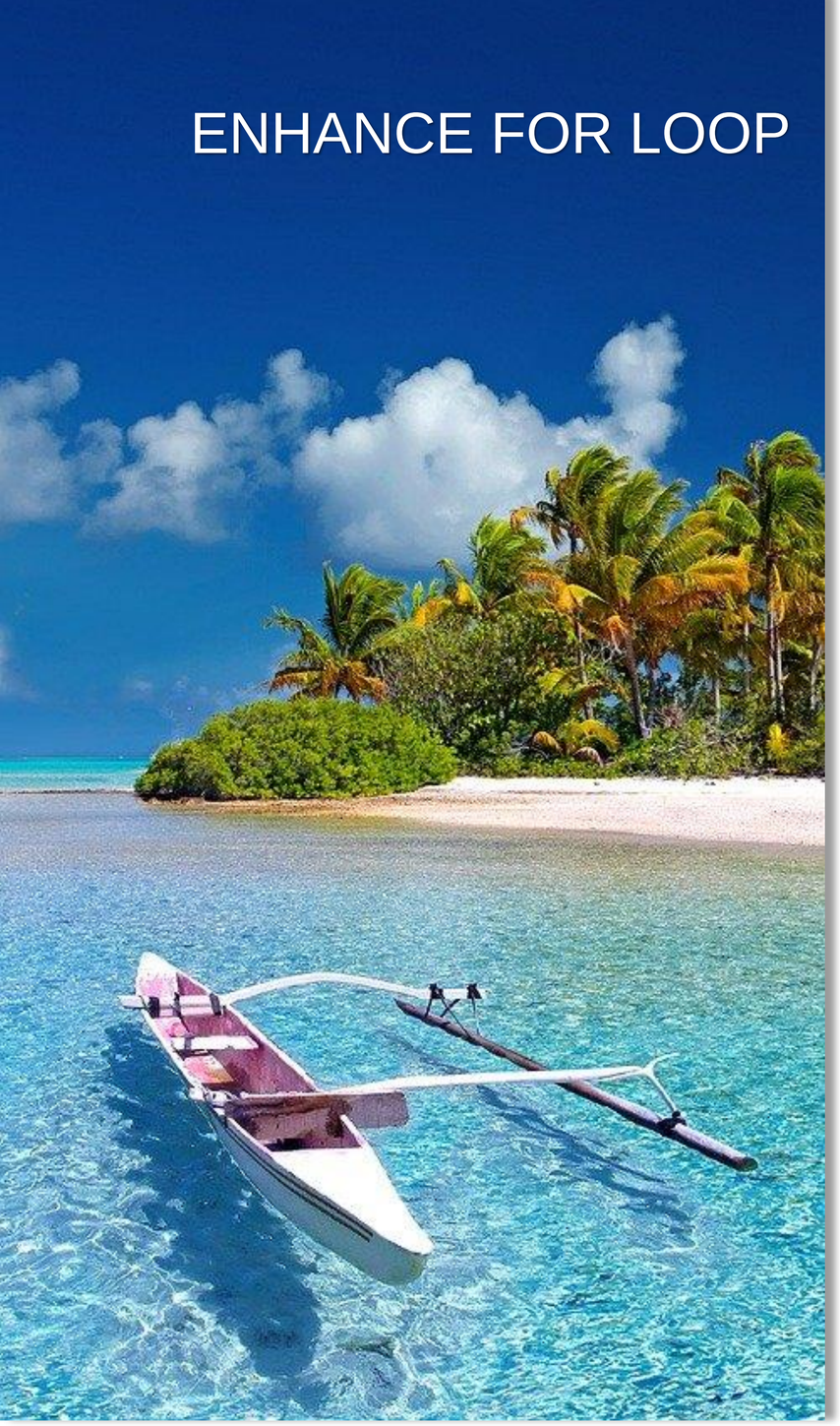
ENHANCE FOR LOOP



- 加強版 for 迴圈 (for each)
  - JavaSE 5.0 所新增的語法。
  - 簡化存取集合性物件的元素內容。
    - 集合性物件：陣列 (array) 或集合(collection)
    - 執行時自動找下一個元素，直到全部擷取完畢為
    - 由 for-loop 實作出來的
  - 讀取集合元素時不可修改元素內容
- for-each 語法

  for (**元素資料型別** **區域變數名稱** ： **元素集合名稱**)

ENHANCE FOR LOOP



- for-loop讀取

```
01 String[] exams = {"SCJP", "SCWCD", "SCMAD"};
02 for(int i=0;i<exams.length;i++) {
03     System.out.print(exams[i] + ", ");
04 }
```

執行結果：**SCJP, SCWCD, SCMAD,**

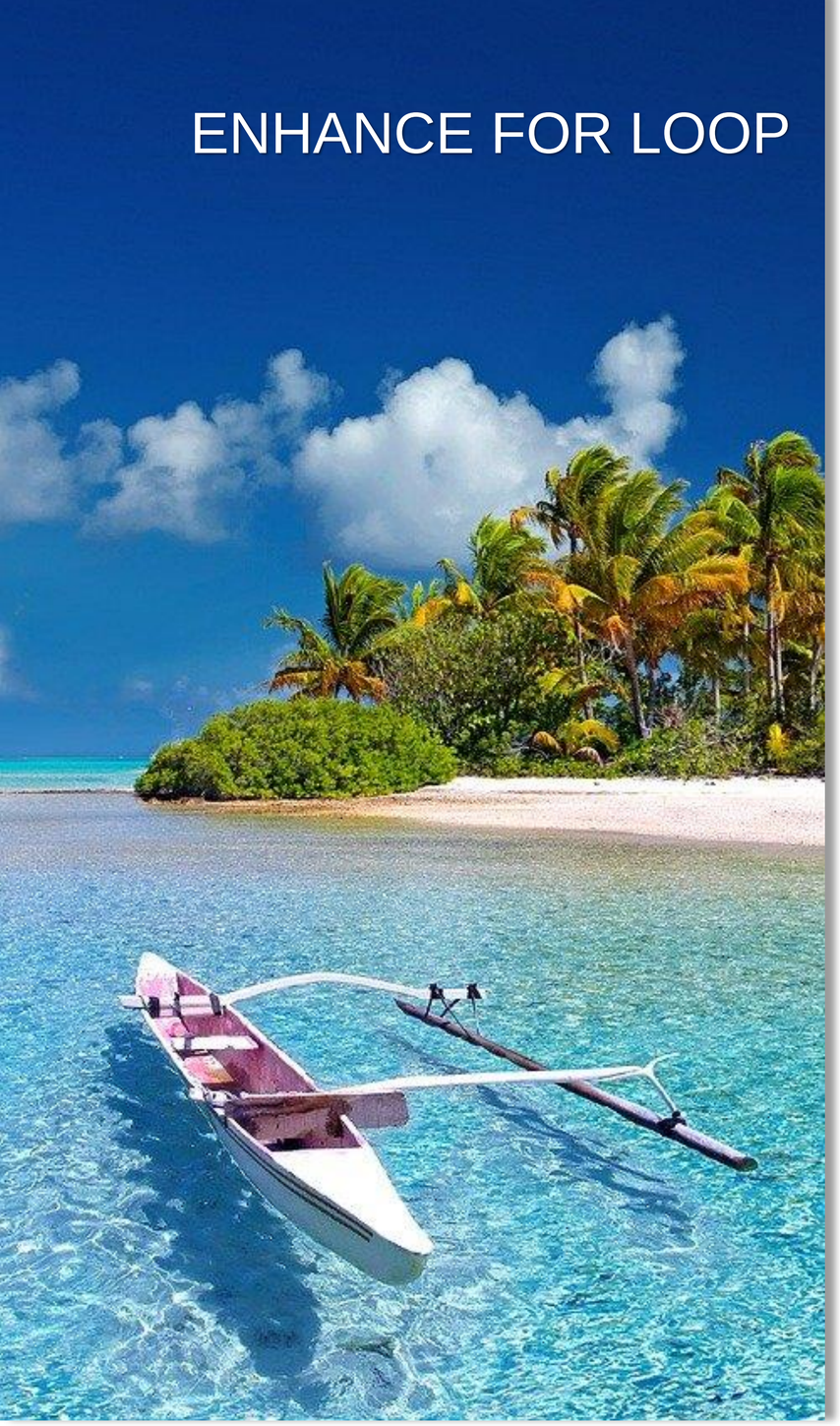
- For-each讀取

```
01 String[] exams = {"SCJP", "SCWCD", "SCMAD"};
02 for(String ex : exams) {
03     System.out.print(ex + ", ");
04 }
```

元素資料型別　區域變數名稱　元素集合名稱

執行結果：**SCJP, SCWCD, SCMAD,**

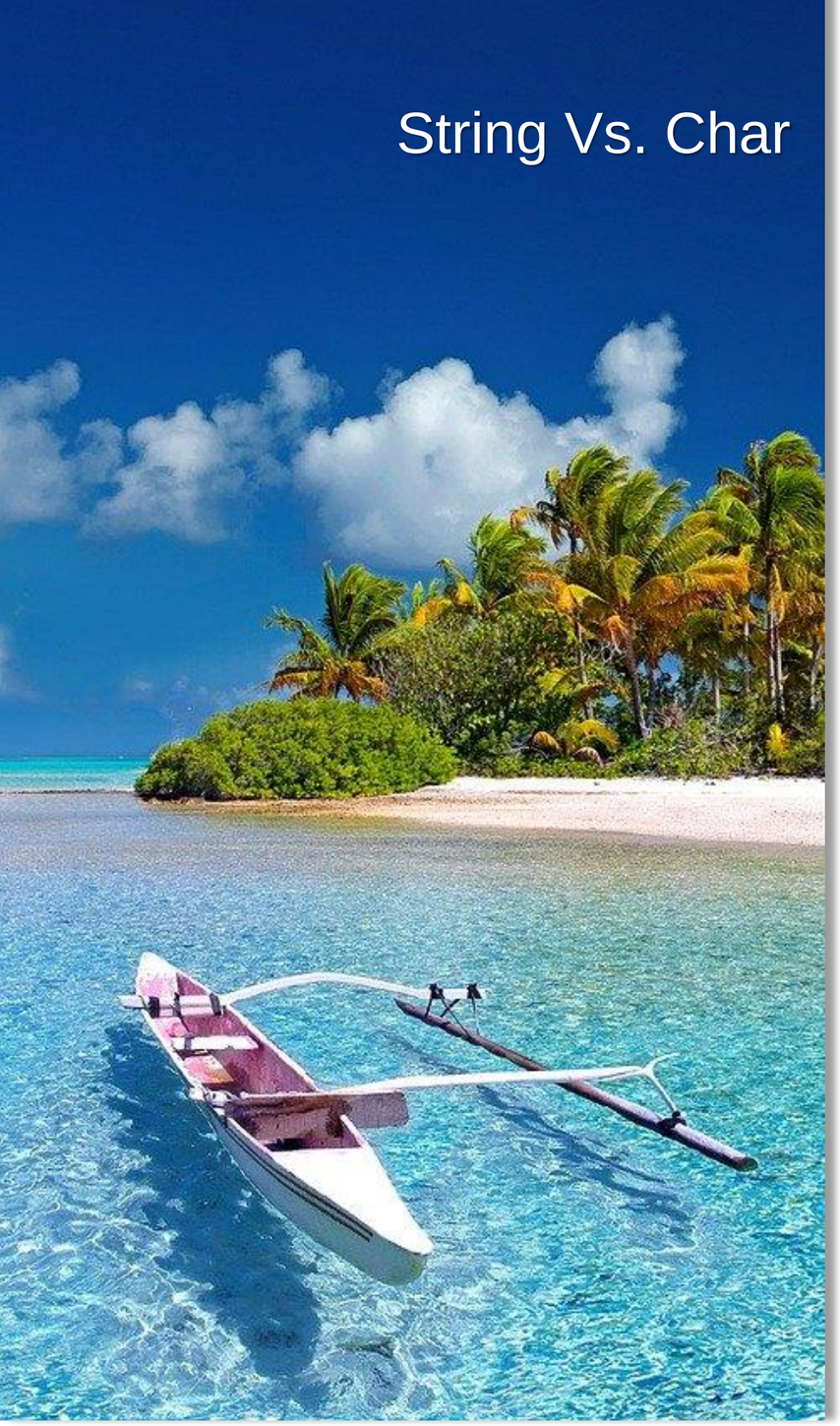
ENHANCE FOR LOOP



```
01 String[][] exams = {
02                     {"SCJP", "SCWCD", "SCMAD"},
03                      {"MCSD", "MCAD", "MCDBA"}
04                    };
05 for(String[] ex : exams) {
06     for(String e : ex) {
07         System.out.print(e + ", ");
08     }
09     System.out.println( );
10 }
```

執行結果：`SCJP, SCWCD, SCMAD,`
`MCSD, MCAD, MCDBA,`

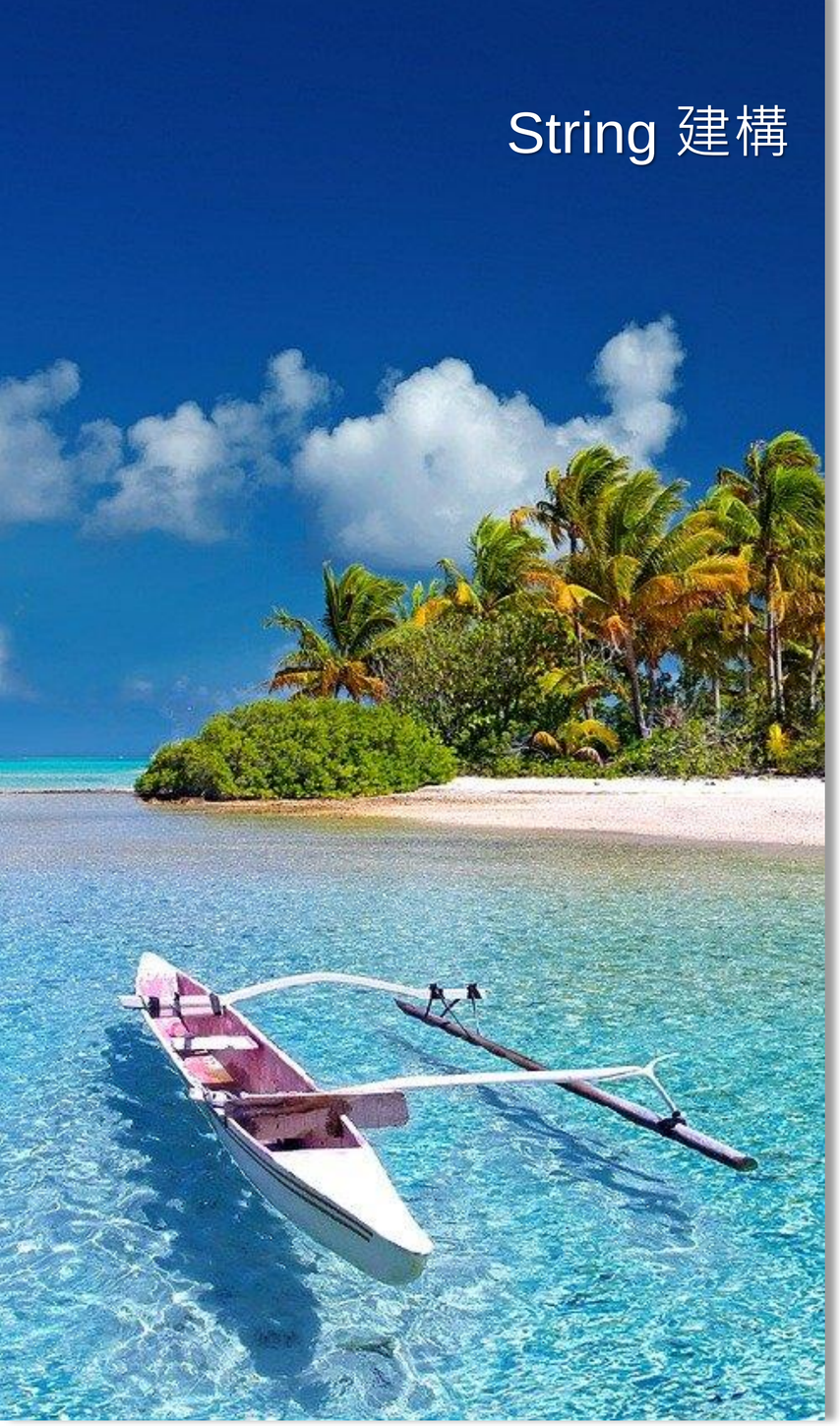
字串 String



String Vs. Char

比較項目	String	char
資料型別(Data type)	參考資料型別 Reference tpye	基本資料型別 Primitive type
Literal 資料的符號 Literal enclosed	使用雙引號（ " ）	使用單引號（ ' ）
抽象表現 Represent	類別（class）	Unicode
比較運算 Operator Compared	equals( ) 或 (==)	雙等號（ == ）
記憶體內容值是否可變？	不可變	可變

String 建構



建立字串物件

標準物件建構語法

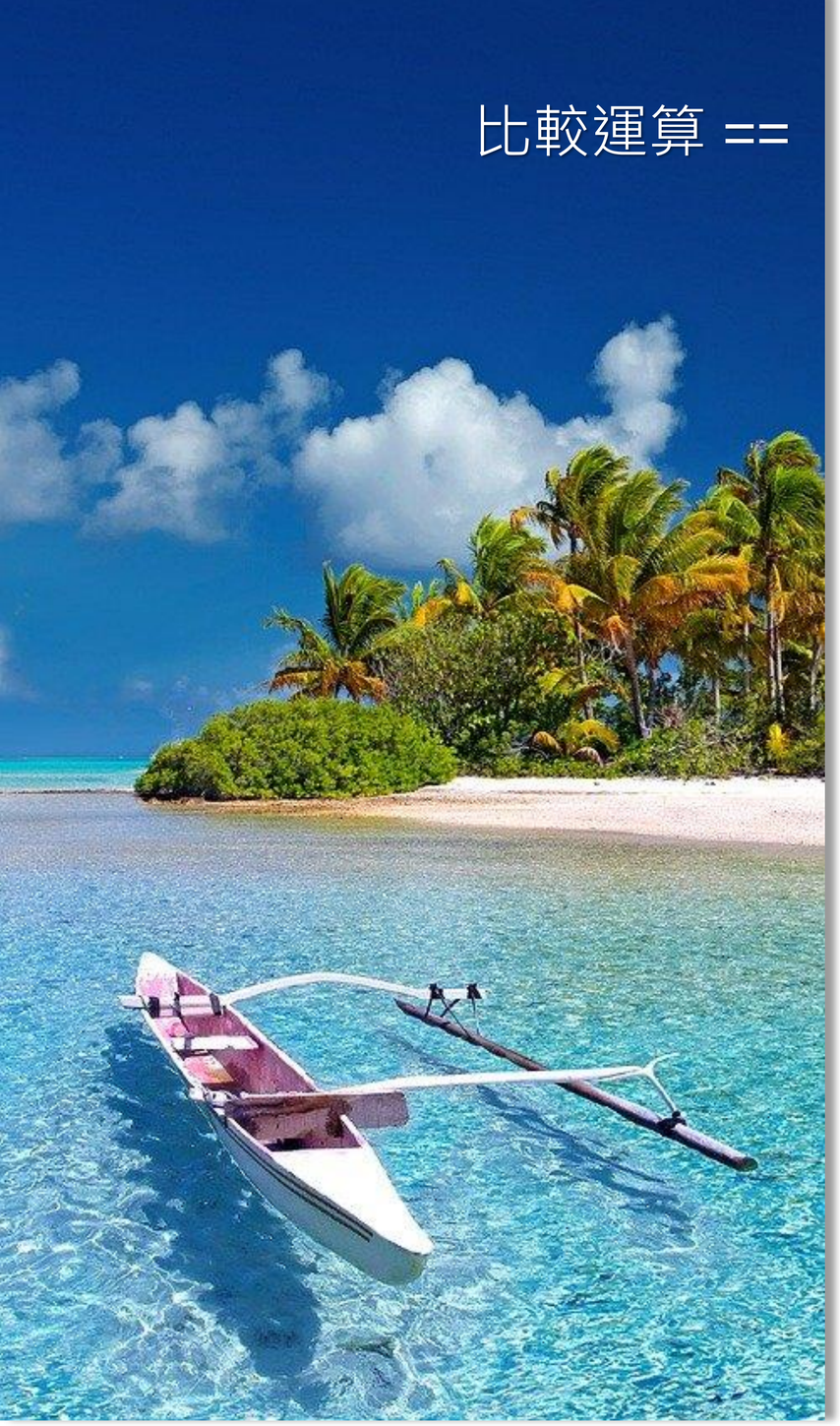
String str1 = new String(“Hello World!”); //一般物件建構

建構子	說明
String( )	建立一個空字元序列的 String 物件
String(byte[] bytes)	使用指定的 byte 陣列內容，建構一個新的 String
String(char[] value)	使用指定的 char陣列內容，建構一個新的 String
String(String original)	建立一個與字串參數相同的新 String 物件
String(StringBuffer buffer)	建立一個與字元緩衝參數相同的新 String 物件
String(StringBuilder builder)	建立一個與字元緩衝參數相同的新 String 物件

非標準物件建構語法

String str2 = “Hello World!”; //literal方式賦值

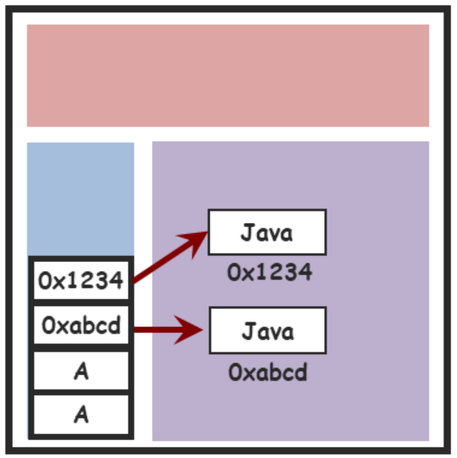
- 字串池重複使用

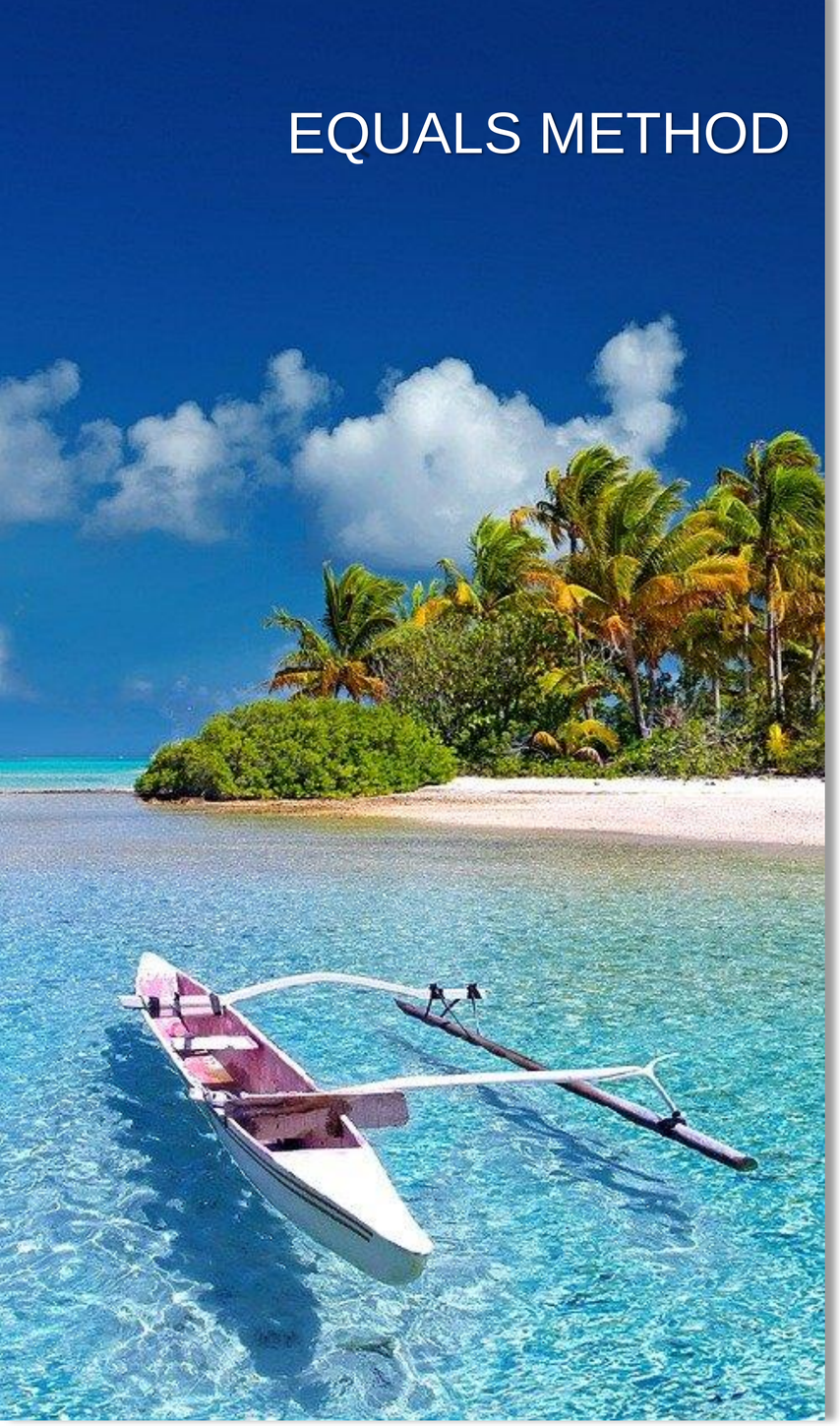


比較運算 ==

- "=="
  - 比較在 stack 中的變數內容是否相等
    - 基本資料型別：變數內容即為其值
    - 參考資料型別：物件是不是同一個

```
char c1 = 'A';
char c2 = 'A';
c1 == c2 ?    true !

String s1 = new String("Java");
String s2 = new String("Java");
s1 == s2 ?   0xabcd == 0x1234 ?  false !
```

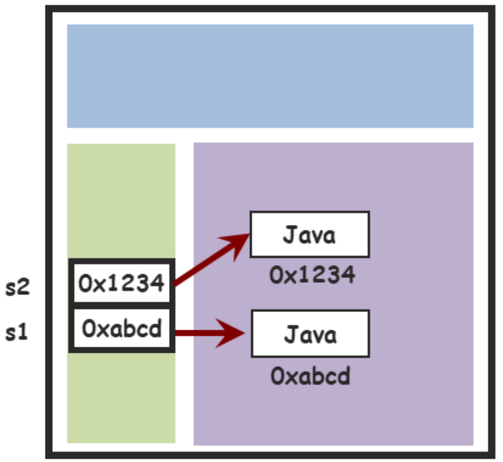




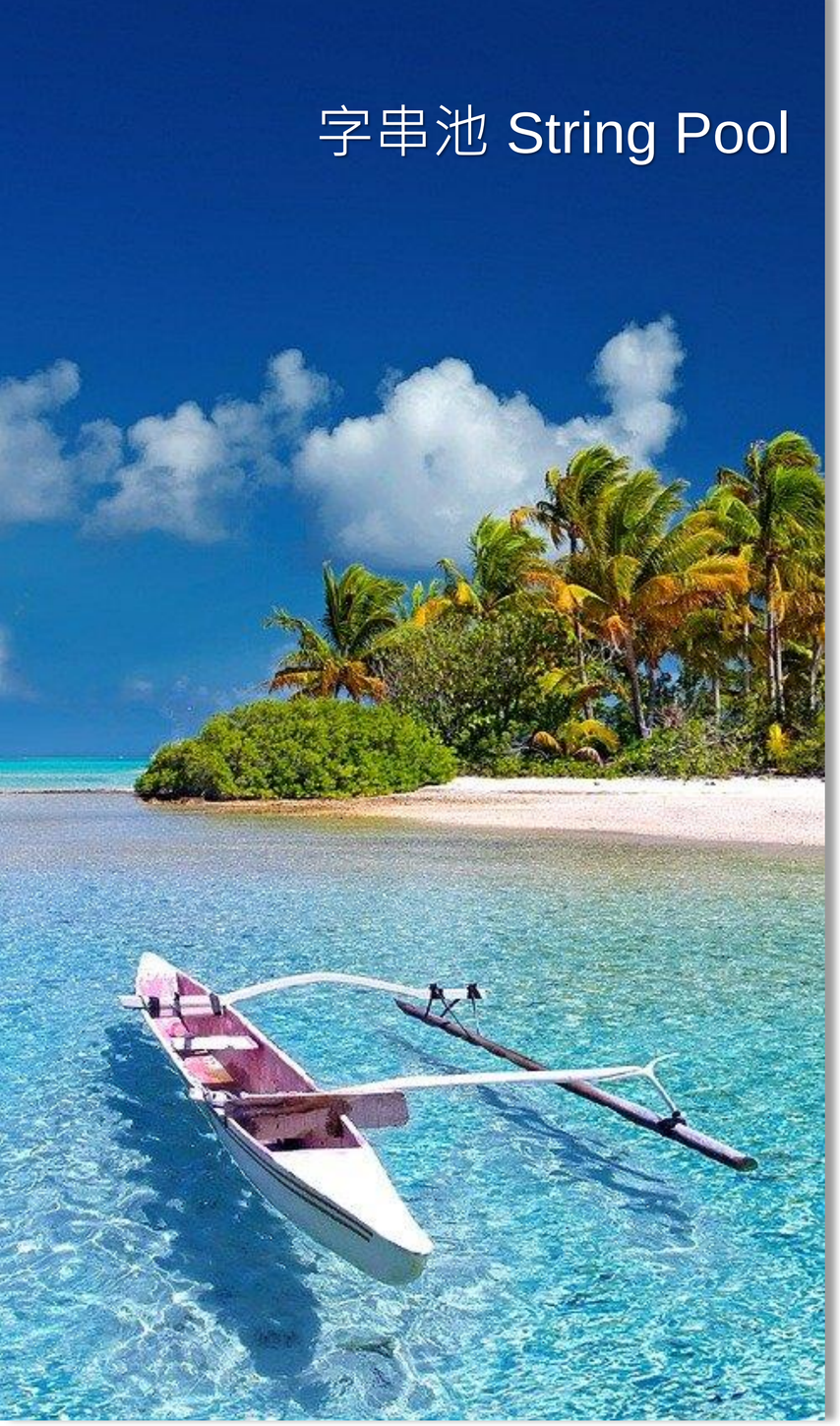
EQUALS METHOD

- equals( ) 方法
  - String 所提供的 equals( ) 來比較字串的實際內容

```
String s1 = new String("Java");
String s2 = new String("Java");

s1.equals( s2 ) ?  "Java" = "Java"    true !
```

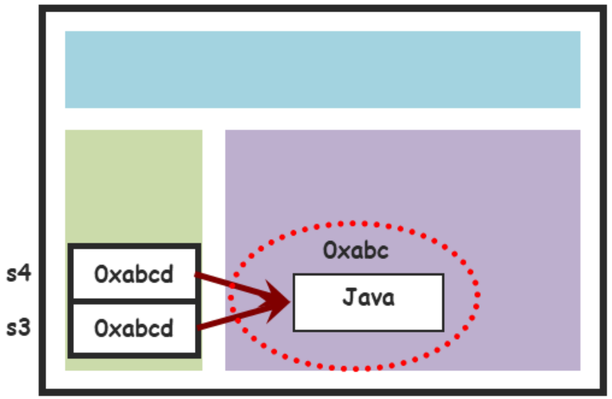


字串池 String Pool

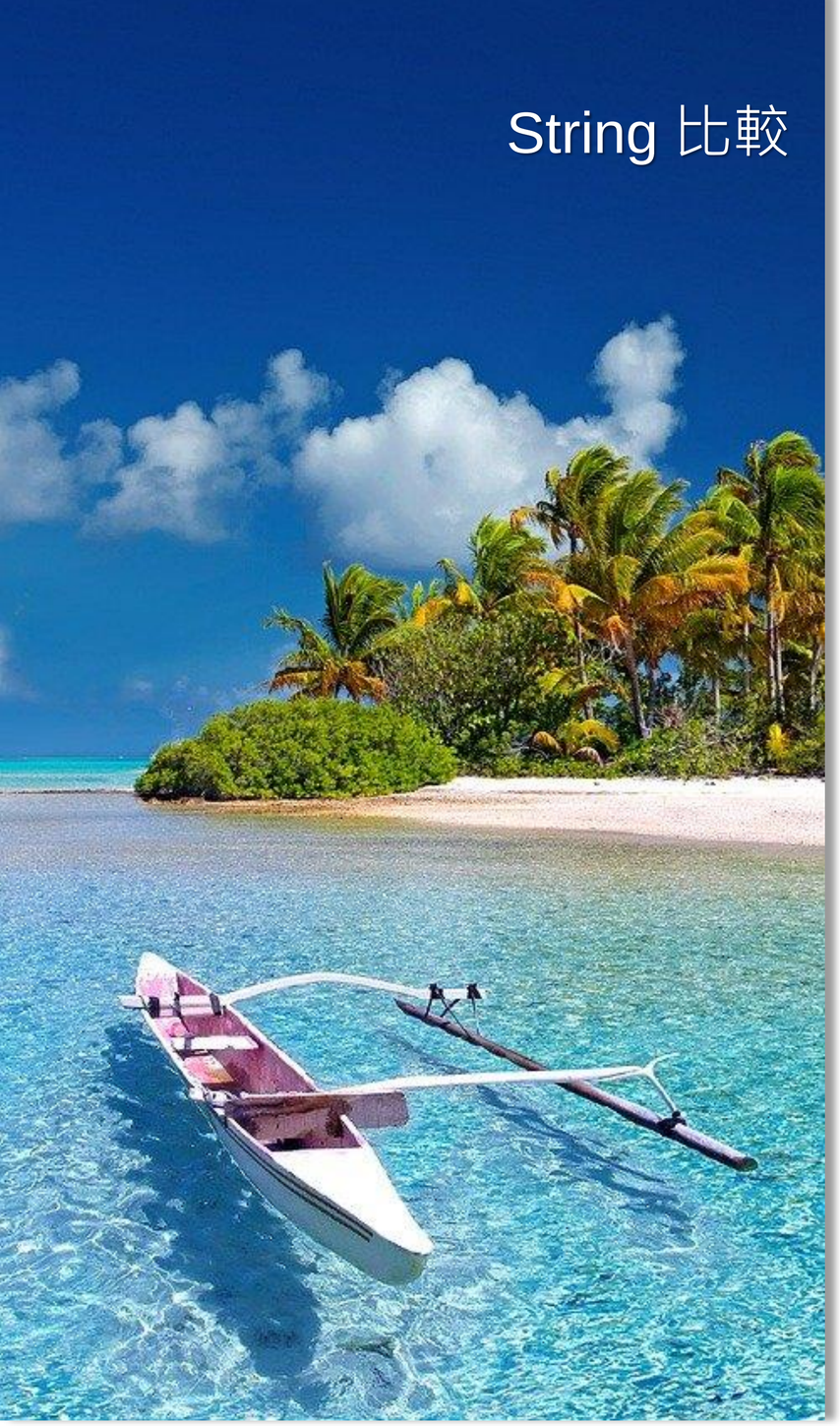


- JVM 中存放字串的共用區域
  - 以非標準物件建構語法方式，宣告的字串物件儲存到字串池中
  - 建立物件前檢查是否已有相同的物件
    - 如果有的話直接傳回參考值，不會新增字串物件

```
String s3 = “Java”;
String s4 = “Java”;

s3 == s4 ?  0xabcd == 0xabcd ?  true !
```



String 比較



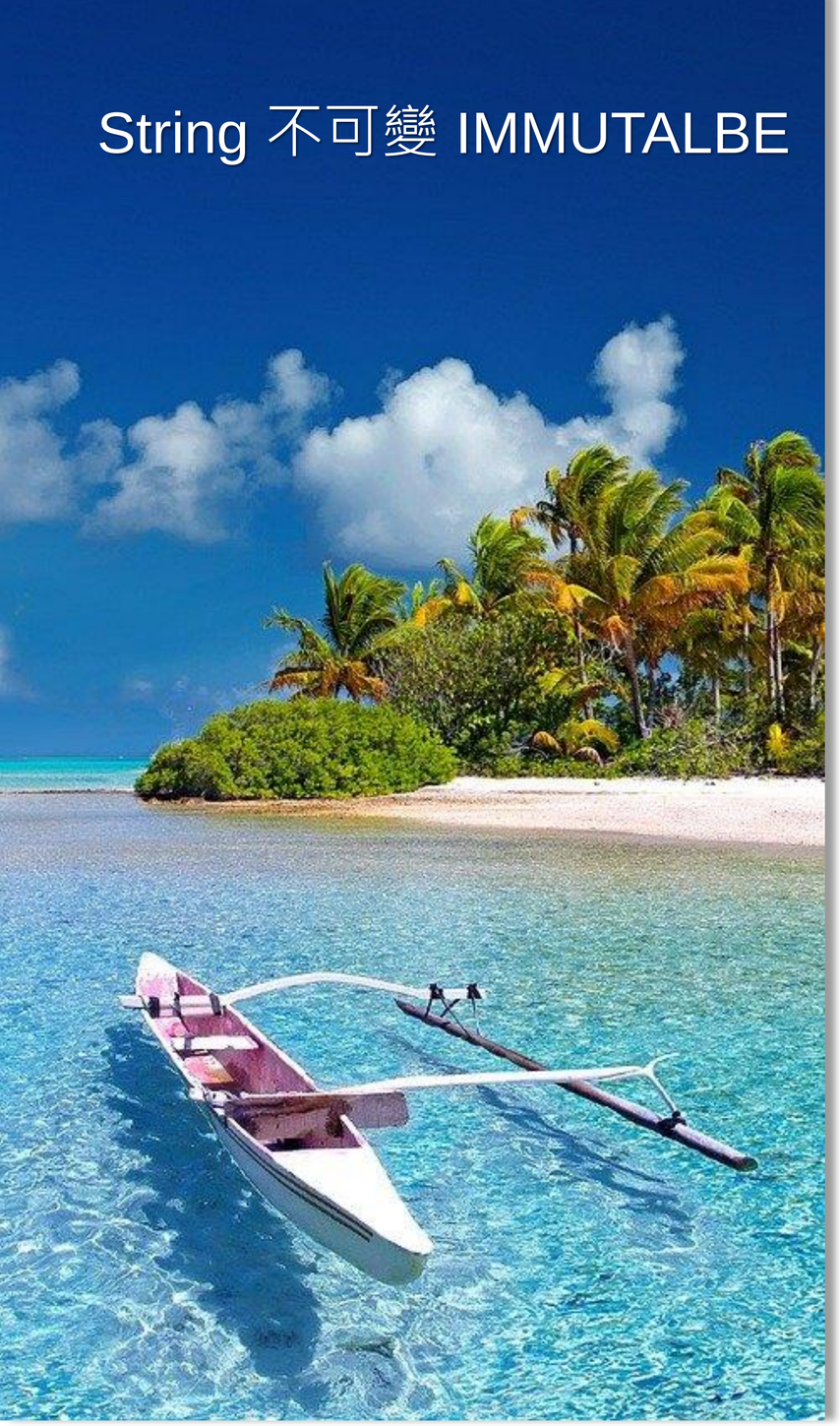
```
01  public class TestString {
02      public static void main(String[] args) {
03          String s1 = new String("Java");
04          String s2 = new String("Java");
05          String s3 = "Java";
06          String s4 = "Java";
07          System.out.println(s1 == s2);
08          System.out.println(s3 == s4);
09          System.out.println(s1.equals(s2));
10          System.out.println(s3.equals(s4));
11      }
12  }
```

➔ `false`

➔ `true // String pool`

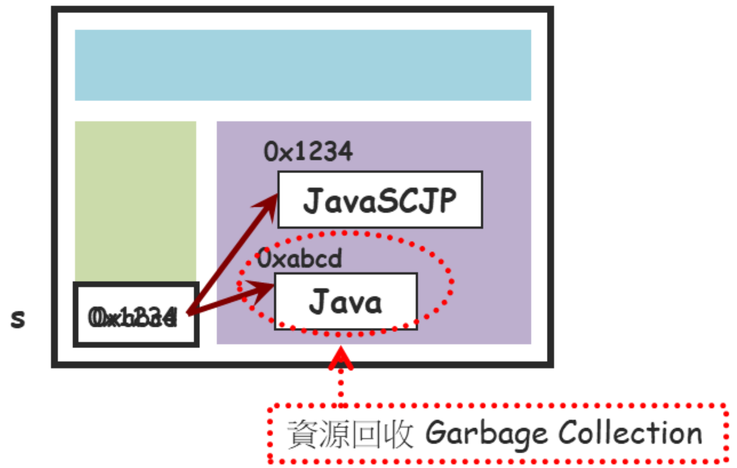
➔ `true // 比較實際內容`

String 不可變 IMMUTALBE

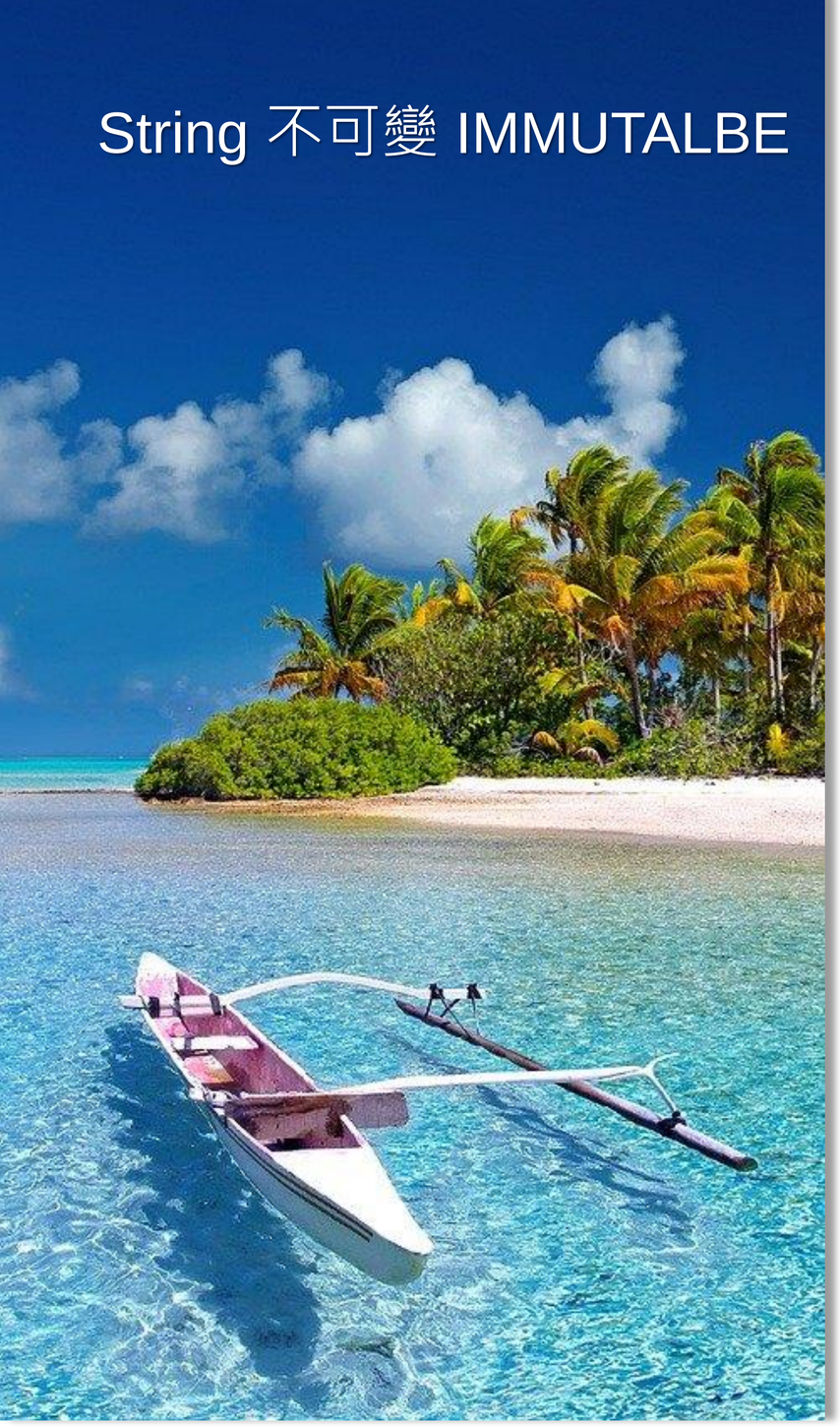


- 字串內容是不可變更的 (immutable)
  - 當字串已在記憶體中被建立出來，該記憶體中的字串內容就不允許被變更。

```
String s = “Java”;
s = s + “SCJP”;
System.out.println(s); ➔ JavaSCJP
```

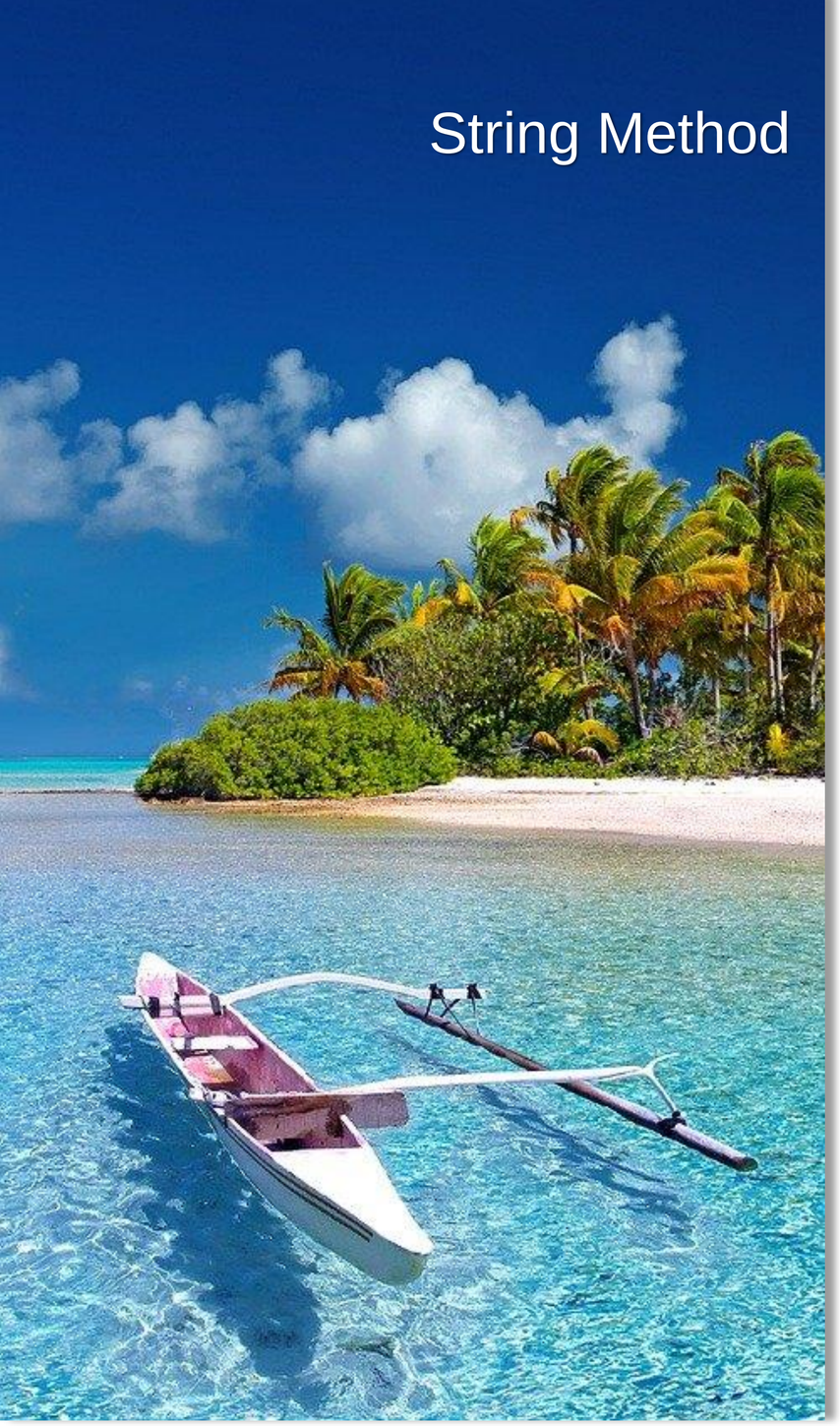


String 不可變 IMMUTALBE



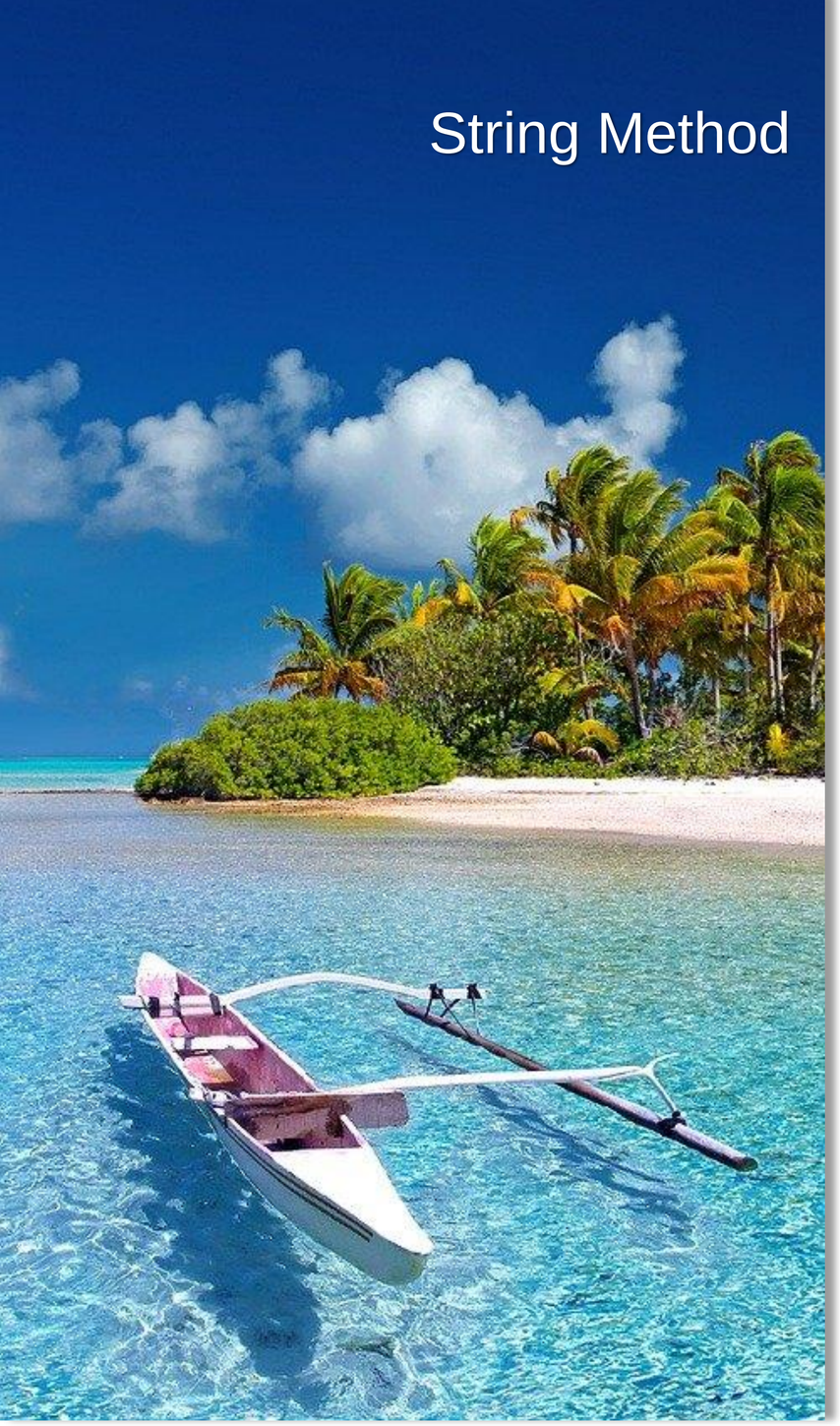
```
01  public class TestImmutable {
02      public static void main(String[] args) {
03          String s1 = “Java”;
04          s1 = s1.concat(“SCJP”);
05          System.out.println(s1);      → JavaSCJP
06          String s2 = “Java”;
07          s2.concat(“SCJP”);
08          System.out.println(s2);      → Java
09          String s3 = “Java”;
10          s3 = s3.replace(‘J’, ‘I’);
11          System.out.println(s3);      → Iava
12          String s4 = “Java”;
13          s4.replace(‘J’, ‘I’);
14          System.out.println(s4);      → Java
15      }
16  }
```

String Method



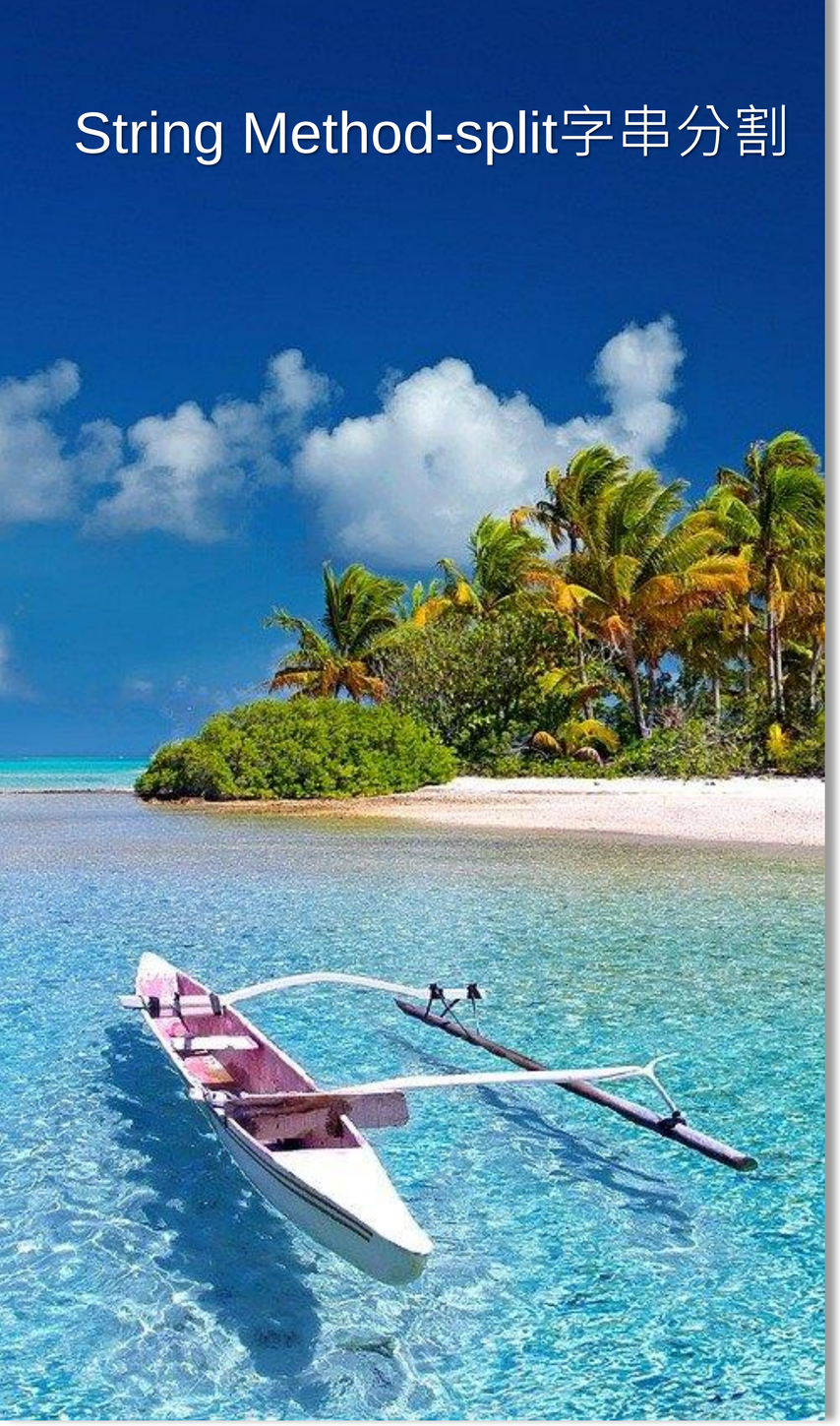
方法名稱	傳回值	說明
charAt(int index)	char	返回指定索引處的 char 值。
equals(Object anObject)	boolean	將此字元串與指定的物件比較。
equalsIgnoreCase(String anotherString)	boolean	將此 String 與另一個 String 比較，不考慮大小寫。
compareTo(String anotherString)	int	按字典順序比較兩個字元串。
compareToIgnoreCase(String str)	int	按字典順序比較兩個字元串，不考慮大小寫。
hashCode( )	int	返回此字元串的雜湊碼。
contains(CharSequence s)	boolean	當當此字元串包含指定的 char 值序列時，返回 true。
startsWith(String prefix)	boolean	測試此字元串是否以指定的前綴開始。
endsWith(String suffix)	boolean	測試此字元串是否以指定的後綴結束。
indexOf(String str)	int	返回指定子字元串在此字元串中第一次出現處的索引。
lastIndexOf(String str)	int	返回指定字元在此字元串中最後一次出現處的索引。
isEmpty( )	boolean	當 length( ) 為 0 時返回 true。
length( )	int	返回此字元串的長度。

String Method



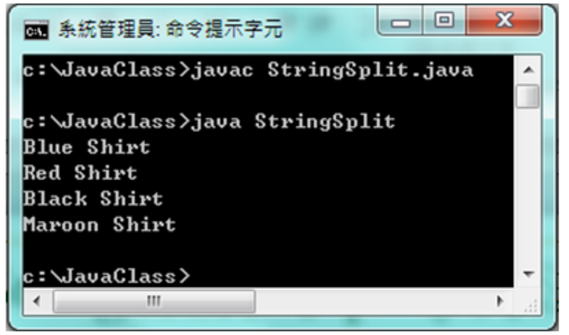
方法名稱	傳回值	說明
concat(String str)	String	將指定字串連接到此字串的結尾。
replace(char oldChar, char newChar)	String	返回一個新的字串，以 newChar 替換此字串中出現的所有 oldChar。
replaceAll(String regex, String replacement)	String	使用給定的 replacement 替換此字元串所有匹配給定的正則表達式的子字元串。
toLowerCase( )	String	使用預設語言環境的規則將此 String 中的所有字元都轉換為小寫。
toUpperCase( )	String	使用預設語言環境的規則將此 String 中的所有字元都轉換為大寫。
trim( )	String	返回字串的副本，忽略前導空白和尾部空白。
substring(int beginIndex)	String	返回一個內容為此字串指定位置開始到結尾位置的子字串。
substring(int beginIndex, int endIndex)	String	返回一個內容為此字串指定位置開始到指定位置結尾的子字串。
split(String regex)	String[]	根據給定正則表達式的匹配拆分此字串。
format(String format, Object... args)	static String	使用指定的格式字元串和參數返回一個格式化字串。

String Method-split字串分割

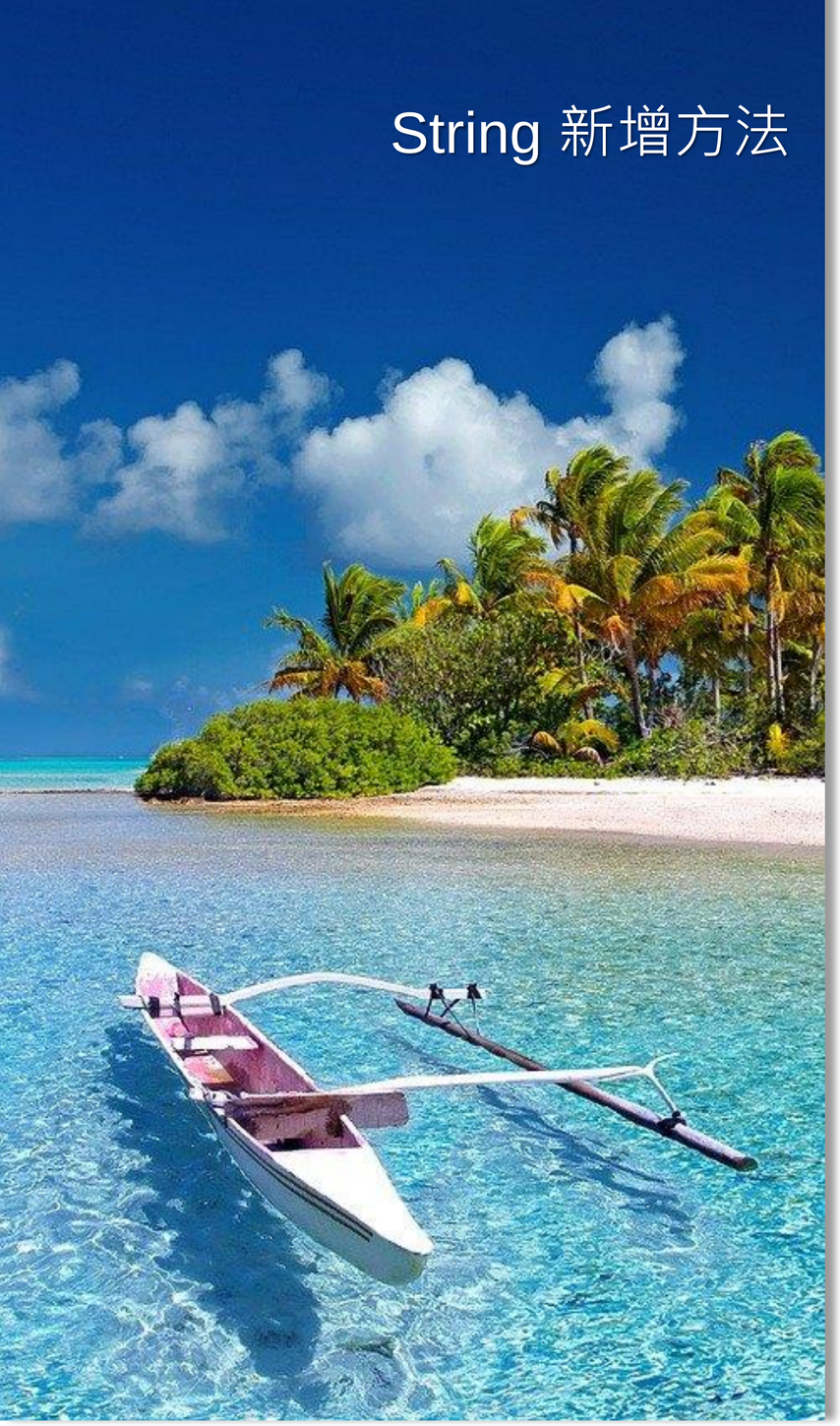


常用方法	傳回值	說明
split(String regex)	String[ ]	以指定正則表示式之比對樣式 regex 來切割字串

```
public class StringSplit {
    public static void main(String[] args) {
        String shirts = "Blue Shirt, Red Shirt,
                         Black Shirt, Maroon Shirt";

        String[ ] results = shirts.split(", ");
        for(int i=0; i<4; i++){
            System.out.println(results[i]);
        }
    }
}
```

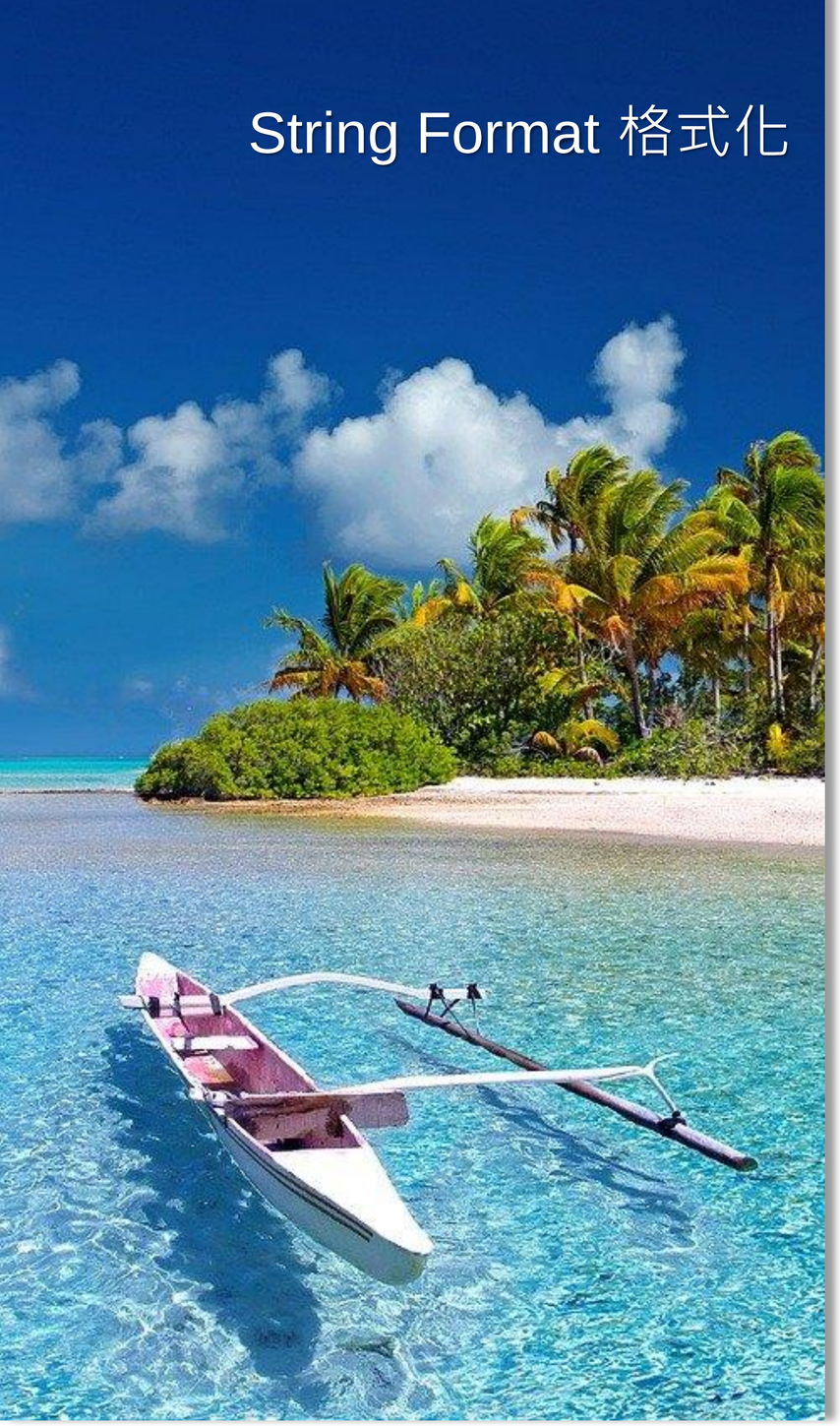


String 新增方法



- strip( )刪除前後空白
- stripTrailing( )刪除後空白
- stripLeading( )刪除前空白
- repeat(n)重覆幾次
- lines( ).count( )計算共有幾行
- blank( ) 判斷是否為空白(此為內容是否為空白,空白字元,TAB相同),不同於isEmpty(此為空字串"")
- intern( )會去字串池內找有無相同字串,若有則回傳該物件的地址,若沒有則在字串池產生一新字串,並返回該字串的地址

String Format 格式化



String 類別 format( )方法

- Java 5.0 新增，程式可簡易的將輸出的資料格式化

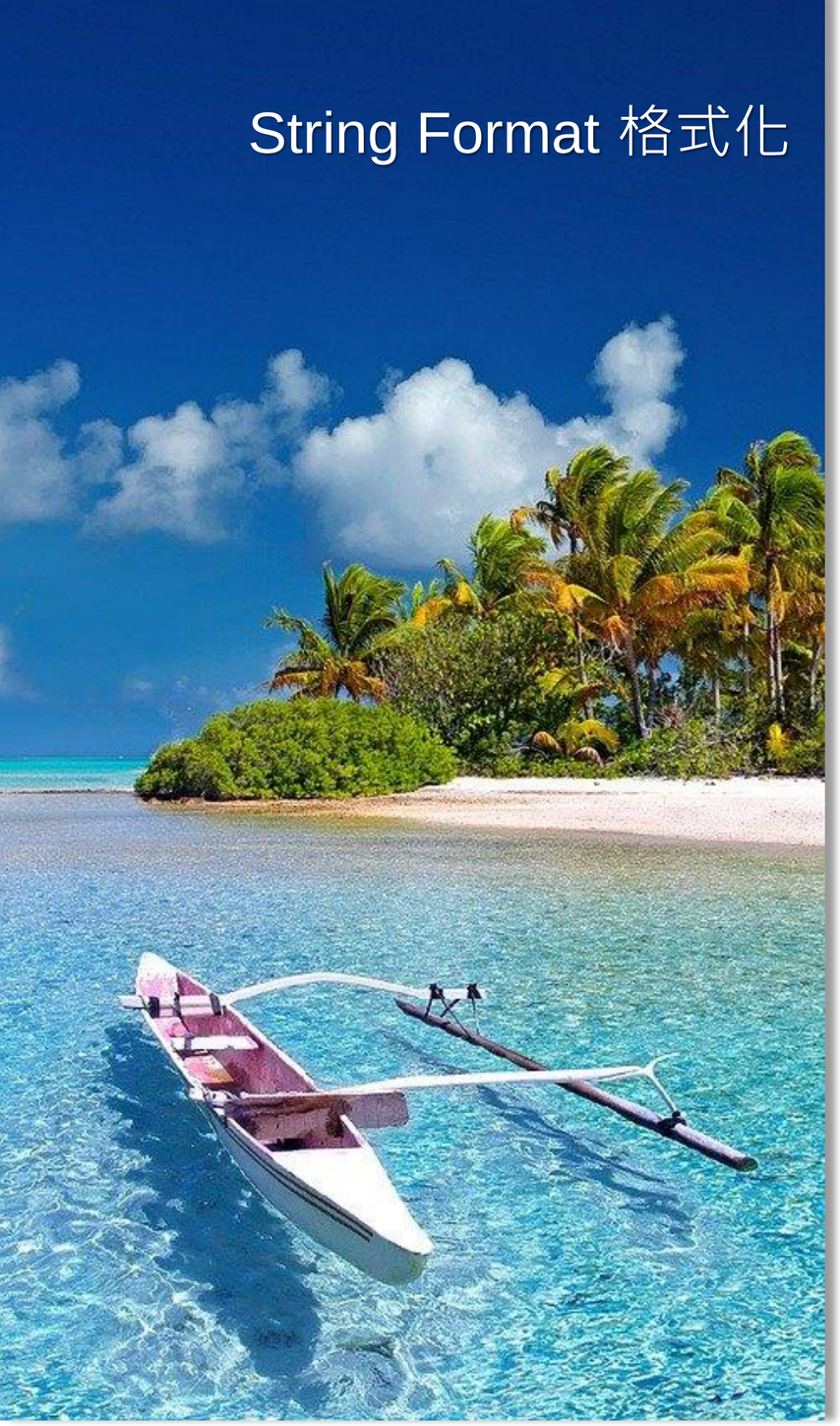
格式化輸出方法	傳回值	說明
format(String format, Object... args)	String	根據指定的格式化規則 (format) 及參數內容 (args)，傳回一個格式化字串

```
int id = 25;
String name = “王小明”;

System.out.printf(“學號: %05d\t姓名: %s%n”, id, name);
```

PrintStream 類別

格式化輸出方法	傳回值	說明
printf(String format, Object... args)	PrintStream	根據指定的格式化規則 (format) 及參數內容 (args) 列印輸出

String Format 格式化



常用格式化輸出表

資料類型	說明
%b, %B	格式化成布林類型。
%d, %o, %x	整數類型(十,八進位,十六進位)。
%f, %g	浮點數類型(浮點數,科學符號)。
%s, %S	格式化成字串類型。
%t, %T	格式化成日期時間類型。
%n	代表換行。

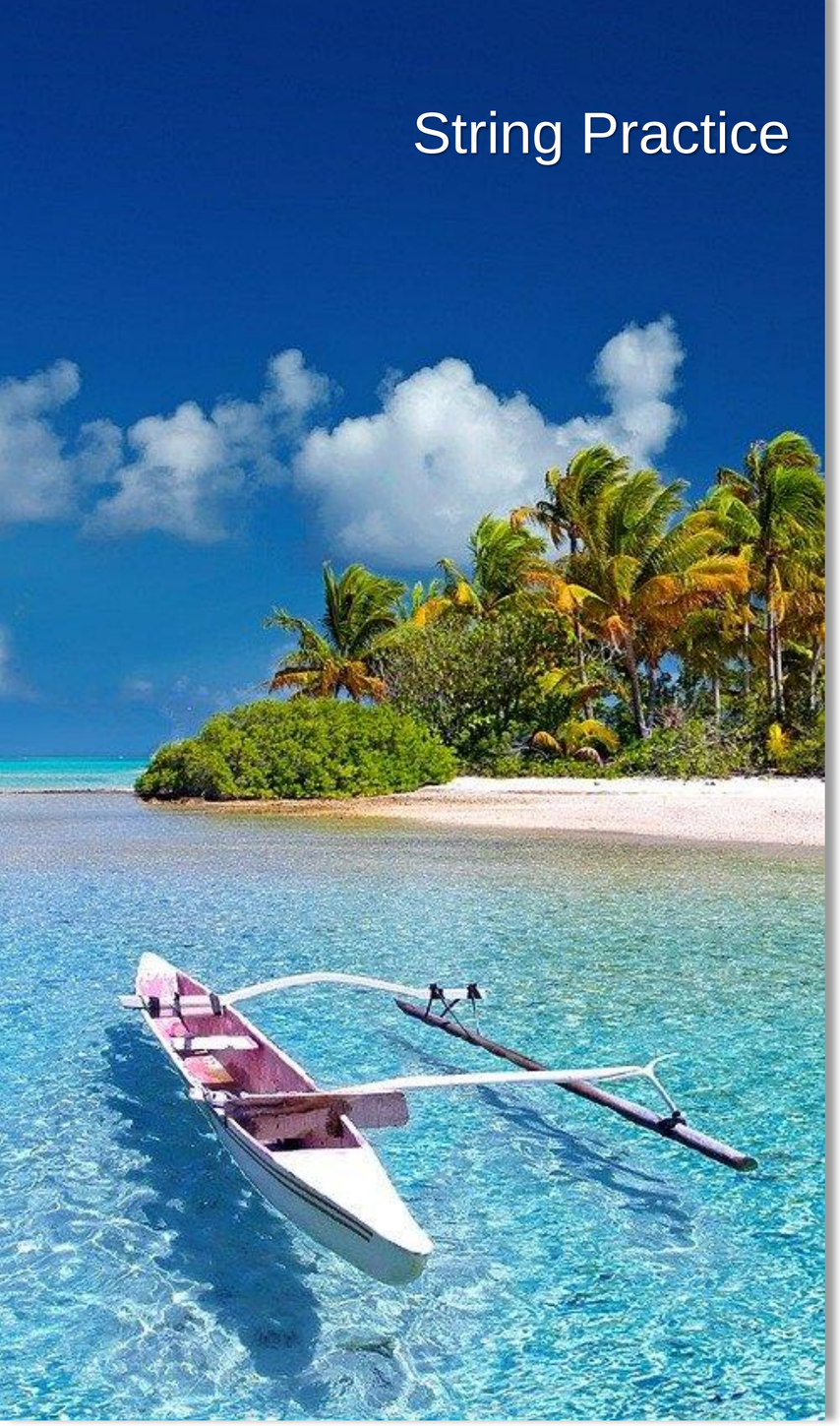
特殊符號	說明
$	指定要格式化的來源。(1$,2$)
<	使用前一個格式化資料來源。
0	使用0來當填充字元。
#	不使填充字元。
,	代表會加上千分位數。
.	代表小數點。

日期	說明
%td	日，共2位，不足部分補0。
%tm	月，共2位，不足部分補0。
%ty	年，共2位，不足部分補0。
%tY	年，共4位，不足部分補0。

時間	說明
%tH	時（24時制），共2位，不足部分補0。
%tI	時（12時制），共2位，不足部分補0。
%tM	分，共2位，不足部分補0。
%tS	秒，共2位，不足部分補0。

日期/時間	說明
%tF	相當於「%tY-%tm-%td」。
%tT	相當於「%tH:%tM:%tS」。

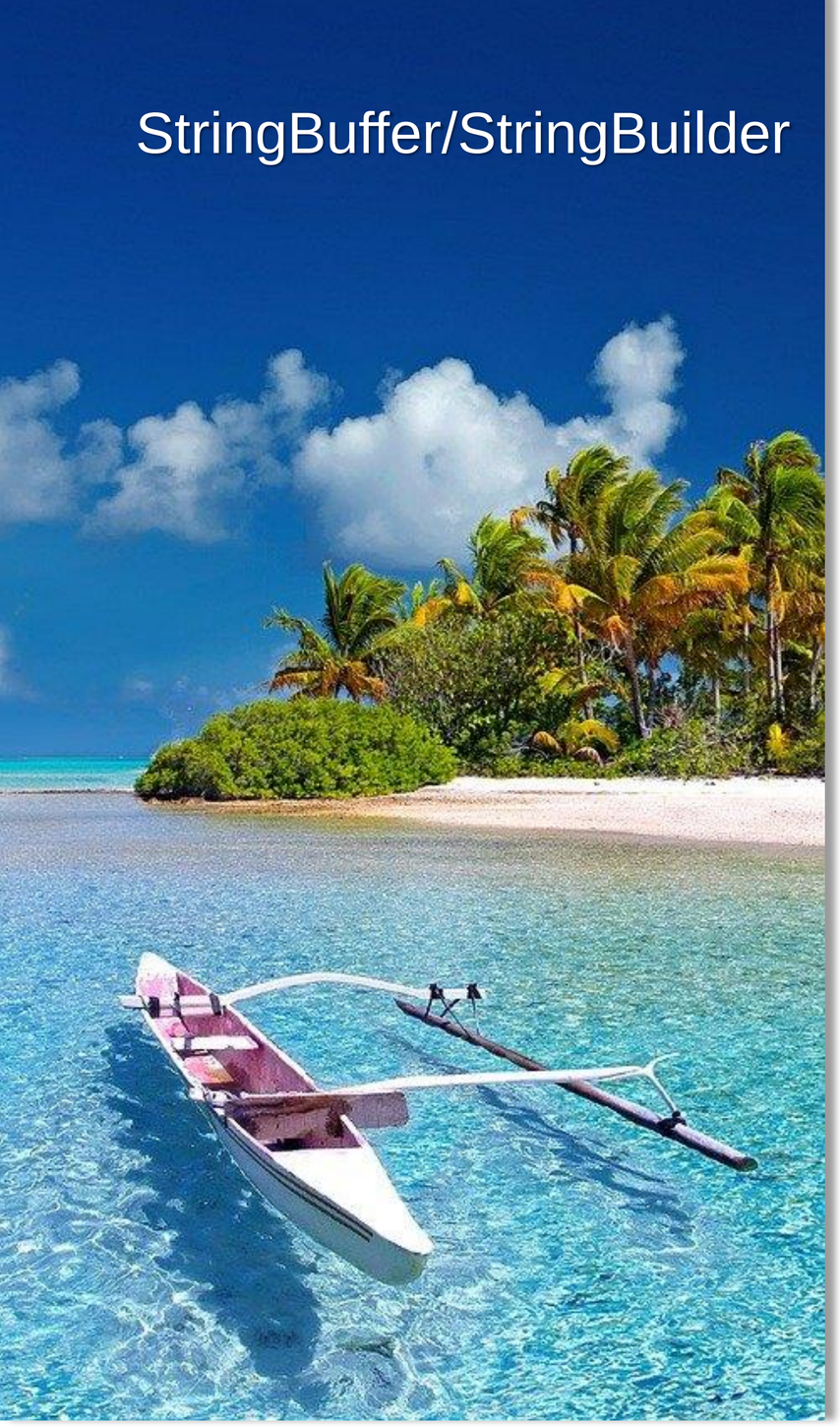
String Practice



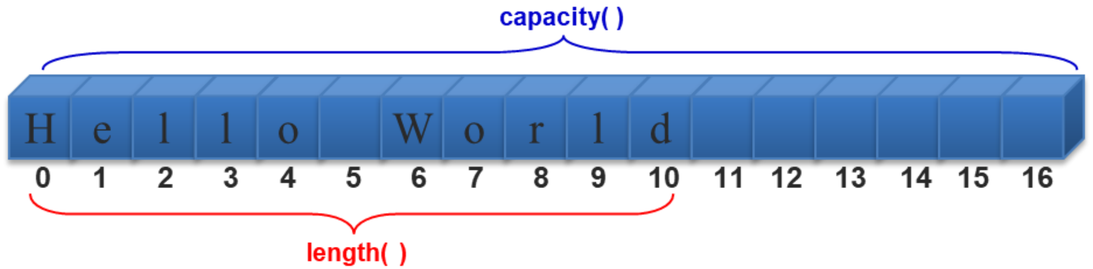
操作重點

1. 撰寫一個 CheckPalindrome.java 類別，檢查使用者輸入的字串是否為迴文
   - 某字串由前面讀取與從後面讀取結果相同稱為迴文 (Palindrome)。
   - 忽略大小差異：將字串先轉為小寫。
   - 檢視字串第一個與最後一個字元，第二個與倒數第二字元，…是否相同。
   - 字串有奇數個字元時，最中間的字元不比對。
   - 使用 printf( ) 顯示結果。

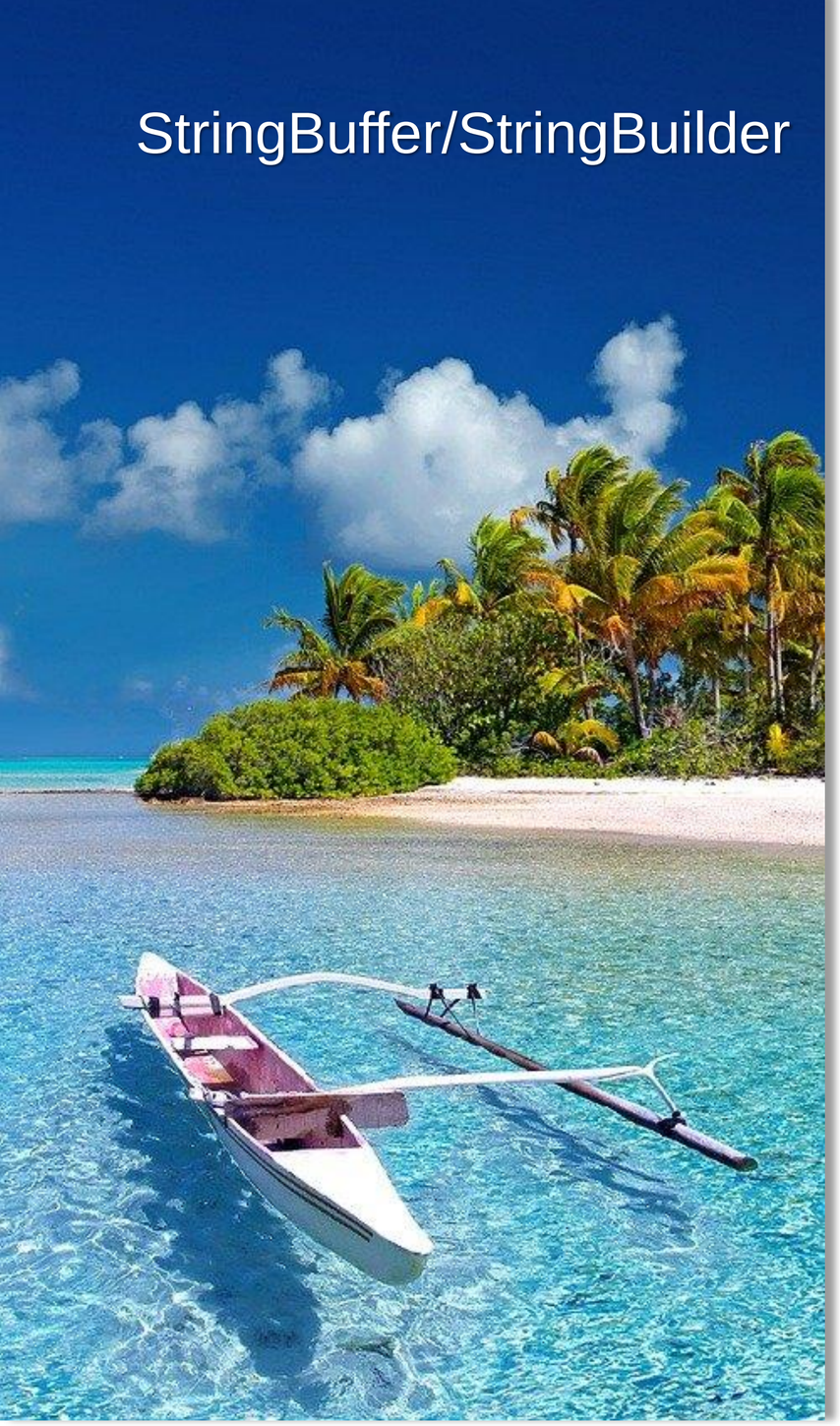
StringBuffer/StringBuilder



- 可變動內容 (Mutable) 的字串型態
  - StringBuffer
    - thread safe
    - synchrounous
  - StringBuilder
    - not thread safe
    - asynchrounous
    - Java 5.0 新增
  - 提供進階處理字串的方法
  - 緩衝區不足時，會自動加大緩衝區



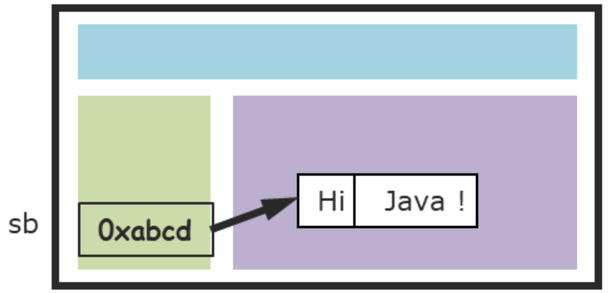
StringBuffer/StringBuilder



常用處理字串的方法

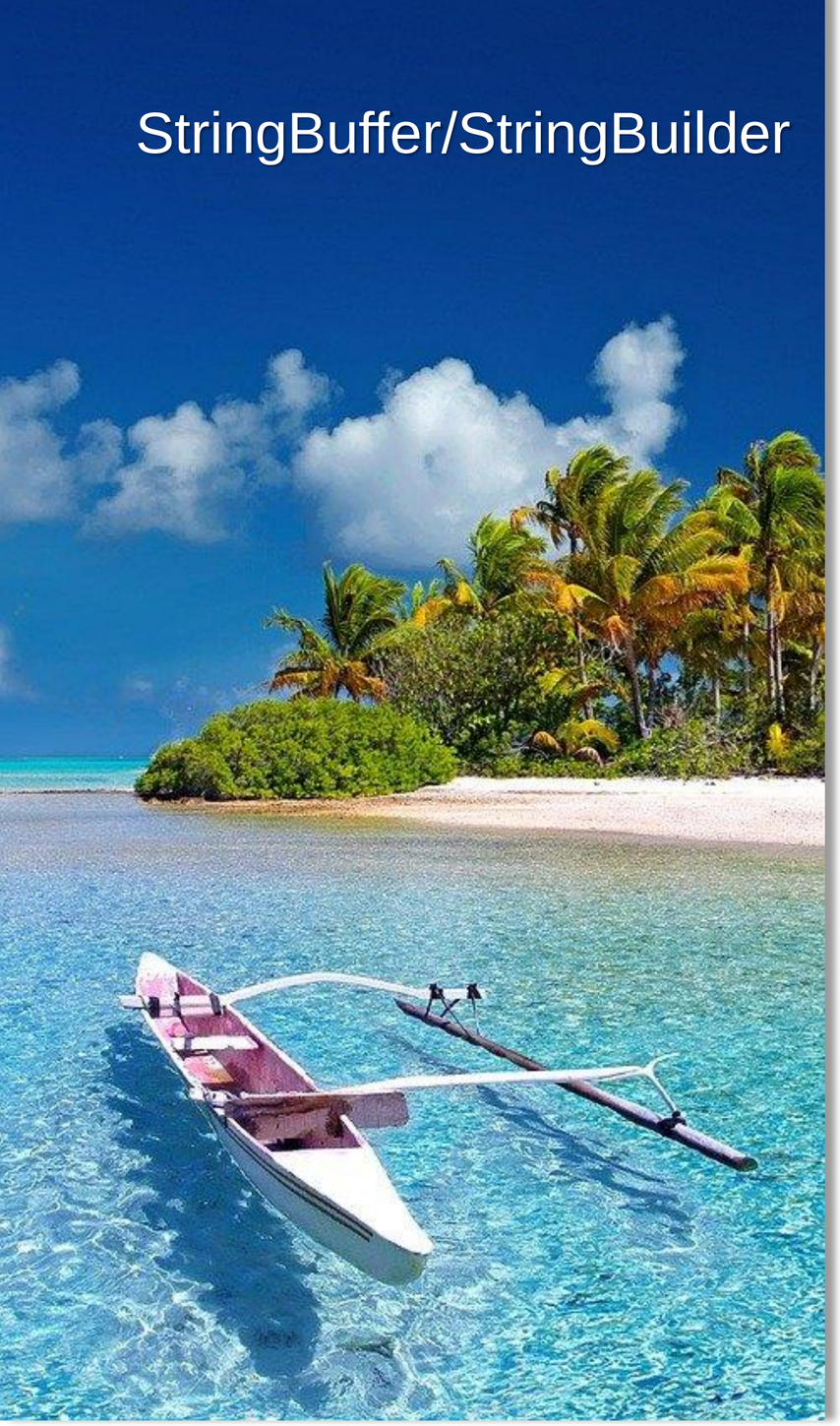
利用 append( ) 來改變字串內容

```
StringBuffer sb = new StringBuffer("Hi");
sb.append(" Java!");
System.out.println(sb.toString( ));
```



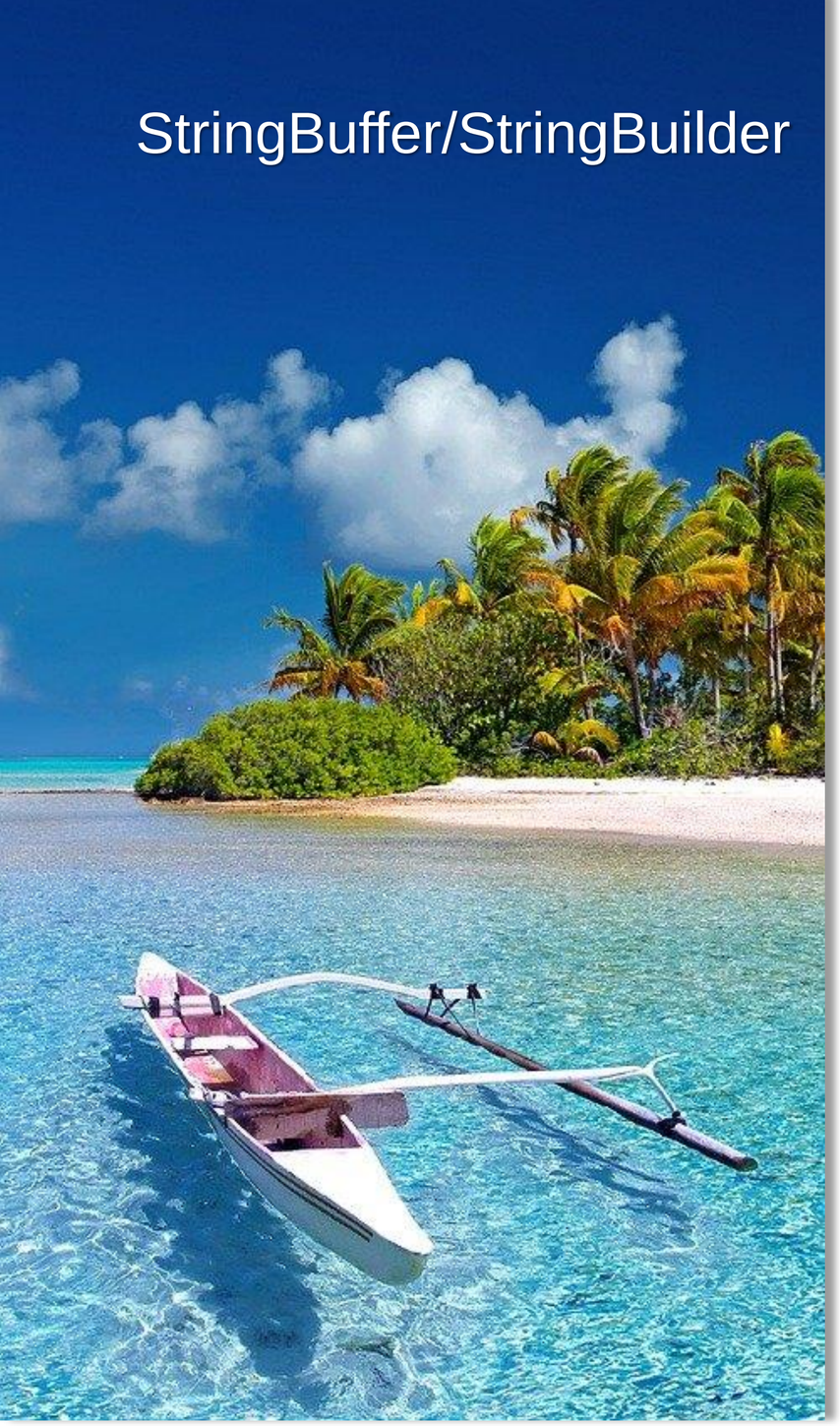
利用 insert( ) 方法來選擇指定加入的位置

```
StringBuffer sb = new StringBuffer("This is a dog");
sb.insert(10, "big ");
System.out.println(sb.toString( ));
```



常用方法

方法名稱	傳回值	說明
append(String str)	StringBuilder	將指定的字串加到此StringBuilder之後
append(StringBuilder sb)	StringBuilder	將指定的StringBuilder加到此StringBuilder之後
length( )	int	傳回此StringBuilder的字元個數
capacity( )	int	傳回此StringBuilder的容量
ensureCapacity(int minimumCapacity)	void	確保StringBuilder容量至少等於指定的最小值
charAt(int index)	char	傳回此StringBuilder中，指定位置的字元值
indexOf(String str)	int	傳回此StringBuilder中，首次出現指定子字串的位置
lastIndexOf(String str)	int	傳回此StringBuilder中，最後出現指定子字串的位置
insert(int offset, String str)	StringBuilder	在StringBuilder指定位置之後，插入指定字串
delete(int start, int end)	StringBuilder	將StringBuilder中，start到end位置中的字元移除
replace(int start, int end, String str)	StringBuilder	將StringBuilder中，start到end位置中的內容以指定字串取代
reverse( )	StringBuilder	將StringBuilder字元順序顛倒
toString( )	String	將StringBuilder變為一個內容不可變更的新字串傳回
substring(int start, int end)	String	將StringBuilder中指定位置變為一內容不可變更的新字串傳回

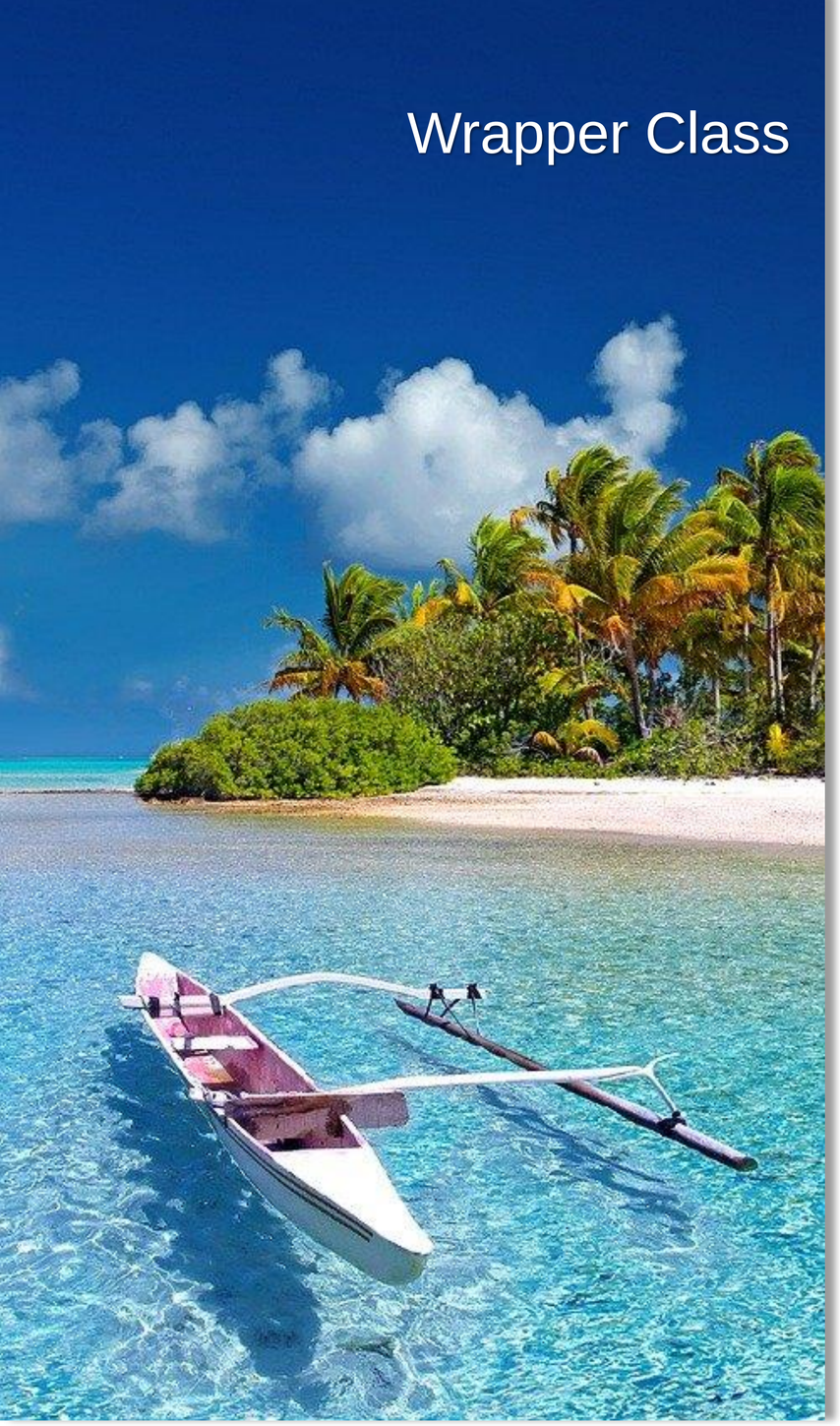


範例

```
01  public class TestStringBuffer {
02      public static void main(String[] args) {
03          StringBuffer sb1 = new StringBuffer("Hi");
04          sb1.append(" Java!");
05          System.out.println(sb1.toString( ));
06
07          StringBuilder sb2 = new StringBuilder("This is a dog");
08          sb2.insert(10, "big ");
09          System.out.println(sb2.toString( ));
10
11          StringBuffer sb3 = new StringBuffer("Hi Java");
12          System.out.println(sb1.equals(sb3));
13      }
14  }
```

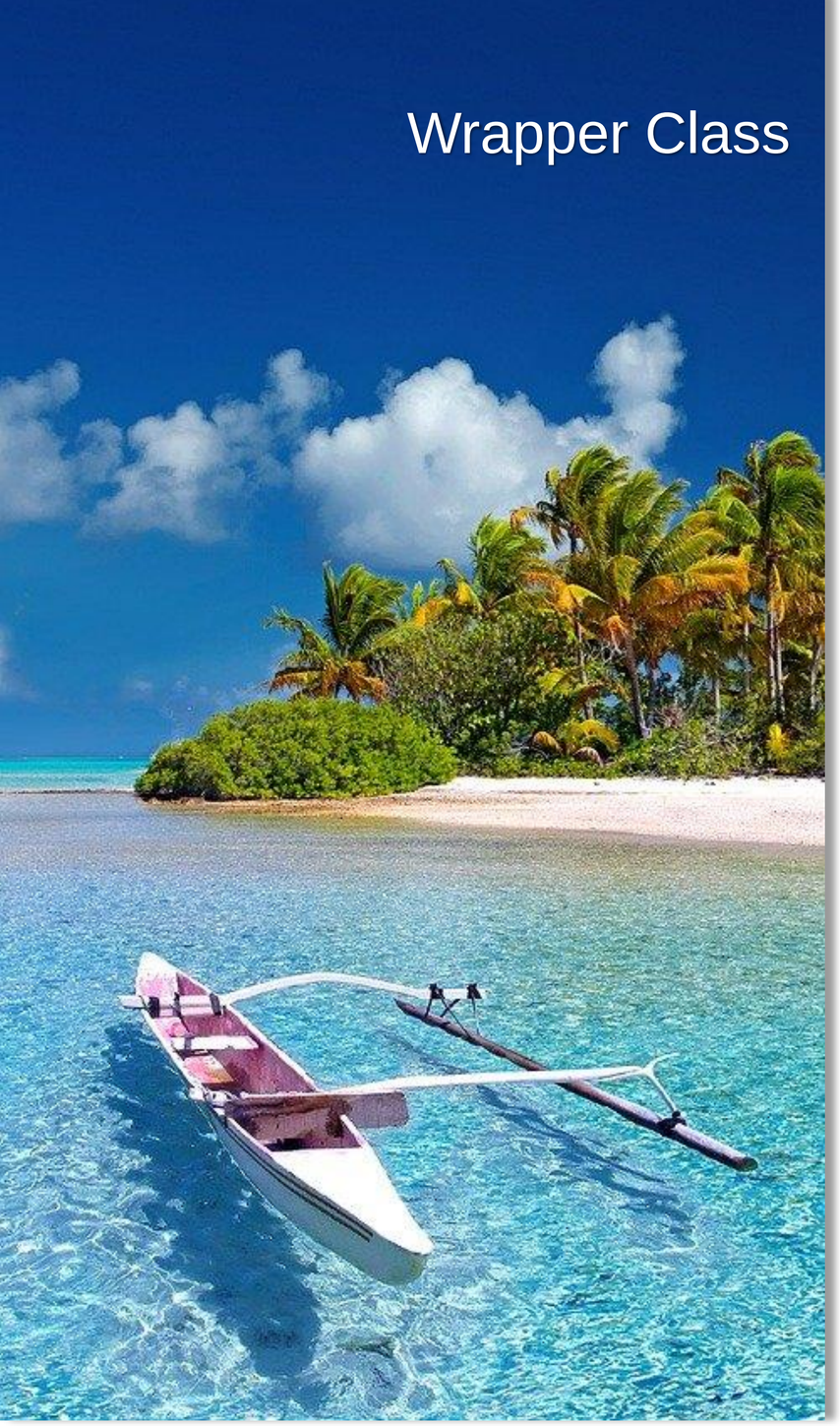
包覆類別 Wrapper Class

Wrapper Class



- 基本資料型別 (primitive type) 有一個對應包覆類別 (Wrapper Class)
  - 將基本型態資料包覆成物件，以物件方式操作
    - 物件包含更多資訊及操作
    - 集合架構只接受物件型態
  - 裝箱 boxing
    - 基本型別轉成包覆物件
  - 拆箱 unboxing
    - 包覆物件中取出基本型別資料

基本資料型別 Primitive type	包覆類別 Wrapper Class	父類別
boolean	Boolean	Object
char	Character	
byte	Byte	Number
short	Short	
int	Integer	
long	Long	
float	Float	
double	Double	



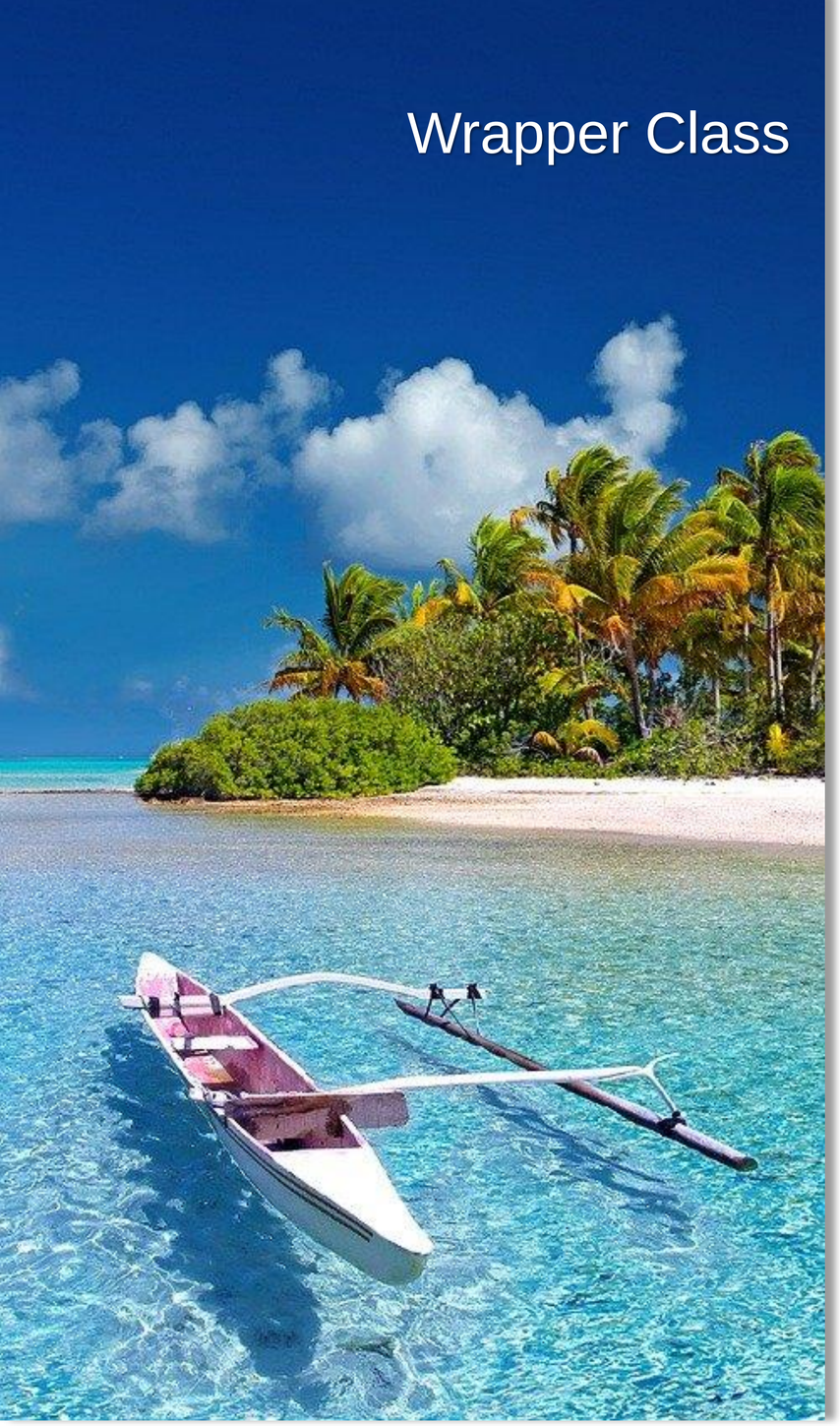
常用方法

方法名稱	傳回值	說明
equals(Object obj)	boolean	比較此 Integer 物件與傳入物件內容是否相等
intValue( )	int	以 int 型別傳回 Integer 物件的值
byteValue( )	byte	以 byte 型別傳回 Integer 物件的值
longValue( )	long	以 long 型別傳回 Integer 物件的值

- 工具方法
  - Ex ：將表示基本型別的字串轉為基本型別

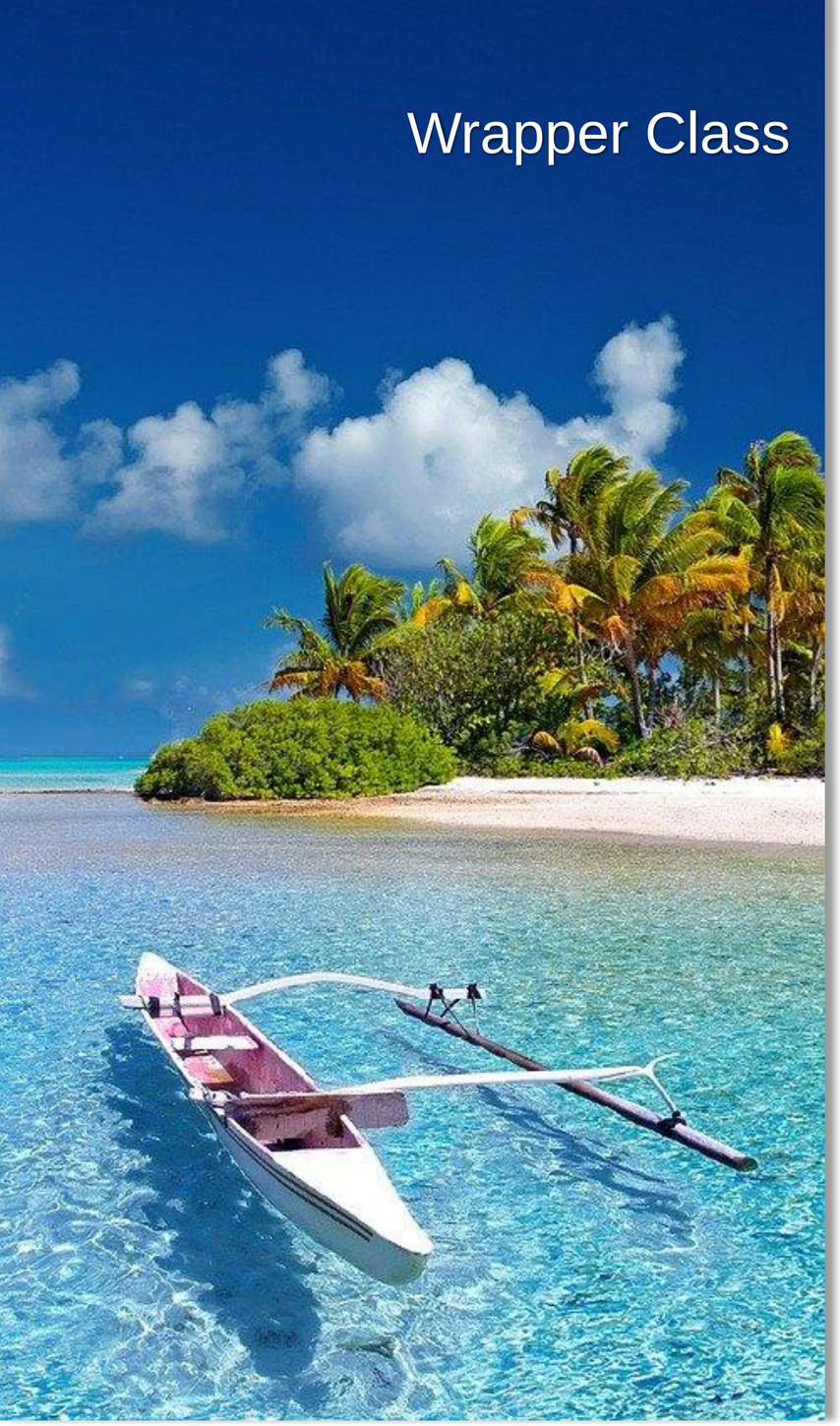
方法名稱	傳回值	說明
parseInt(String s)	int	將表示 int 的字串轉換成 int 型別
getInteger(String nm)	Integer	取得指定名稱對應的整數型態系統屬性值

```
int  i = Integer.parseInt ("20");
```



Wrapper Class

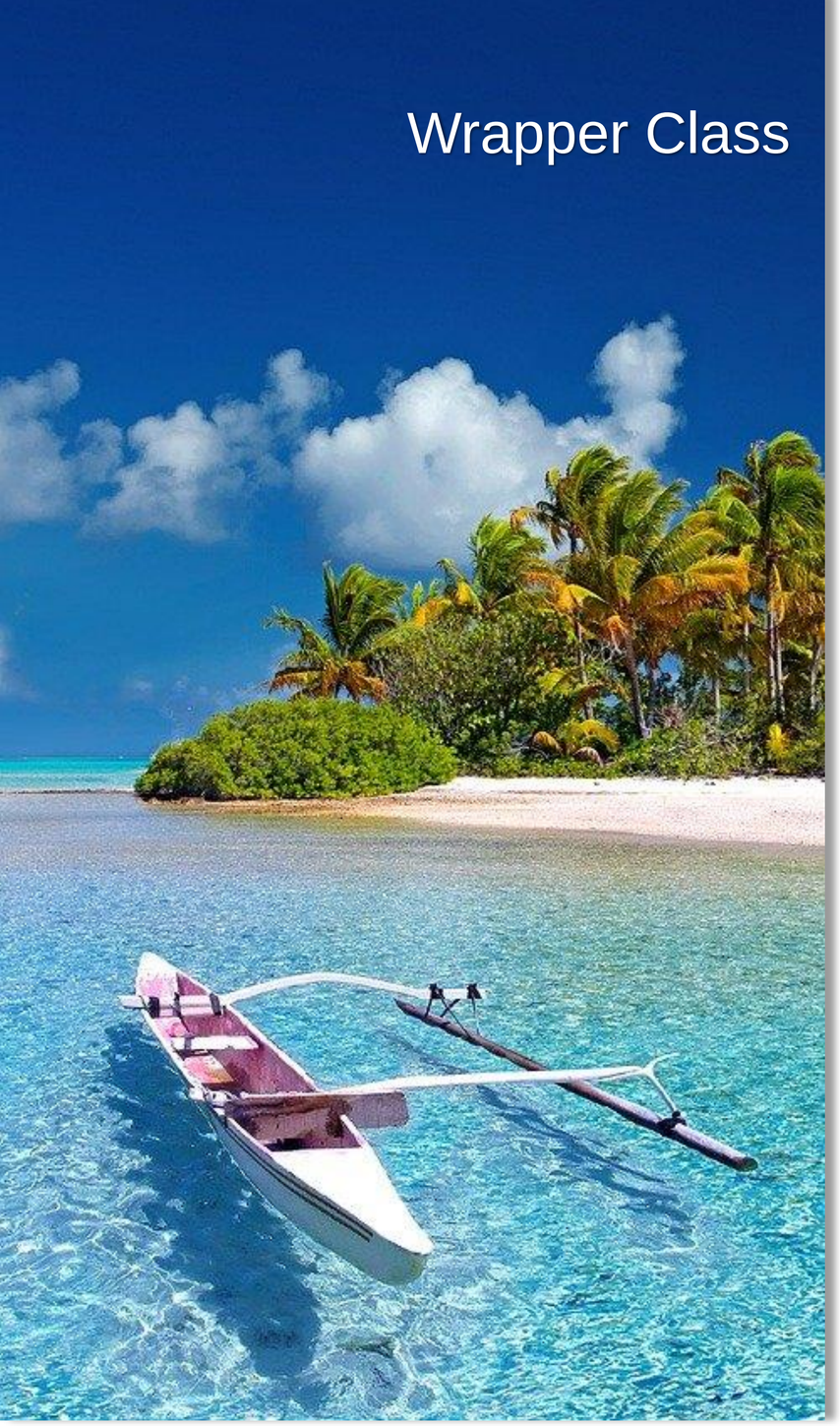
- 傳統包覆類別寫法
  - Integer number=new Integer("100");
  - int x=number.intValue();
  - System.out.println(x);
- 在se5平台上,可以自動裝箱及拆箱
  - Integer n=100;//裝箱,將int轉成Integer
  - int y=n;//拆箱,將Integer 轉成int
  - System.out.println(y);
- 基本資料型別之裝箱
  - Character c='a';
  - Byte b=15;
  - Short s=20;
  - Integer i=50;
  - Long l=100L;
  - Float f=32.3F;
  - Float f1=32;//此行編譯不行,因其是類別Integer無法轉Float,Float f1=32F,或是 Float f1=(float)32才可
  - float f2=32;//基本資料型別會自動轉型 int 轉 float
  - Double d=12.34;



數字字串轉型

- 字串轉數字 Integer.parseXXX
- 數字轉字串
  - Integer.toString()
  - String.valueOf()

```
01  public class TestArgs {
02      public static void main(String[] args) {
03              System.out.println("Total  is: " + (args[0]+args[1]));
04               int arg0 = Integer.parseInt(args[0]);
05               int arg1 = Integer.parseInt(args[1]);
06               System.out.println("Total  is: " + (arg0+arg1));
07          }
08  }
```



Integer 內容值比較

```
01 public class TestInteger {
02     public static void main(String[] args) {
03         Integer i1 = new Integer(20);
04         Integer i2 = new Integer("20");
05         Integer i3 = Integer.valueOf(20);
06         Integer i4 = Integer.valueOf("20");
07         System.out.println(i1 == i2);          → false
08         System.out.println(i3 == i4);          → true   // Integer Cache
09         System.out.println(i1.equals(i2));     → true   // 比較實際內容
10         System.out.println(i3.equals(i4));
11     }
12 }
```

java為了加快運算在使用字面值或valueOf()為-128~127間的值,都會加以重覆使用,不過對於使用new的方式則沒有作用

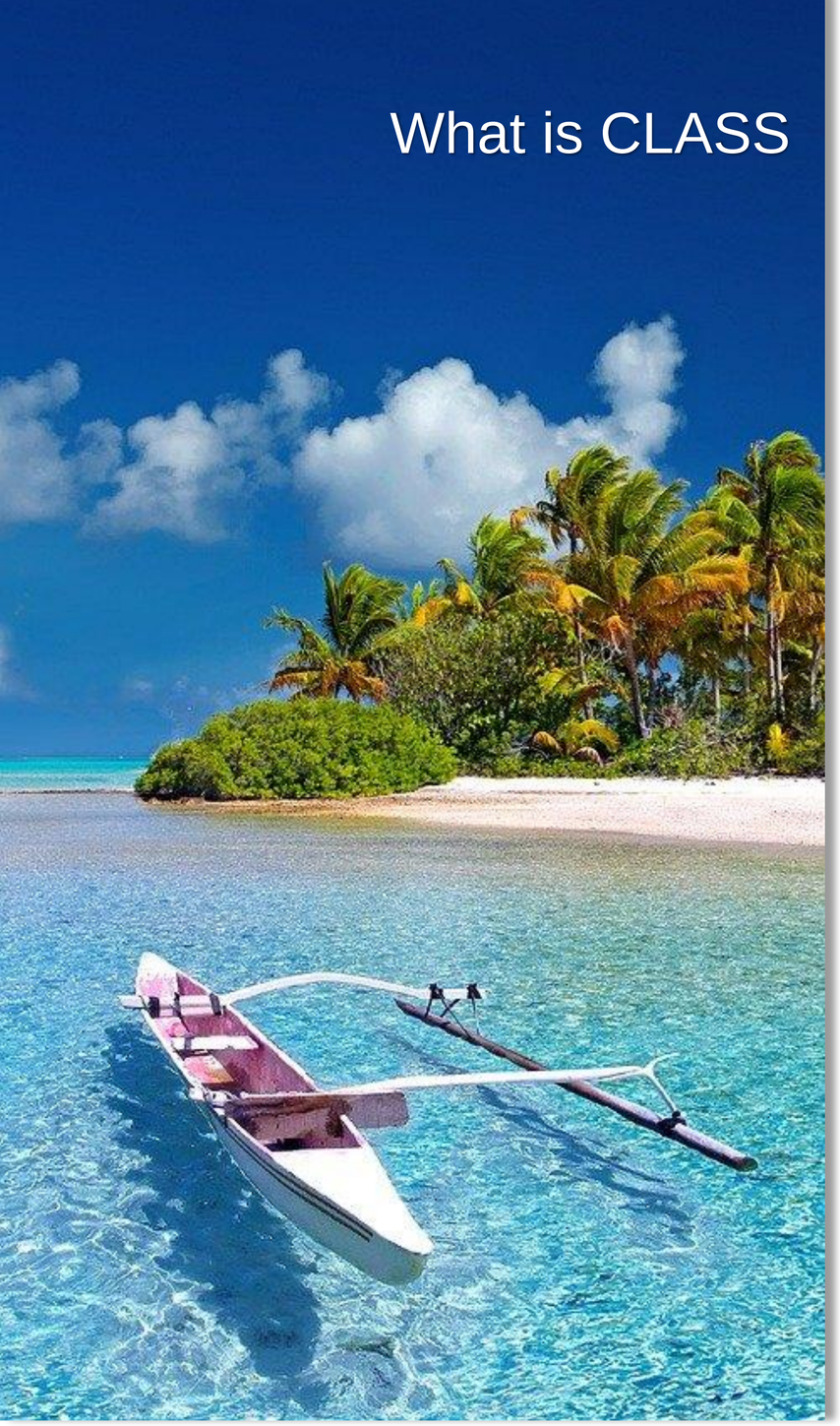
```
01 public class TestInteger {
02     public static void main(String[] args) {
03         Integer i1 = new Integer(200);
04         Integer i2 = new Integer("200");
05         Integer i3 = Integer.valueOf(200);
06         Integer i4 = Integer.valueOf("200");
07         System.out.println(i1 == i2);          → false
08         System.out.println(i3 == i4);          → false  // 超過Integer Cache 範圍
09         System.out.println(i1.equals(i2));     → true   // 比較實際內容
10         System.out.println(i3.equals(i4));
11     }
12 }
```

System.out.println(10>5?1:1.0);//此行不管條件為true或false答案皆為1.0

因為其會自動轉型,所以1會轉型為1.0

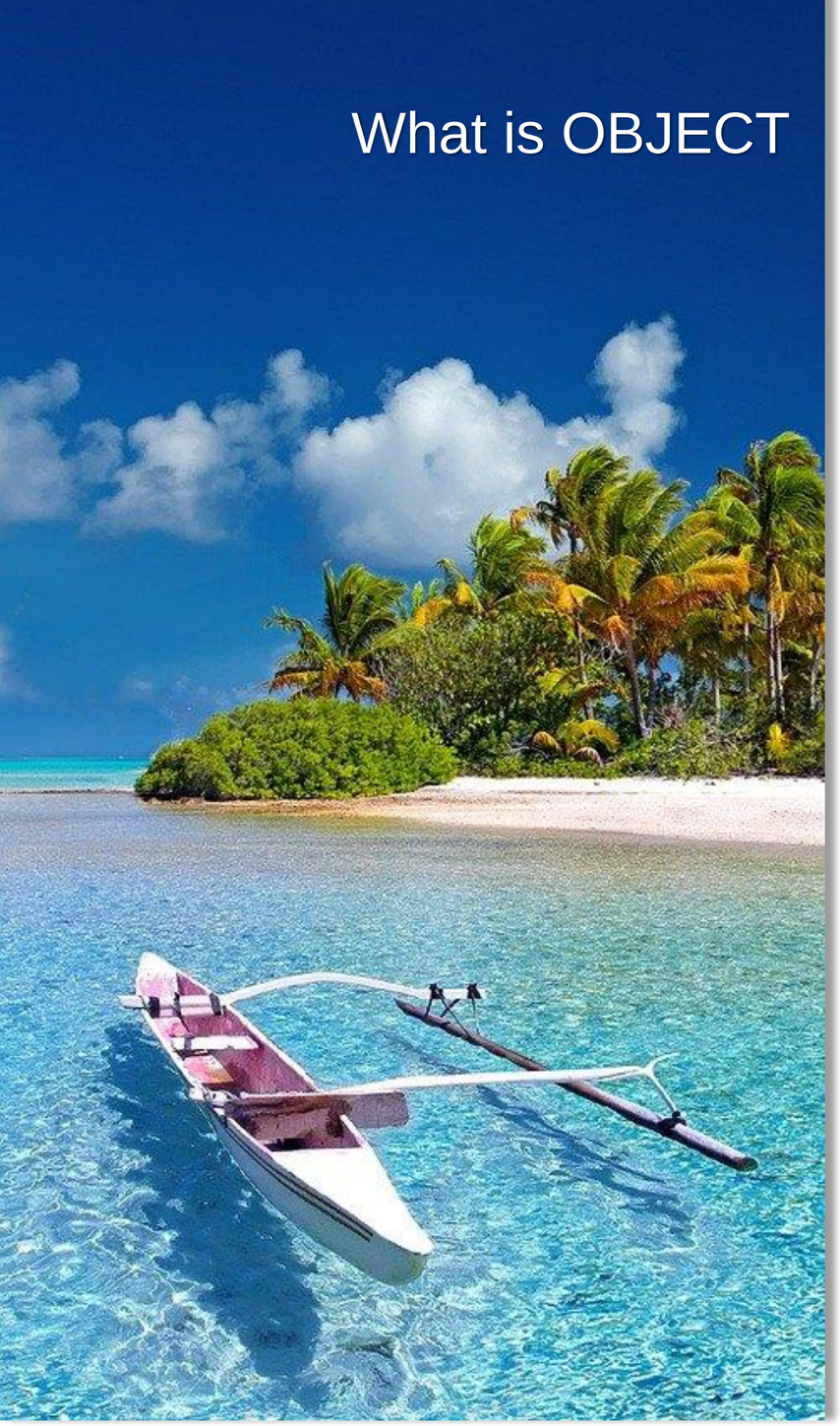
類別設計 Class Design

What is CLASS



什麼是類別

- 類別是一群具有相同性質的物件集合，在真實世界中，常將事物依其特性歸類為不同的類別，而類別的實體則可以稱為物件
- 可以是設計圖或分類，有其特定的性質，車子與人是不同的類別，沒有實體的概念

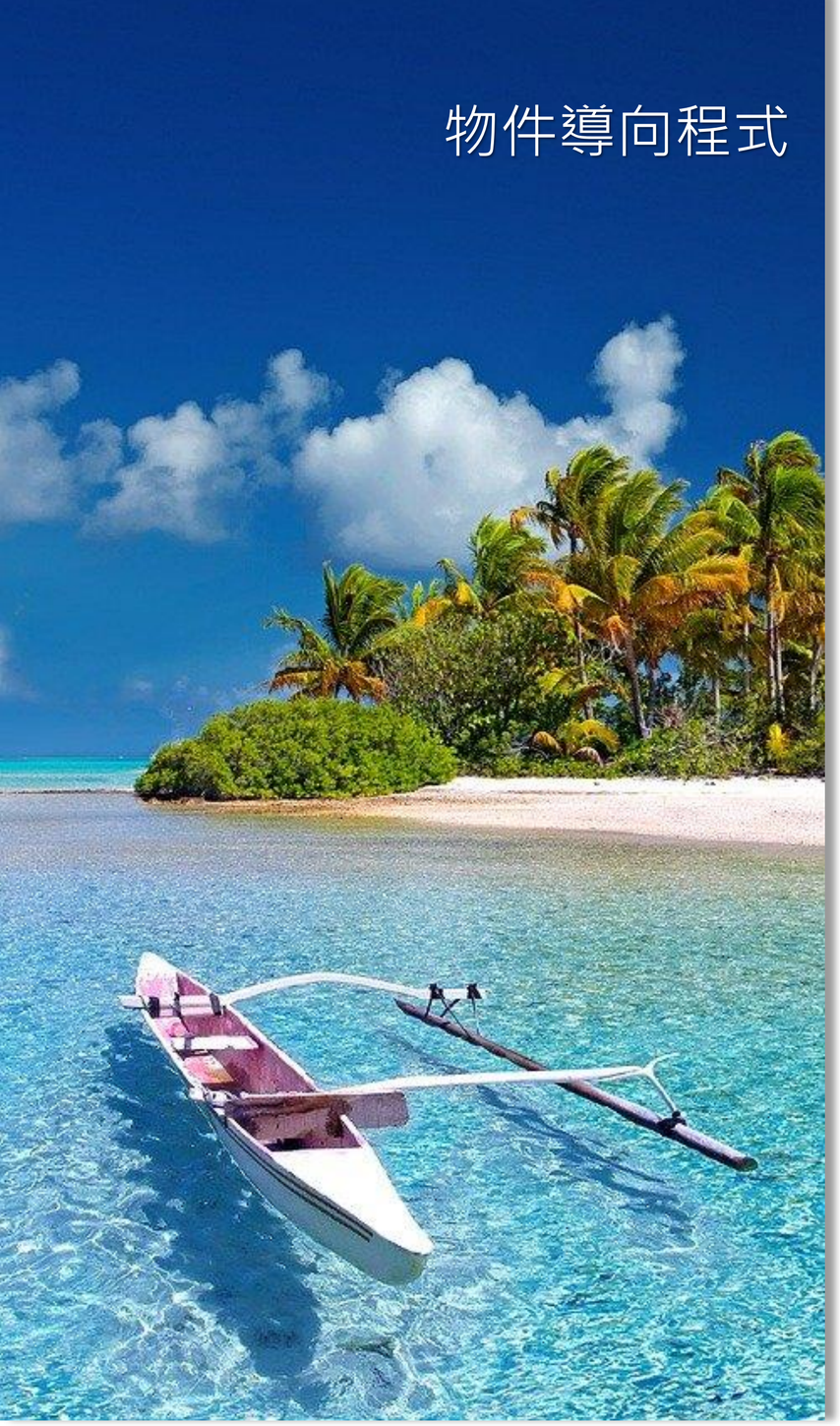


什麼是物件

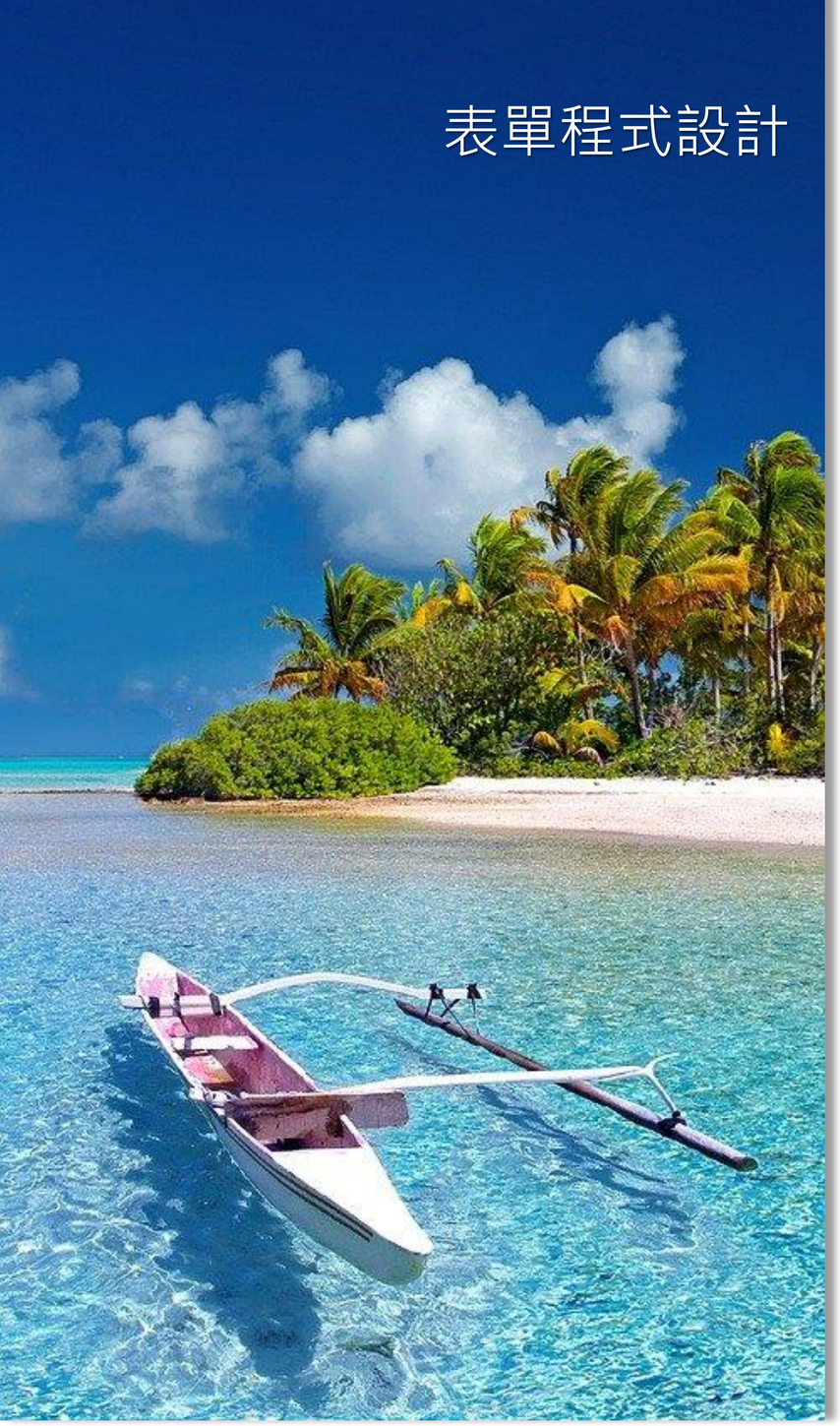
- 看的到，摸的到，可以是一個個體或很多個體的組成，具有特徵（屬性）或某些行為（方法）

類別	物件
車子	轎車，腳踏車，公車，三輪車
人	您，我，小明，他，馬英九

物件導向程式



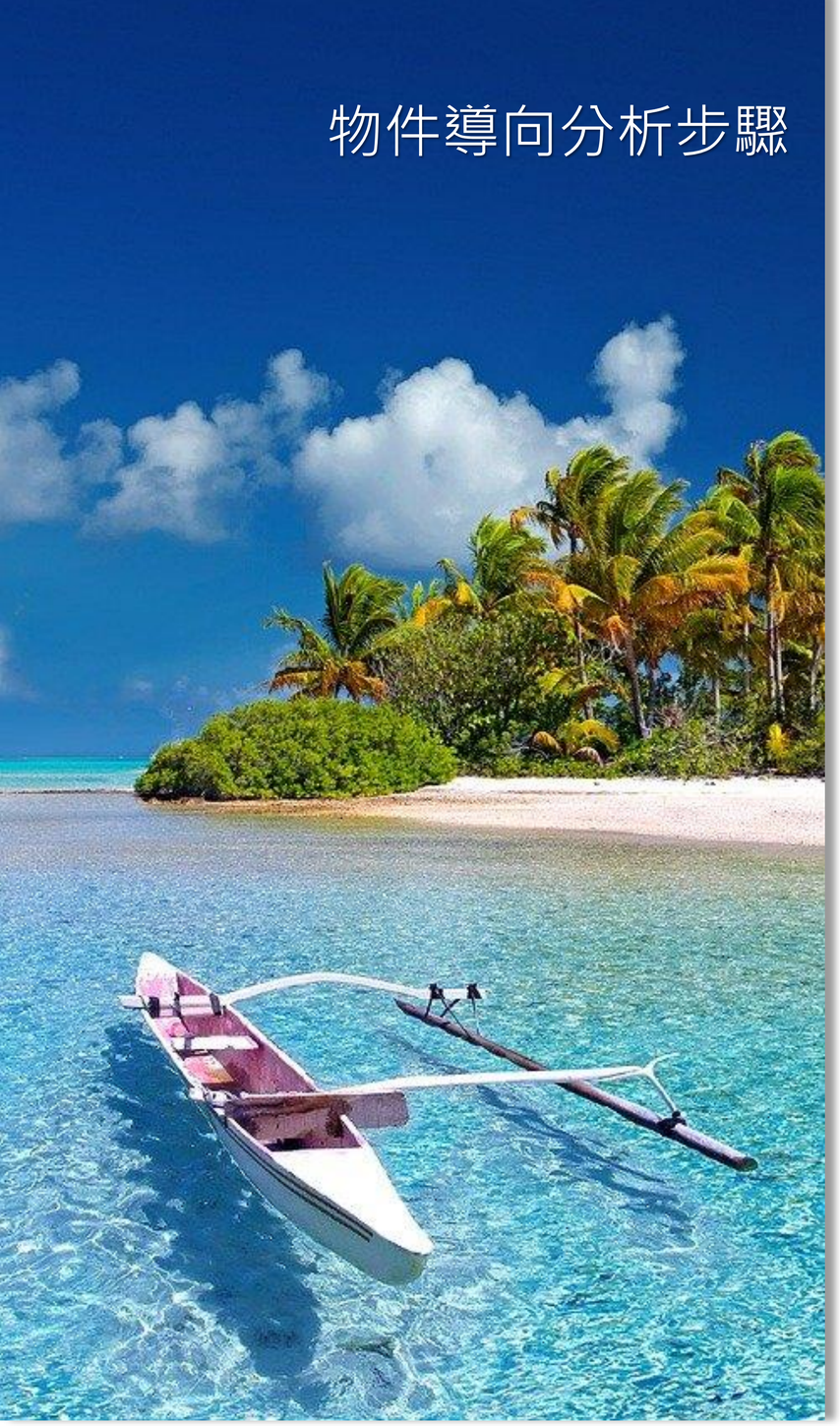
- 在物件導向程式中，為了使用程式碼模擬真實世界中物件的特徵與行為，因此物件的特徵以資料屬性（屬性）來描述，物件的行為以程式功能（方法）來描述。
- 類別：定義了物件的成員，成員包含物件的屬性與方法
- 物件：是依照類別定義的規範所建構出來的一個實體



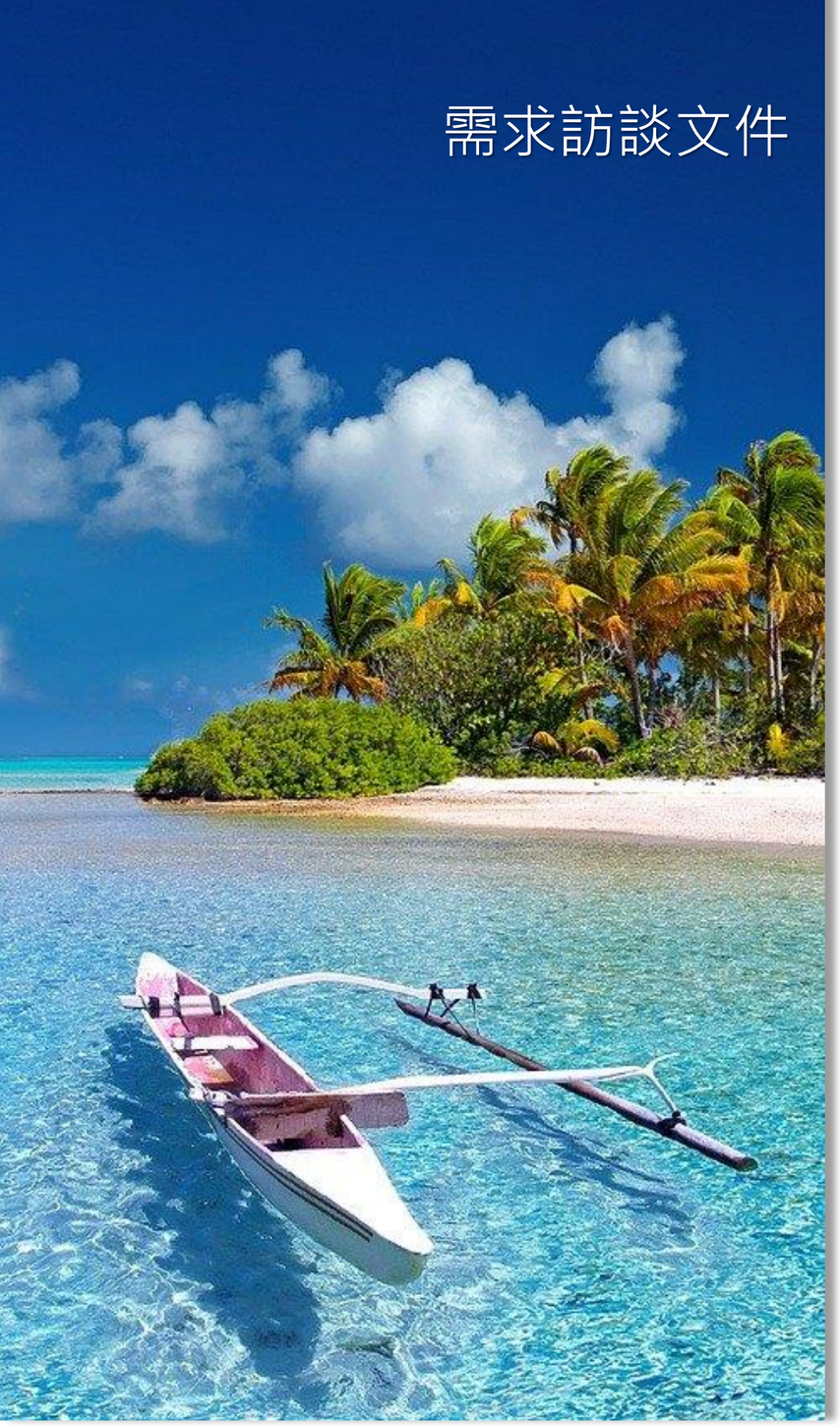
表單程式設計

- 設計表單程式時所需要的按鈕，當我們需要多個按鈕時，不需要重新設計，而由按鈕類別實作（new)按鈕物件就可以使用了
- 屬性：按鈕的文字，文字色彩，按鈕色彩
- 方法：滑鼠按下時，將資料送出

物件導向分析步驟



- 需求訪談文件 Problem Statement
- 辨識問題領域 Identify Problem Domain
- 辨識物件 Identify Object
- 確認物件 Recognized Object
- 辨識物件之屬性及行為 Identify Object’s Attribute & Operation

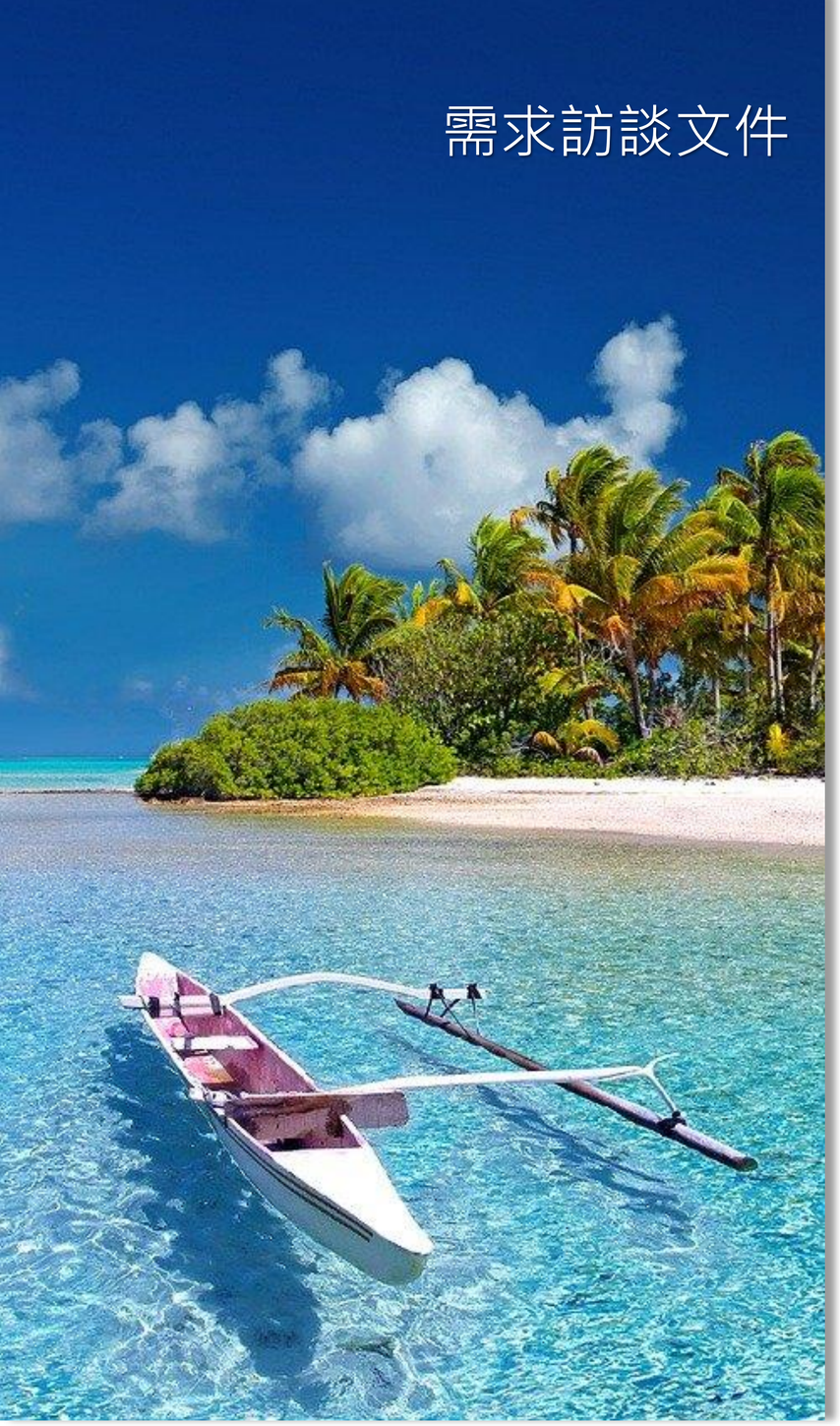


需求訪談文件

Duke's Choice Inc. sells clothing from their catalog. Business is growing 30 percent per year, and they need a new order entry system. You have been contracted by Duke's Choice to design the new system.

Duke's Choice produces an online catalog every three months and mails it to subscribers. Each shirt in the catalog has an item identifier (ID), one or more colors (each with a different color code), one or more sizes, a description, and a price.

Duke's Choice accepts checks and all major credit cards.



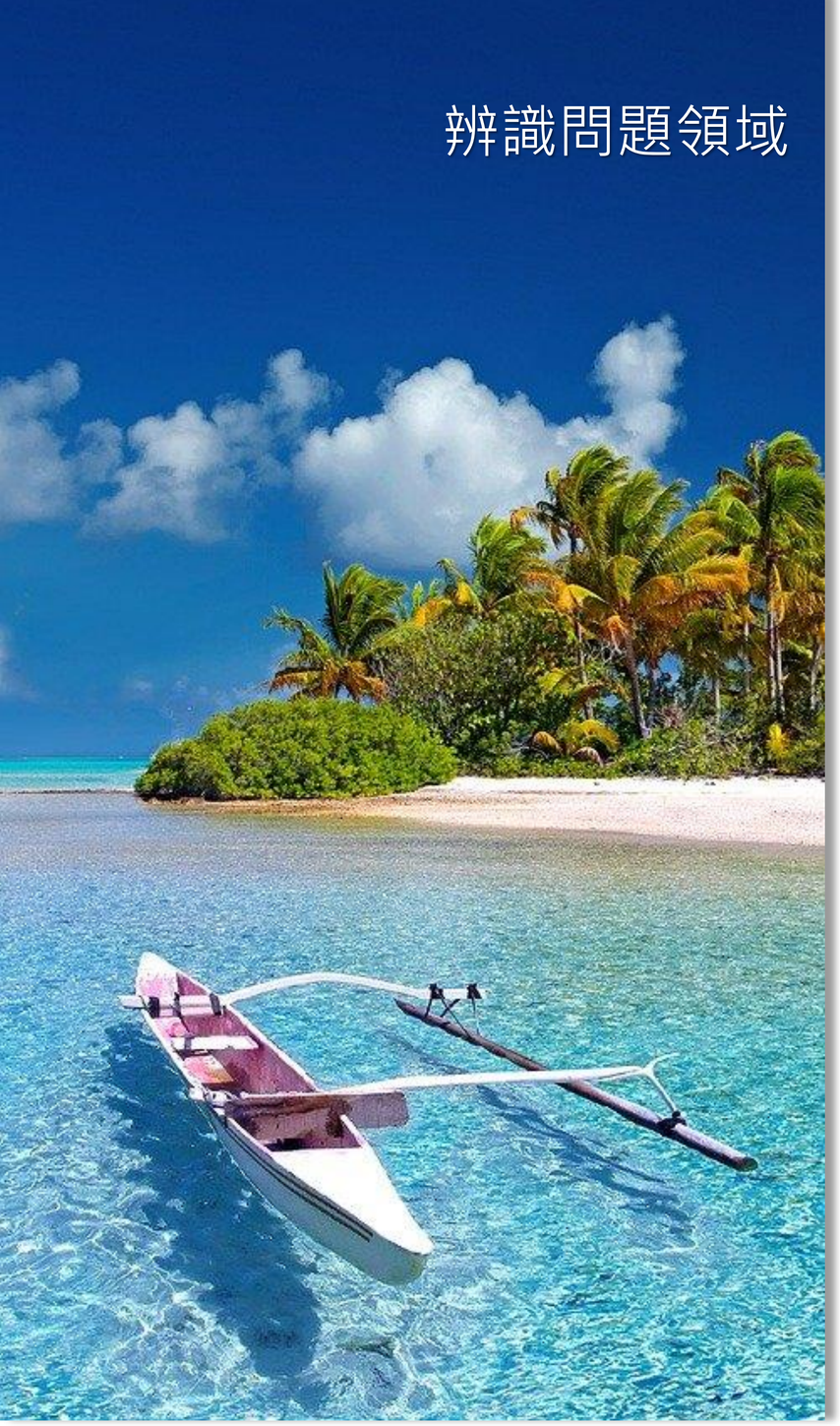
需求訪談文件

As the customer enters the order online, each item's availability (quantity-on-hand) is verified. If one or more items are not currently available (in Duke's Choice's warehouse), then the item is marked as backordered until the items arrive at the warehouse. After all of the items are available, payment is verified and the order is submitted to the warehouse for assembly and for shipping to the customer's address.

When the order is received, the customer is given an order ID, which is used to track the order throughout the process.

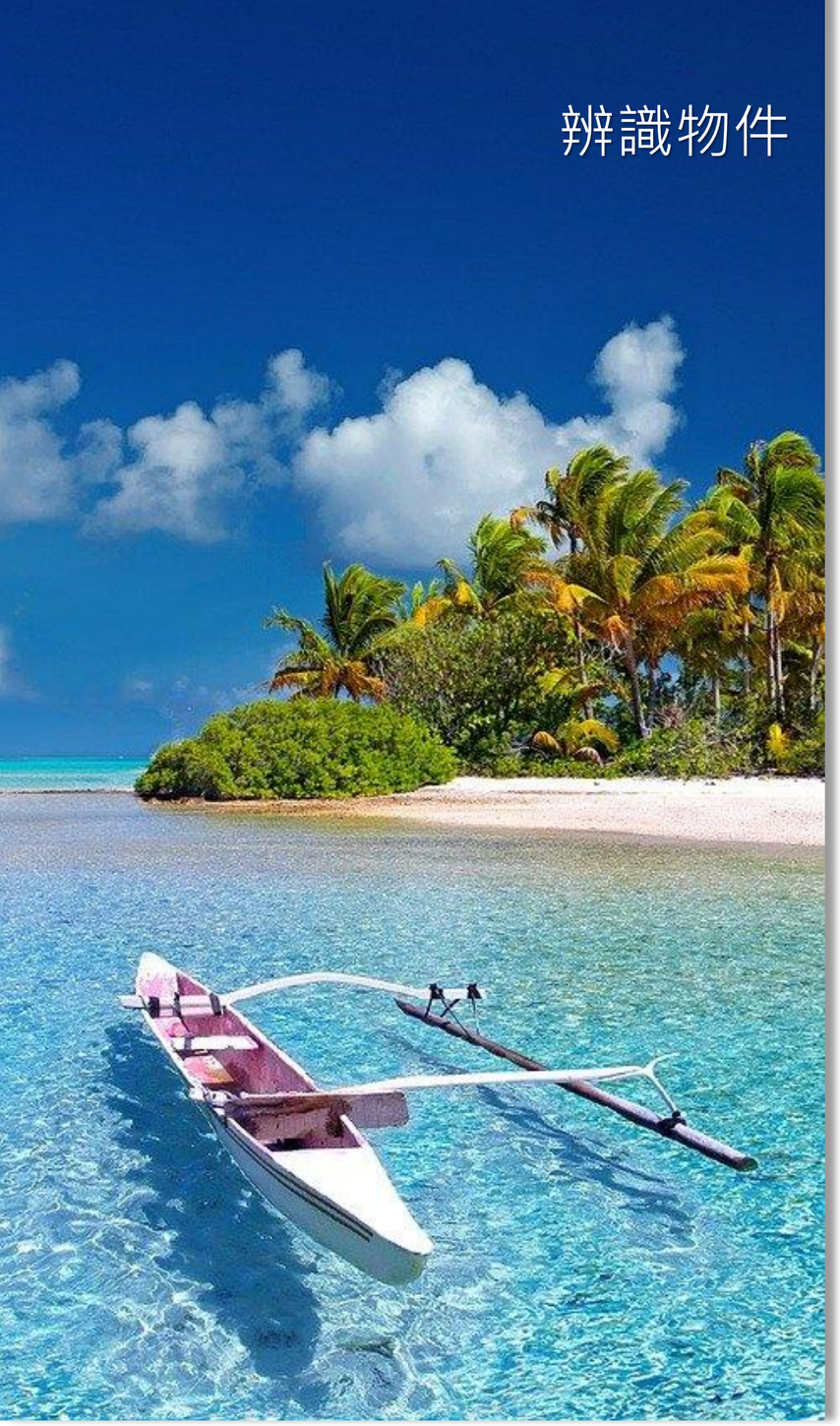
Orders that are phoned in are entered by a CSR.

辨識問題領域



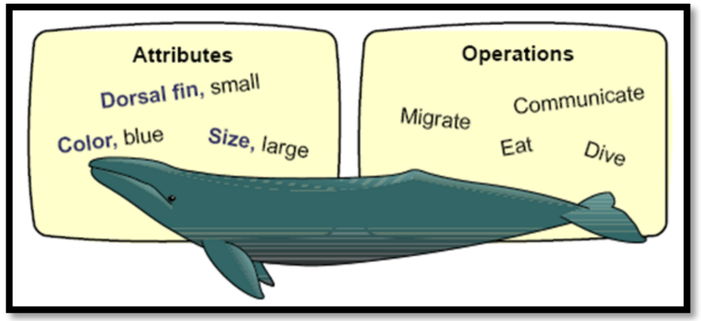
辨識系統要解決之問題的範圍

Create a system allowing the online order entry method to accept and verify payment for an order.

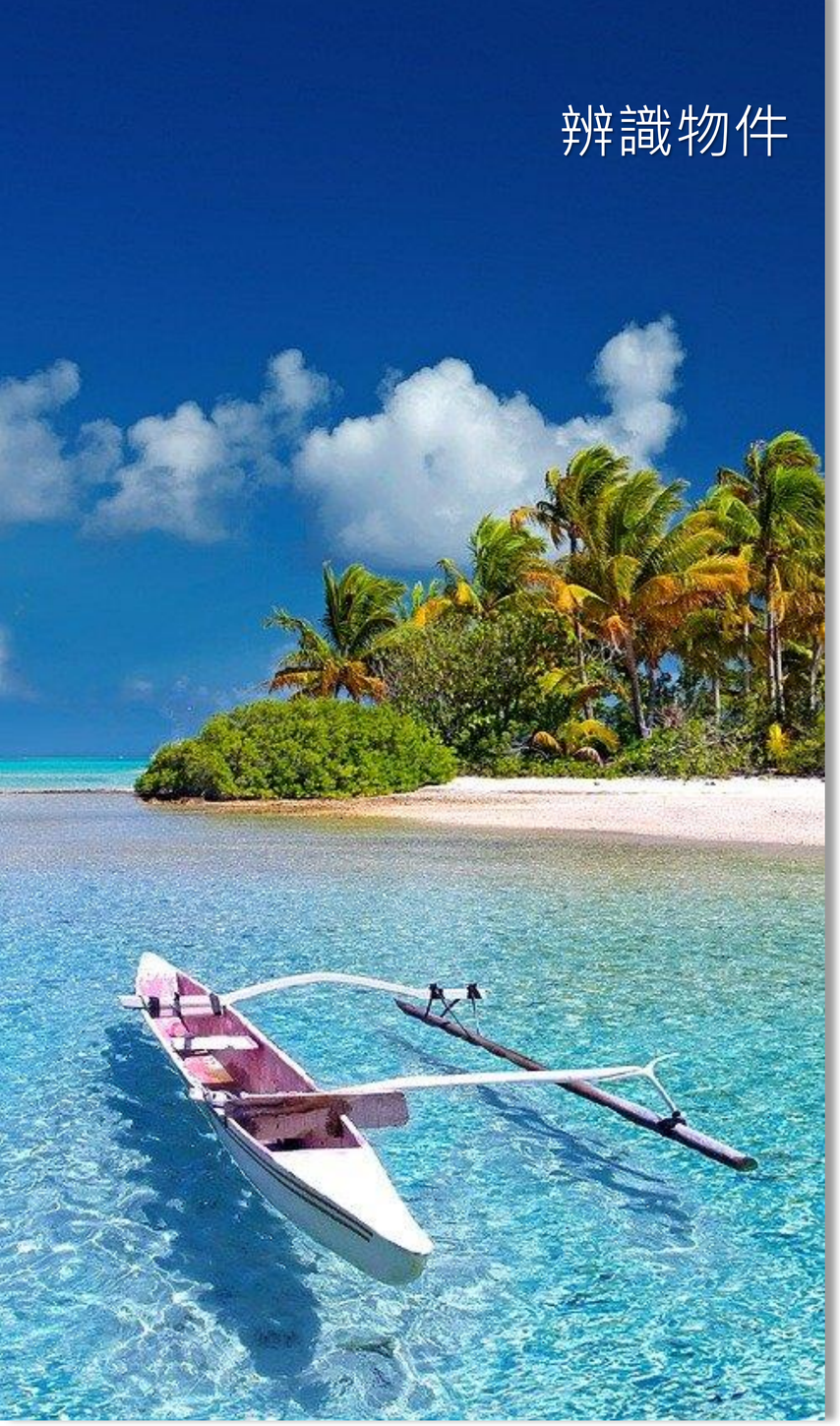


辨識物件

- 物件
  - 物件可能是實體或抽象概念
  - 物件通常為名詞
    - Account, Shirt



- 物件屬性
  - 代表其特徵、狀態
  - 屬性通常也為名詞
    - color, size
- 物件操作
  - 物件可以做的事情,行為
  - 操作通常為動詞或名詞與動詞的組合
    - display, submit order



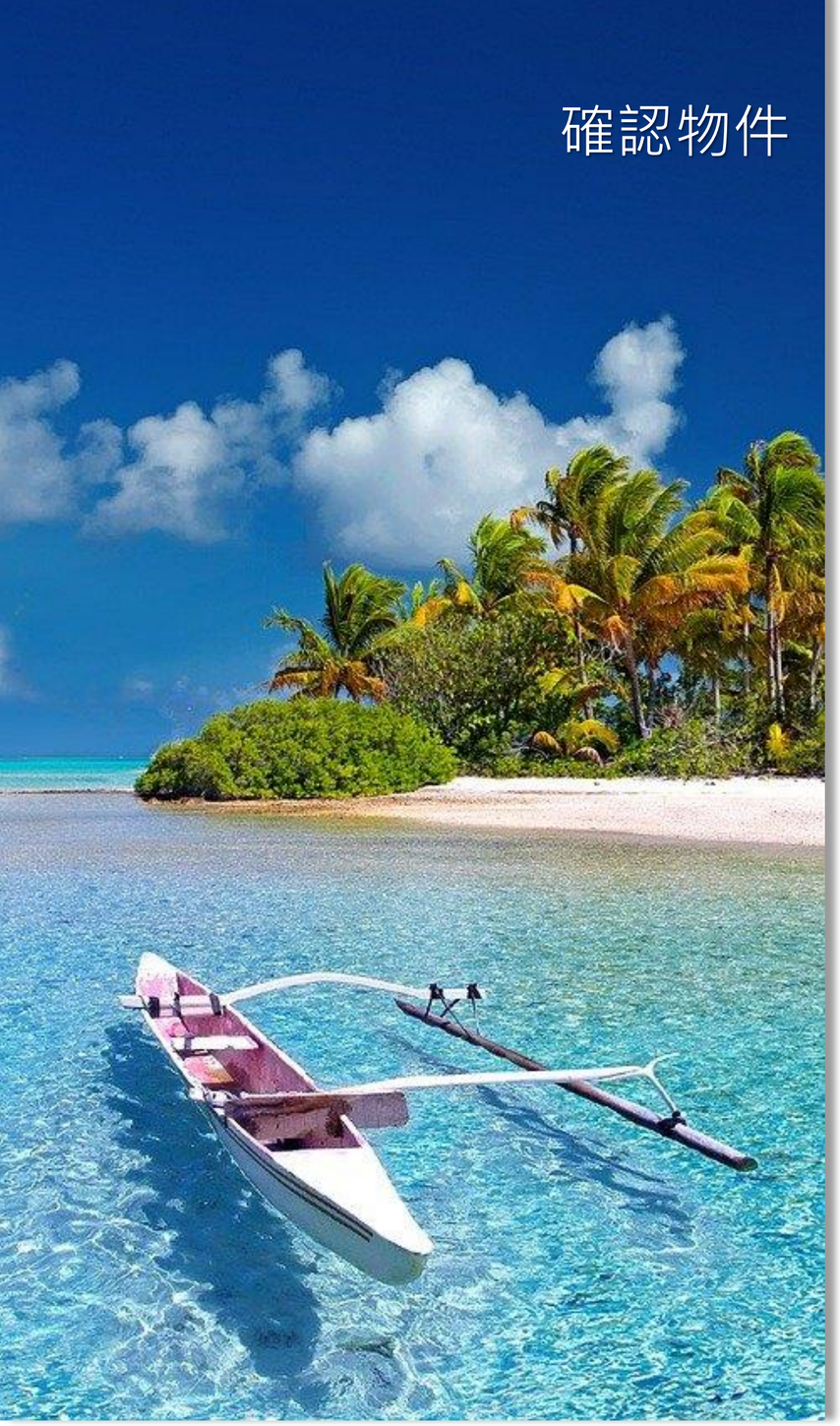
辨識物件

Duke's Choice Inc. sells clothing from their catalog. Business is growing 30 percent per year, and they need a new order entry system. You have been contracted by Duke's Choice to design the new system.

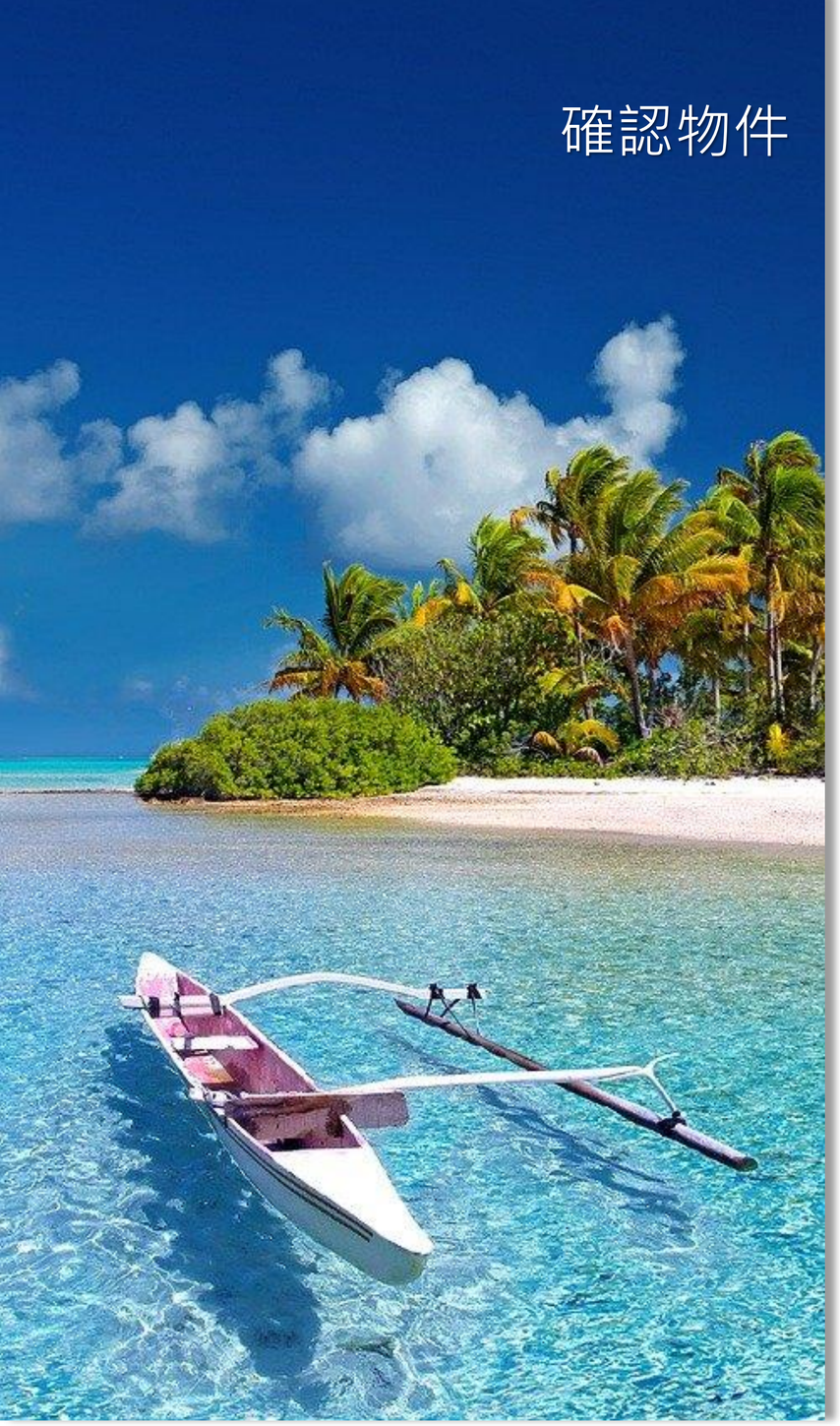
Duke's Choice produces an online catalog every three months and emails it to subscribers. Each shirt in the catalog has an item identifier (ID), one or more colors (each with a different color code), one or more sizes, a description, and a price.

Duke's Choice accepts checks and all major credit cards.

確認物件



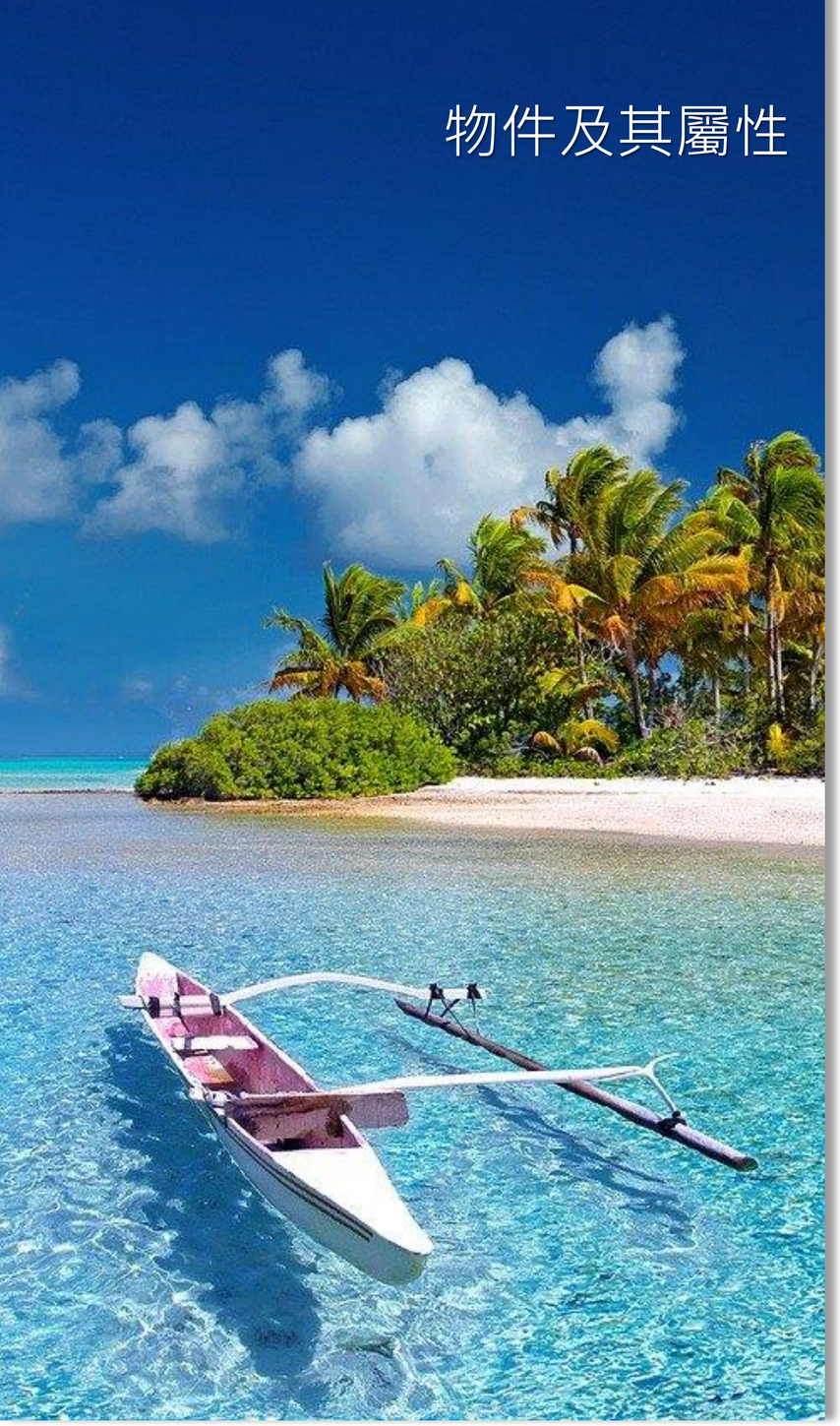
- 與問題領域的相關性
  - 物件是否存於問題領域邊界之內
  - 物件是否為解決問題所必需的
  - 物件是否為用戶和系統之間的相互作用的一部分
- 獨立存在性
  - 不具獨立存在性的名詞通常為屬性



確認物件

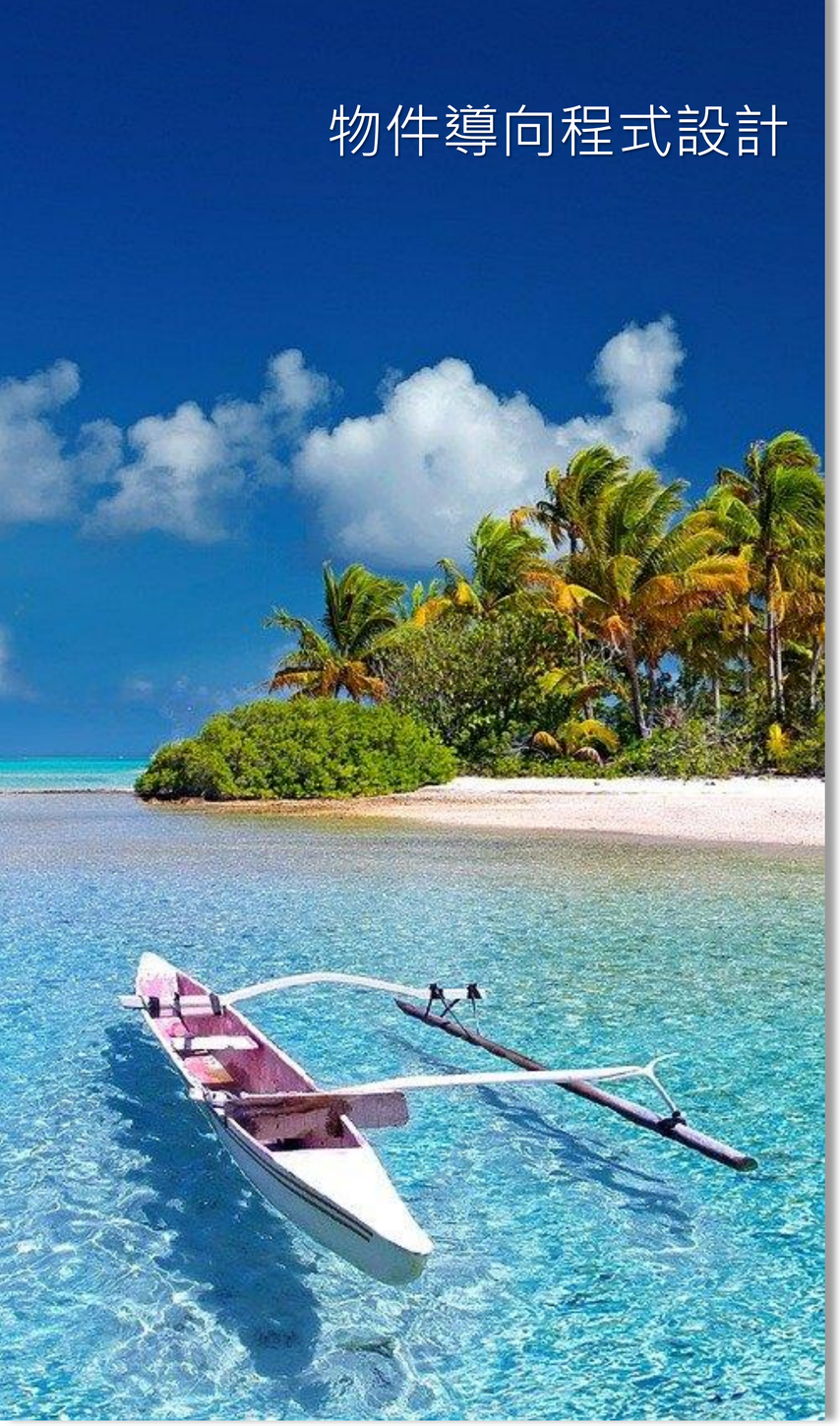
Order (form)	Shirt	Customer
Payment (form)	**Catalog**	**CSR**

物件及其屬性

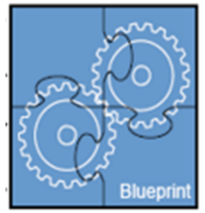


Customer	Order	Shirt
customer ID name address phone number email address *Order	order ID data *Shirt(s) total price *Form of payment *CSR status	shirt ID price description size color code
assign a customer ID	calculate order ID calculate the total price add shirt to order remove shirt from order submit the order	calculate shirt ID display shirt information

物件導向程式設計

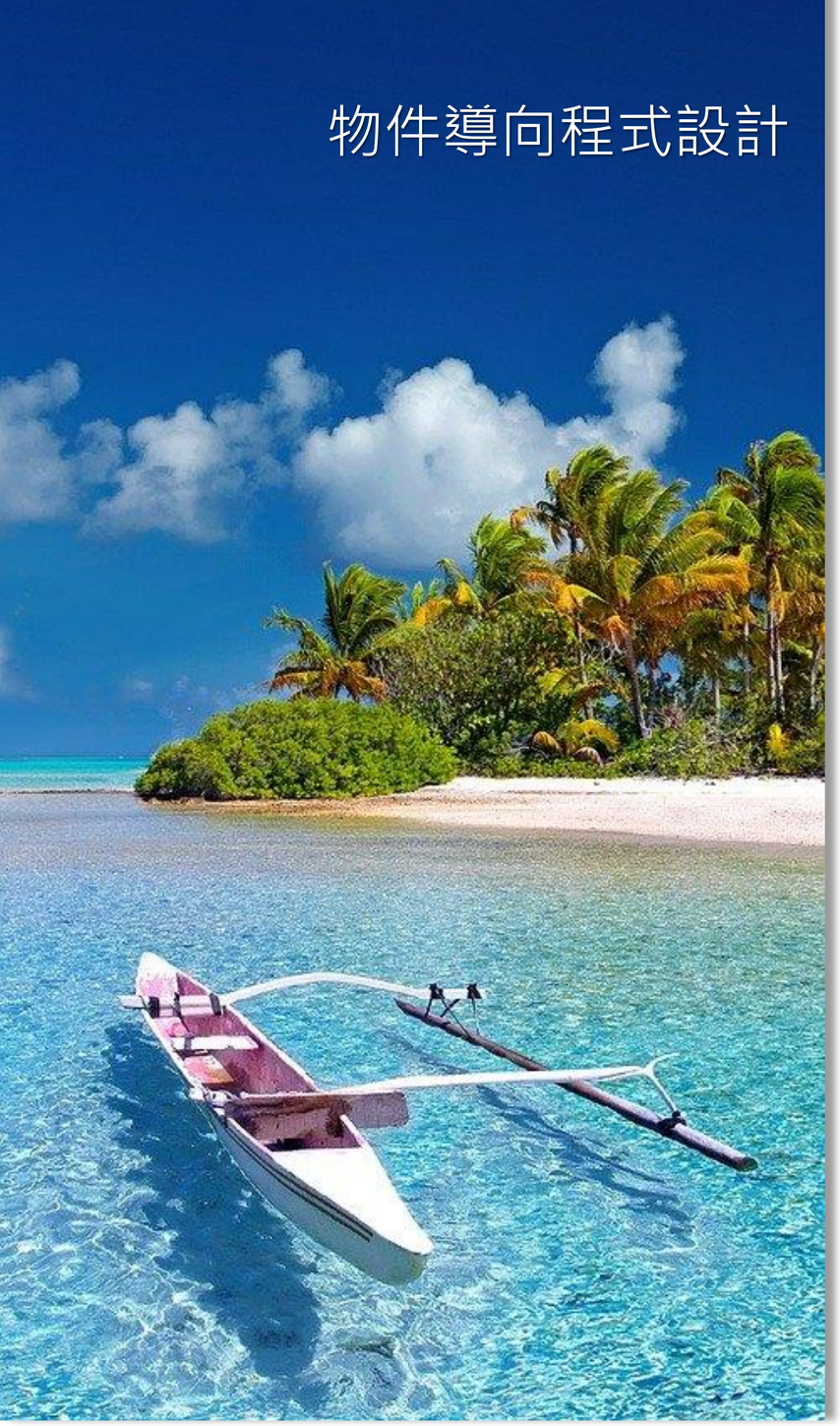


- 類別設計
  - 為物件設計類別 (藍圖)
  - 物件間溝通的介面
  - 定義屬性資料型態
  - 定義行為方法
- 重複使用類別元件
  - JDK 類別函式庫 Library
  - 其他廠商開發之 Java 元件
  - 自行開發的元件



Class	**Order**	**Shirt**	**Customer**
Attributes	order ID date *Shirt(s) total price *Form of payment *CSR status	shirt ID price description size color code	customer ID name address phone number email address *Order
Behaviors	calculate order ID calculate the total price add shirt to order remove shirt from order submit the order	calculate shirt ID display shirt information	assign a customer ID

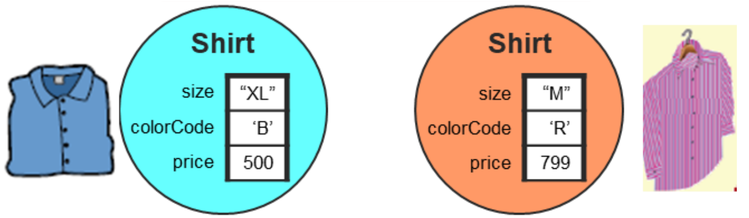
Class	**Form of Payment**	**Catalog**	**CSR**
Attributes	customer ID name address phone number email address *Order	*Shirt(s)	name extension
Behaviors	verify credit card number verify check payment	add a shirt remove a shirt	process order



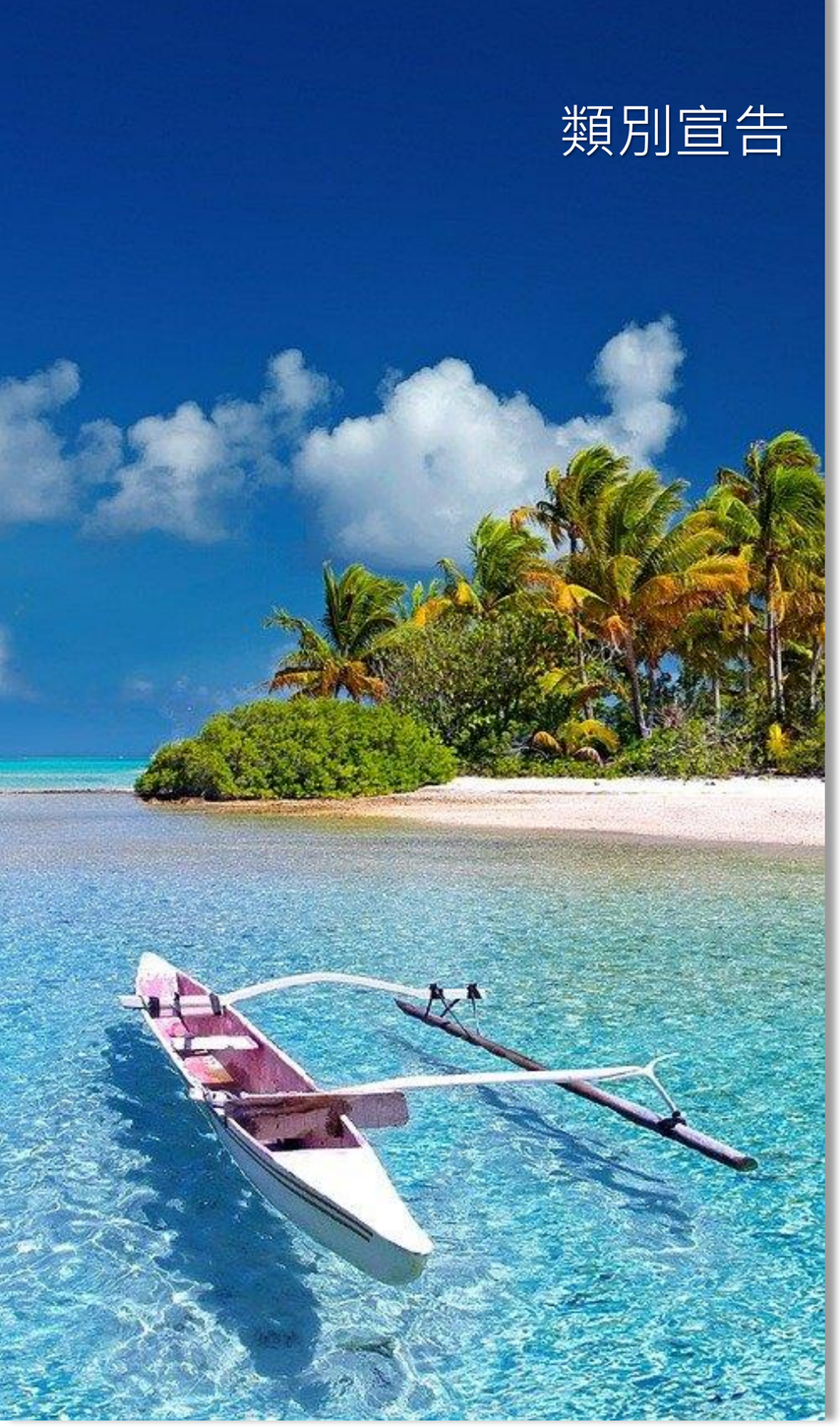
Modeling Class 類別塑模

- 視覺化所設計的類別
- UML 統一塑模語言
  - 類別名稱
  - 屬性 (名稱, 型態, 權限)
  - 建構 (參數列)
  - 方法 (名稱, 參數列, 傳回值, 權限)

Shirt
+shirtID: int +colorCode: char +size: String +price: double
+Shirt (color: char, price: double, description: String)
+calculateShirtID() : int +displayInformation()



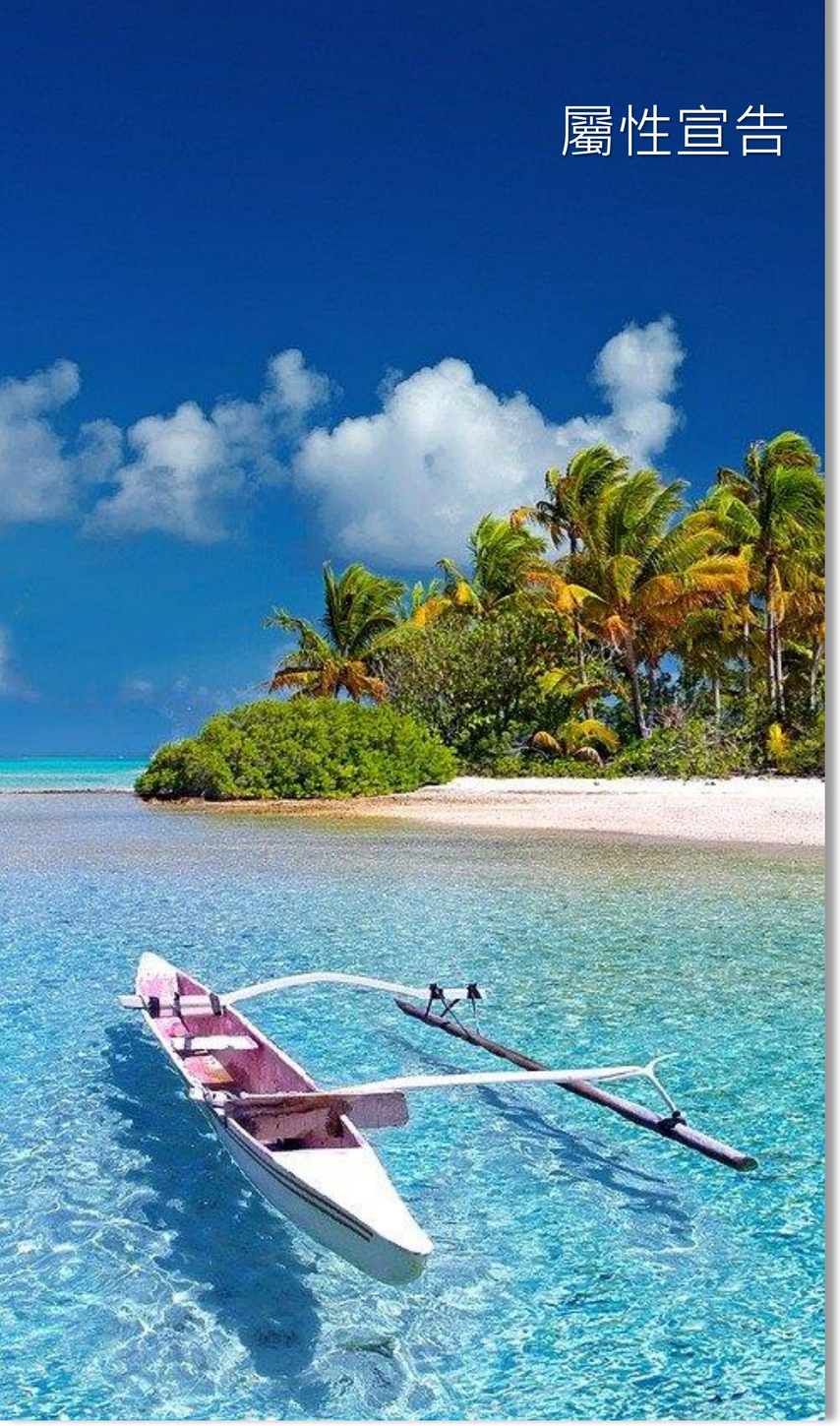
類別宣告



```
[modifiers] [modifiers] class <class_name> {
    ......
    ......
}
```

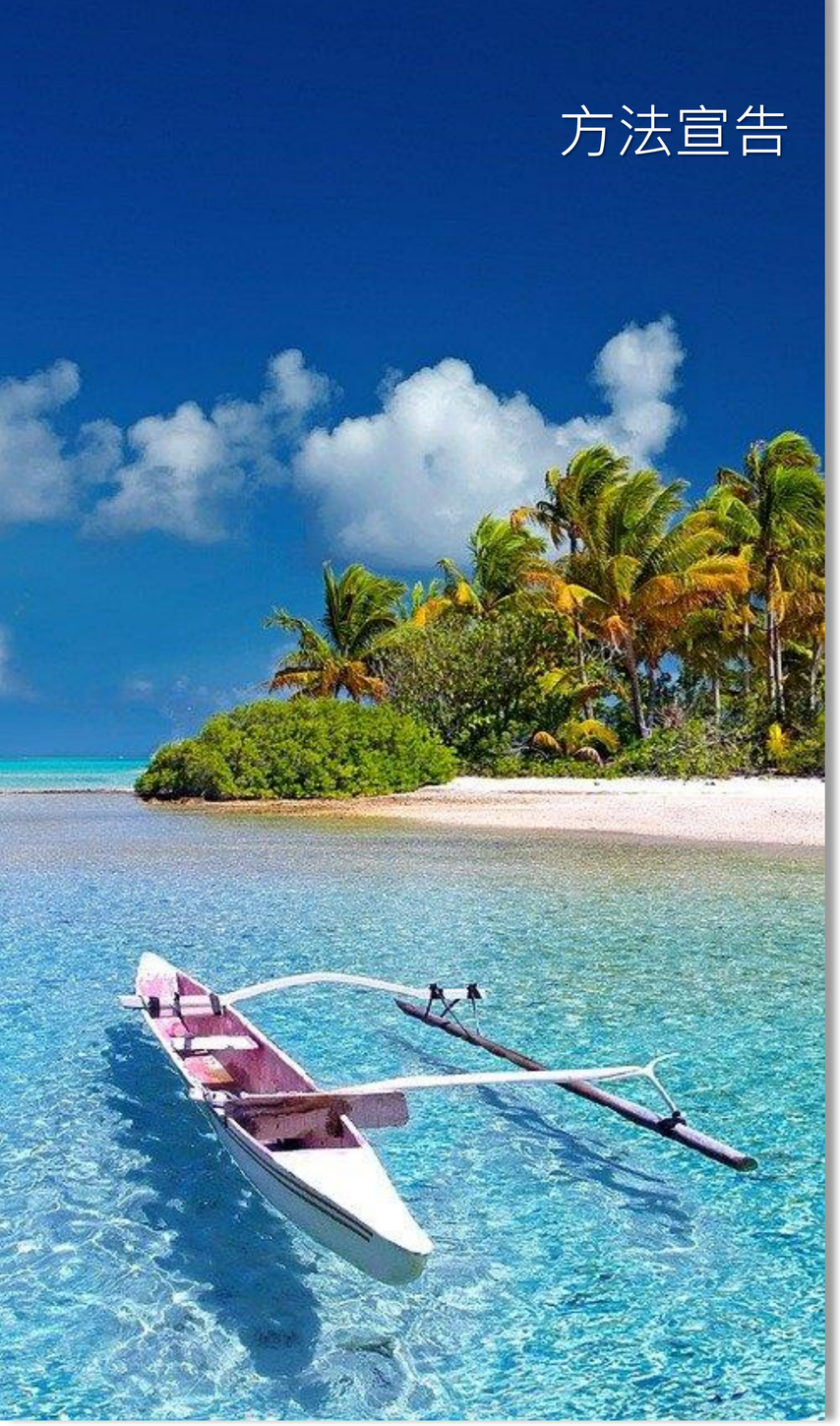
- Modifier 為 public 的類別，需存在 class_name.java 的檔案中
- 類別存取修飾字：**public** 或 不宣告（外部類別只能此二種，內部類別才可宣告為 **private** 或 **protected**）
- 修飾語：**static abstract** 或 **final**
- 類別名稱：若存取修飾字為**public**，則類別檔名必須與類別名稱相同
- 一個 Java 檔案，可包含一個以上的類別宣告
- 多個類別宣告在一個 Java 檔案中，只能有一個 public 的類別

屬性宣告



```
[modifiers] class <class_name> {
    [modifiers] <type> <name> [=initial_values];
}
```

- 程式中用變數來存放需使用到的資料
- 變數定義在 **class body** 內用來表示屬性
- 存取權限 **modifier** 有 **public, protected, (default), private**
- 物件變數：屬於物件各自獨立維護
- 類別變數：若將變數宣告為 **static**，則此變數屬於類別的，將由此類別實作出來的物件共同維護

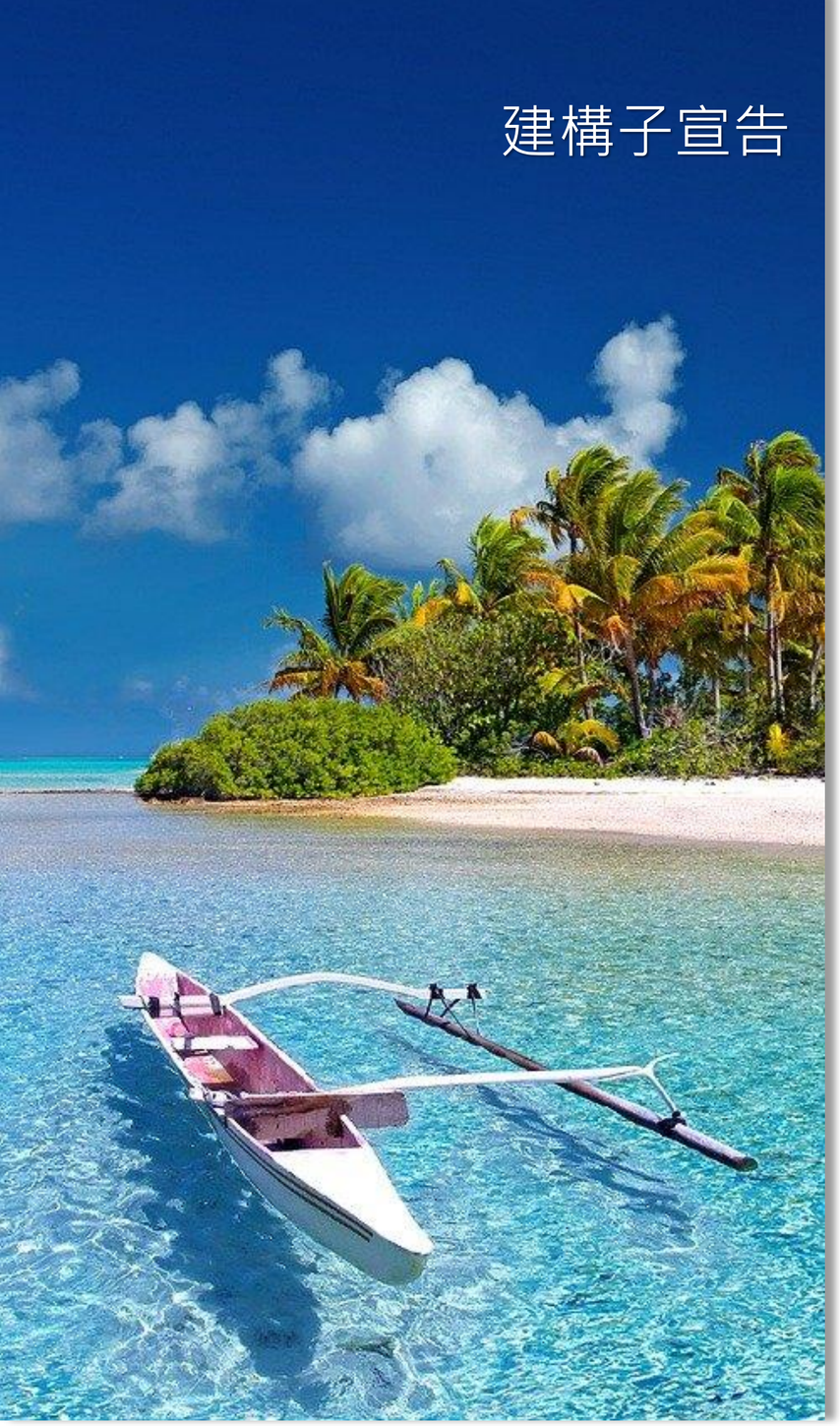


方法宣告

```
[modifiers] <return_type> <name> ([arguments]) {
        ......
        return return_type;
    }
```

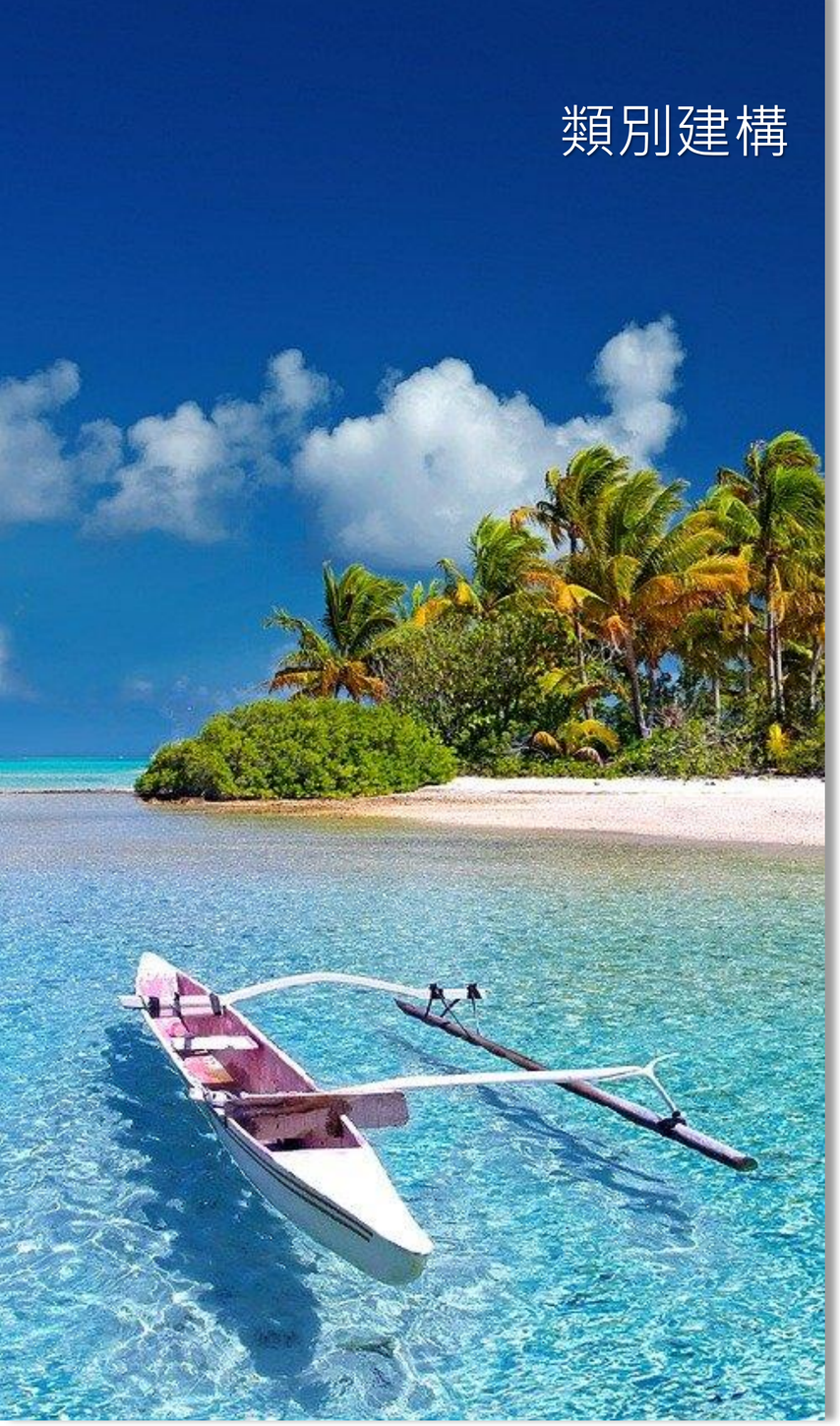
- 傳回值型態需與方法區段內 return 的資料型態相符
  - 方法如沒有傳回值，傳回值型態為 void
- 傳入參數列 (Arguments List)
  - 格式為 Type Name
  - 可有 0~N 個，超過一個，用 , 隔開

建構子宣告



```
[modifiers] class <class_name> {
    [modifiers] class_name ([arguments]) {
        ......
    }
}
```

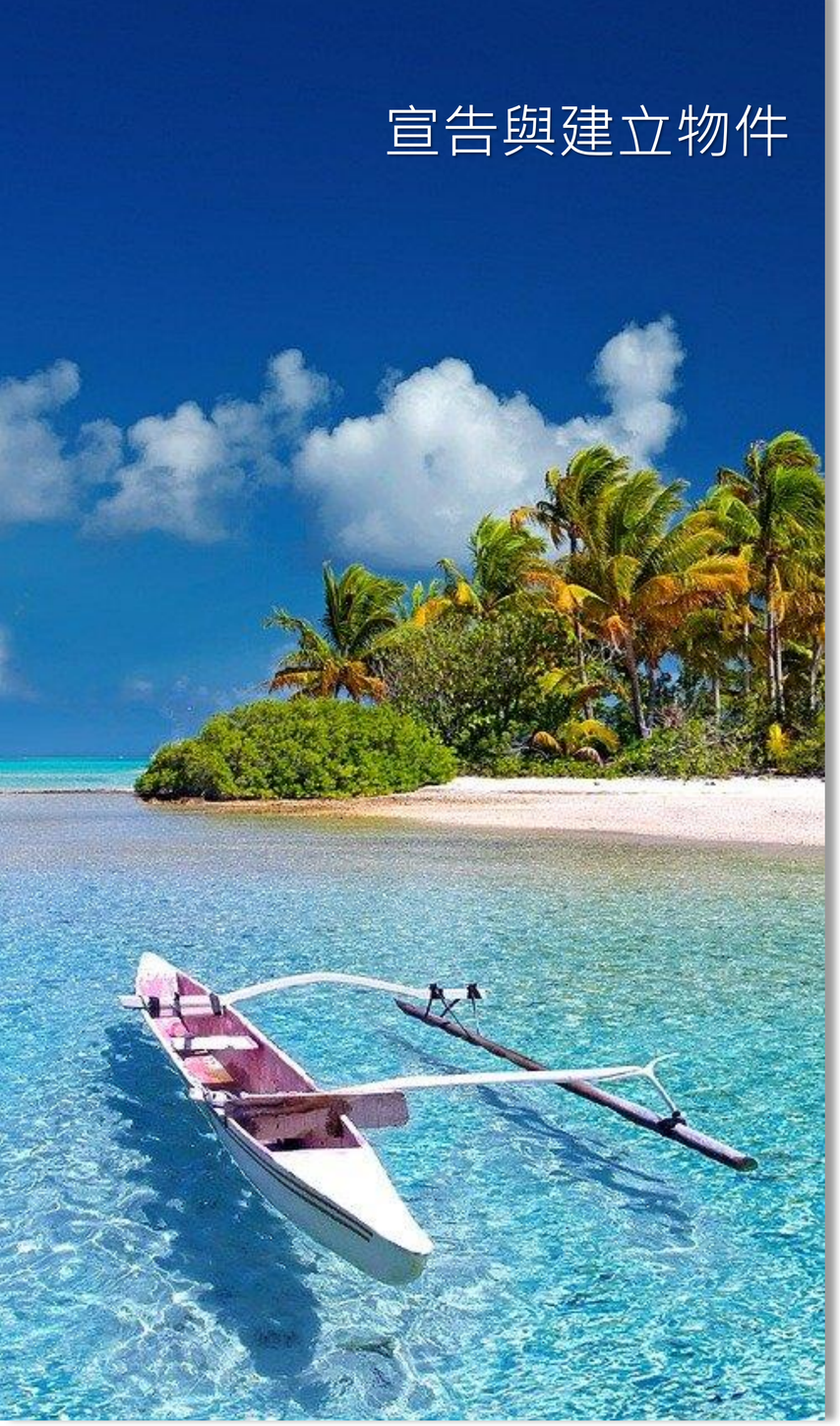
- 建構子名稱需與類別名稱相同，且存取權限預設與類別相同
- 建構子沒有回傳類型



類別建構

Shirt
+shirtID: int +colorCode: char +size: String +price: double +description : String
+Shirt (c: char, s: String, p: double, d: String)
+setPrice(p : double) +getPrice ( ) : double +displayInformation ( )

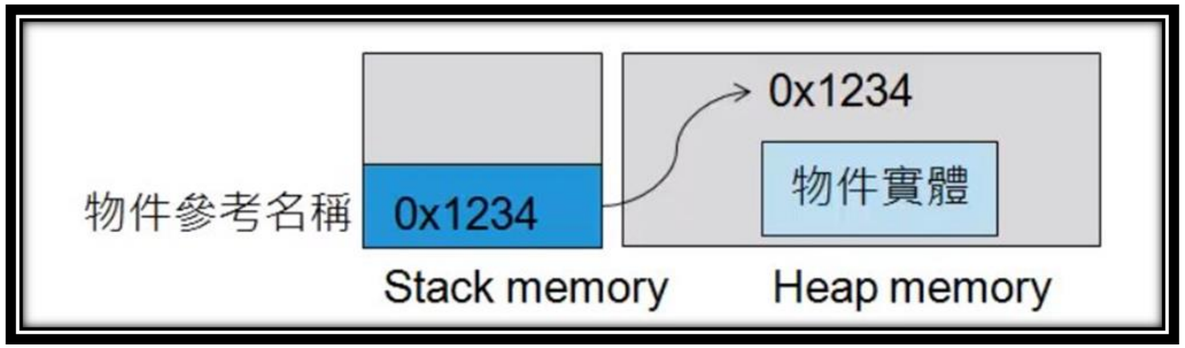
```
01  public class Shirt {
02
03      public int shirtID = 0;                          物件屬性
04      public char colorCode = 'G';                     (欄位)
05      public String size = "XL" ;
06      public double price = 299.00;
07      public String description = "Polo Shirt";
08
09      public Shirt(char c, String s,
10                   double p, String d) {
11          colorCode = c;
12          size = s;
13          price = p;
14          description = d;
15      }                                                建構子
16
17      public void setPrice(double p) {                 物件方法
18          price = p;                                   (操作)
19      }
20      public double getPrice( ) {
21          return price;
22      }
23      public void displayInformation() {
24          System.out.println("Shirt ID:" + shirtID);
25          System.out.println("Color:" + colorCode);
26          System.out.println("Size:" + size);
27          System.out.println("Price:" + price);
28      }
29
30  }
```



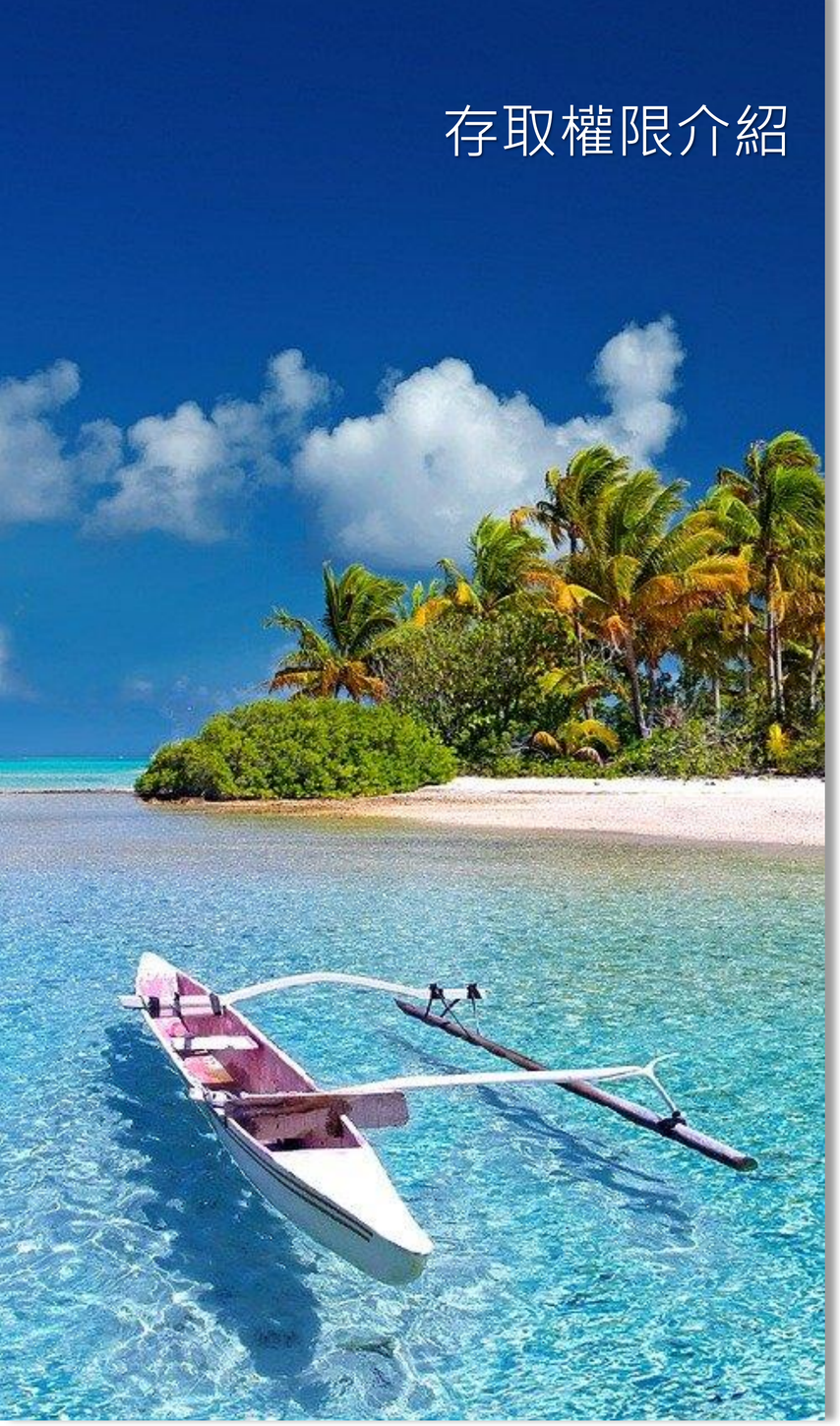
宣告與建立物件

- 語法一
- 類別名稱　物件參考名稱；
- 物件參考名稱 = new 類別名稱（ ）；

- 語法二
- 類別名稱　物件參考名稱 = new　類別名稱（ ）；



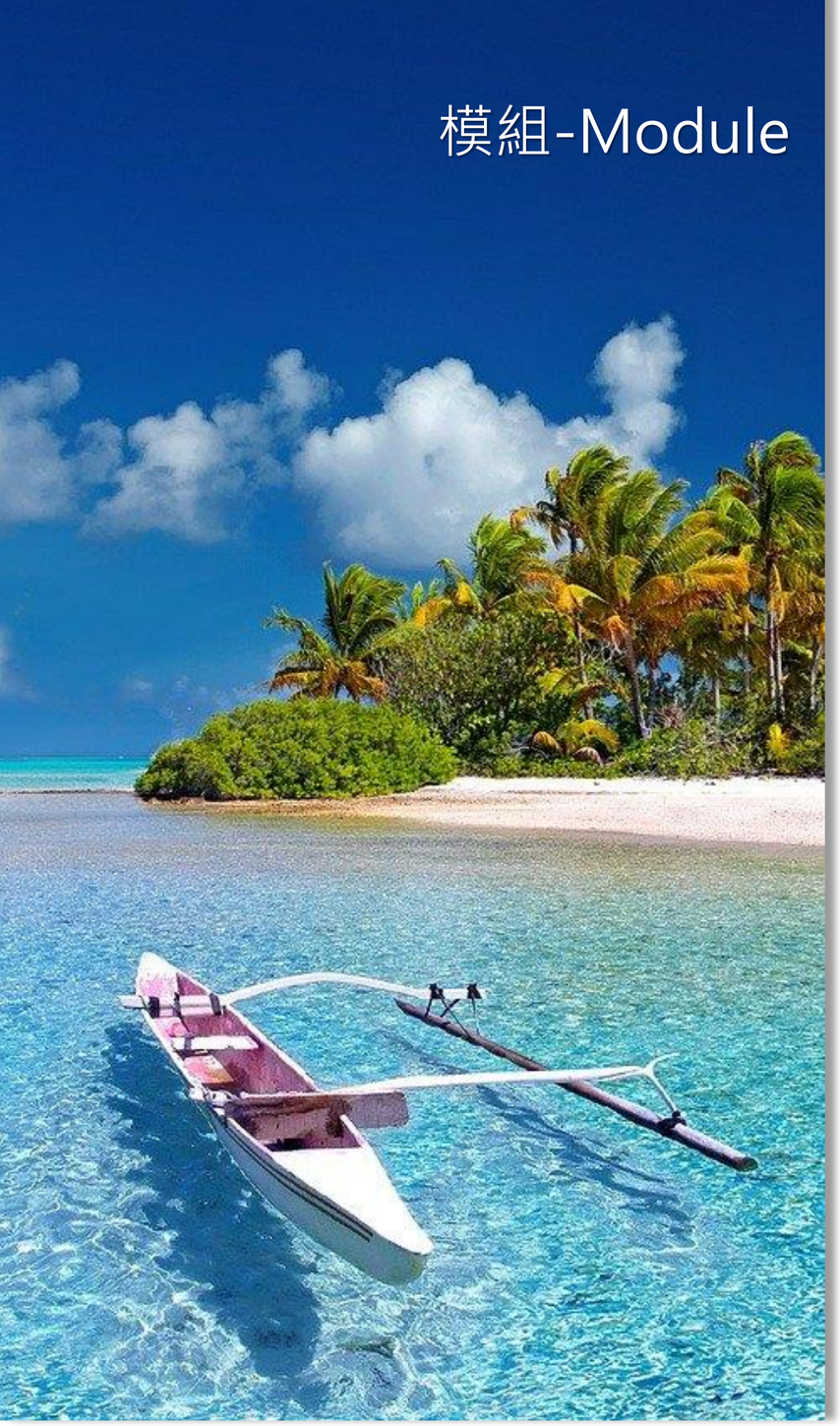
存取權限介紹



存取修飾字	同一類別（A）	同一套件（B）	子類別(C)	全域(D)
private	O	X	X	X
default	O	O	X	X
protected	O	O	O	X
public	O	O	O	O

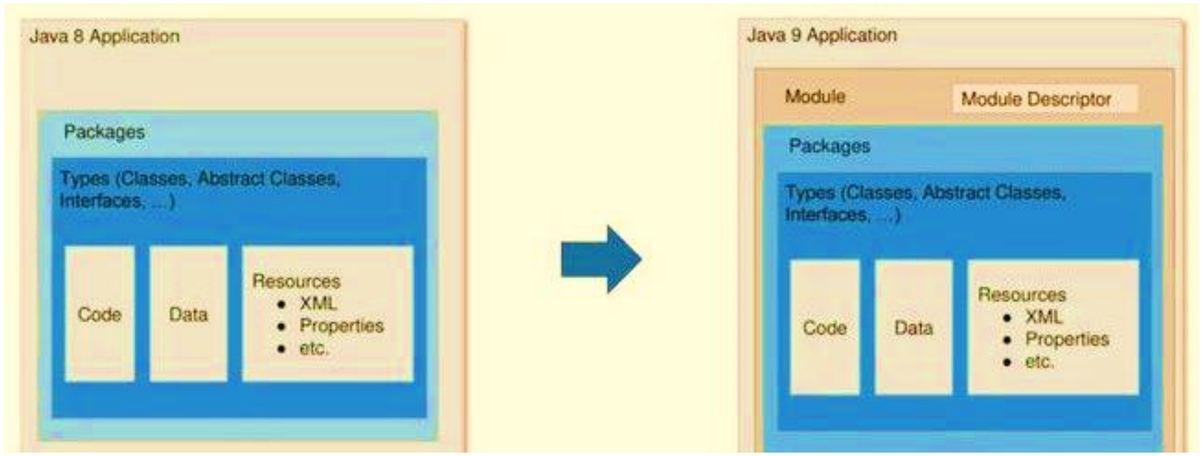
package 1		package 2	
A.class(父)	B.class	C.class(子)	D.class
private int w=1;	~~w~~	~~w~~	~~w~~
int x=2	x	~~x~~	~~x~~
protected y=3	y	y	~~y~~
public z=4	z	z	z

模組 Module



模組-Module

- JDK 8之前 問題
  - JAVA 運行環境的膨脹和過於肥胖,每次Jvm運行時需加載rj.jar,不管有無運行到全部會加載上去
  - 每一個public類別都可以被其它類別訪問到,這樣會導致無意中使用了並不想被公開的 api
- JDK 9之後 模組化解決問題
  - 所以模組就是用來管理各個package,通過設定package的顯示與隱藏,設定誰可存取,所以模組使得程式更安全
  - 可看成一個電商系統,可區分為 支付模組 購物車模組 會員模組 商品模組.....
- java –list-modules



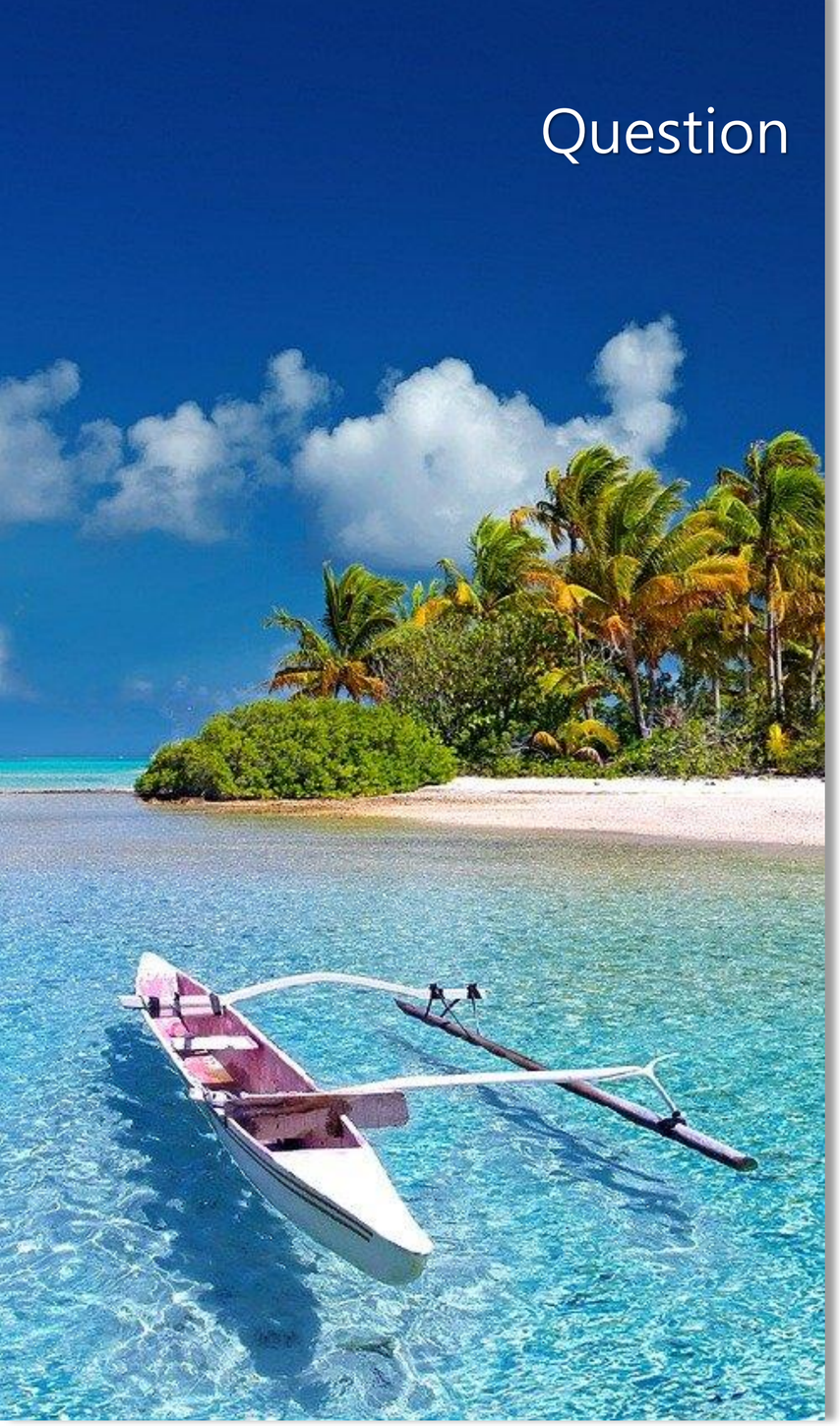
Accessibility (JDK 1 – JDK 8)

- public
- protected
- package
- private

Accessibility (JDK 9)

- *public to everyone*
- *public but only to friend modules*
- *public only within a module*
- protected
- package
- private

Question



Which two modules include APIs in the Java SE Specification?

(Choose two.)

A. java.logging

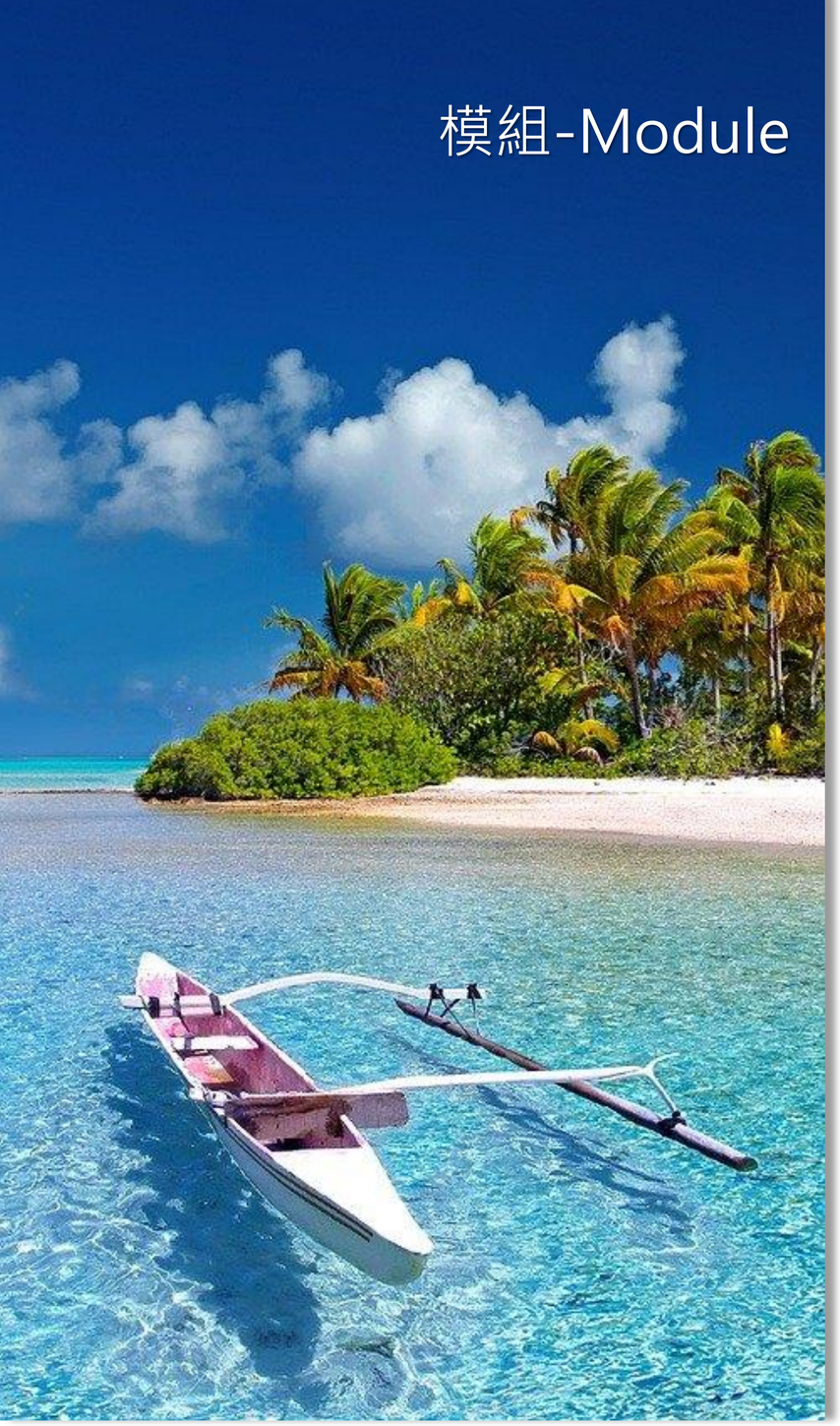
B. java.desktop

C. javafx

D. jdk.httpserver

E. jdk.jartool

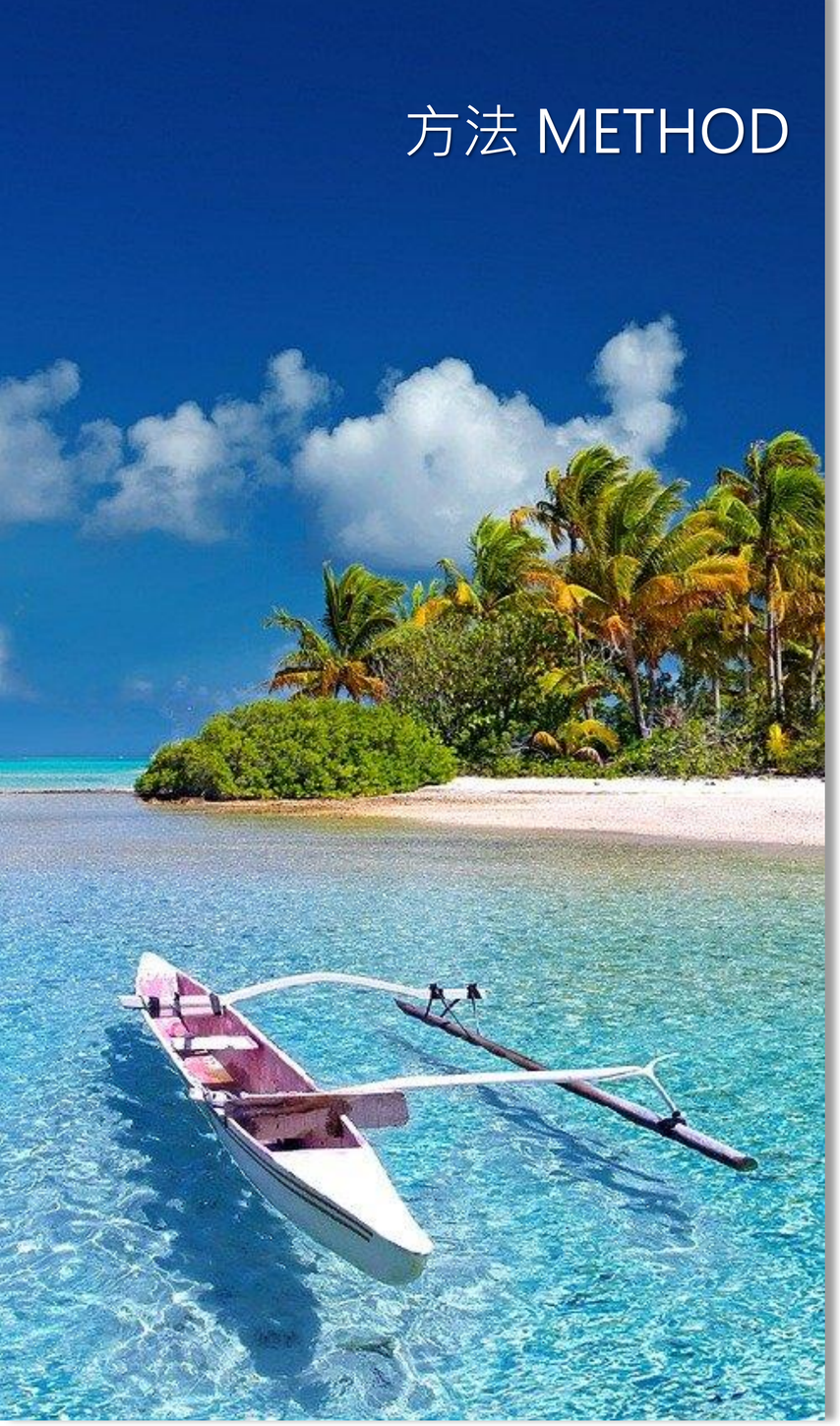
模組-Module



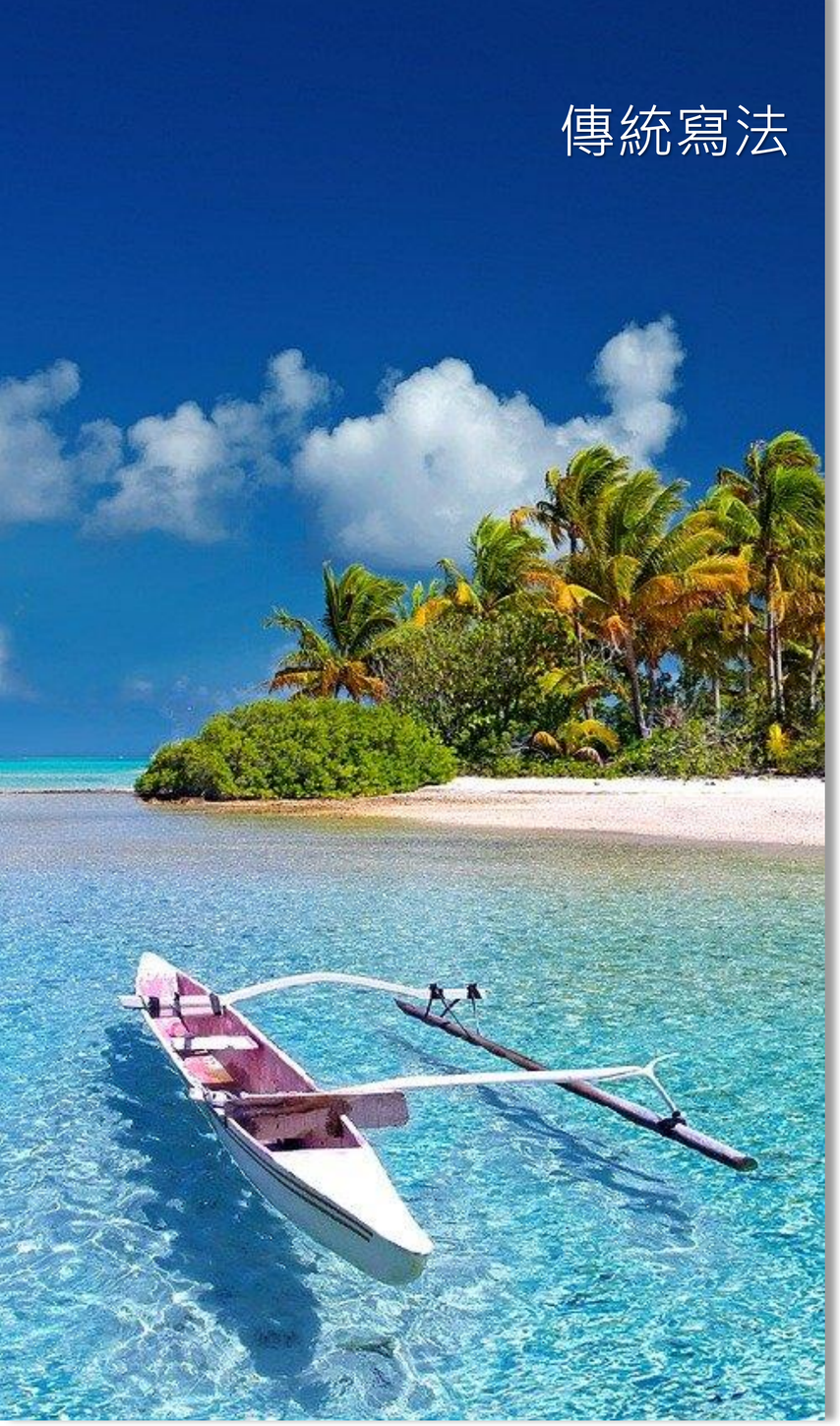
- 一個模組必須提供一個模組描述器 — 描述模組相依性、該模組能讓其他模組使用的套件等中介資料。模組描述器是一個經過編譯的模組宣告，定義於 module-info.java 檔案中，每個模組宣告從關鍵字 module 開始，緊接著一個獨一無二的模組名稱，以及括在大括弧內的模組內容：
- module modulename {
  - requires 模組名稱
  - exports 套件名稱 to模組名稱
- }
- 模組宣告的主體可以是空的，或是包含各種模組指令 (directive)，像是 requires、 exports、 provides...with、 uses 及 opens ，編譯模組宣告會建立模組描述器，存放在模組根目錄的 module-info.class 檔案中，

方法 METHOD

方法 METHOD



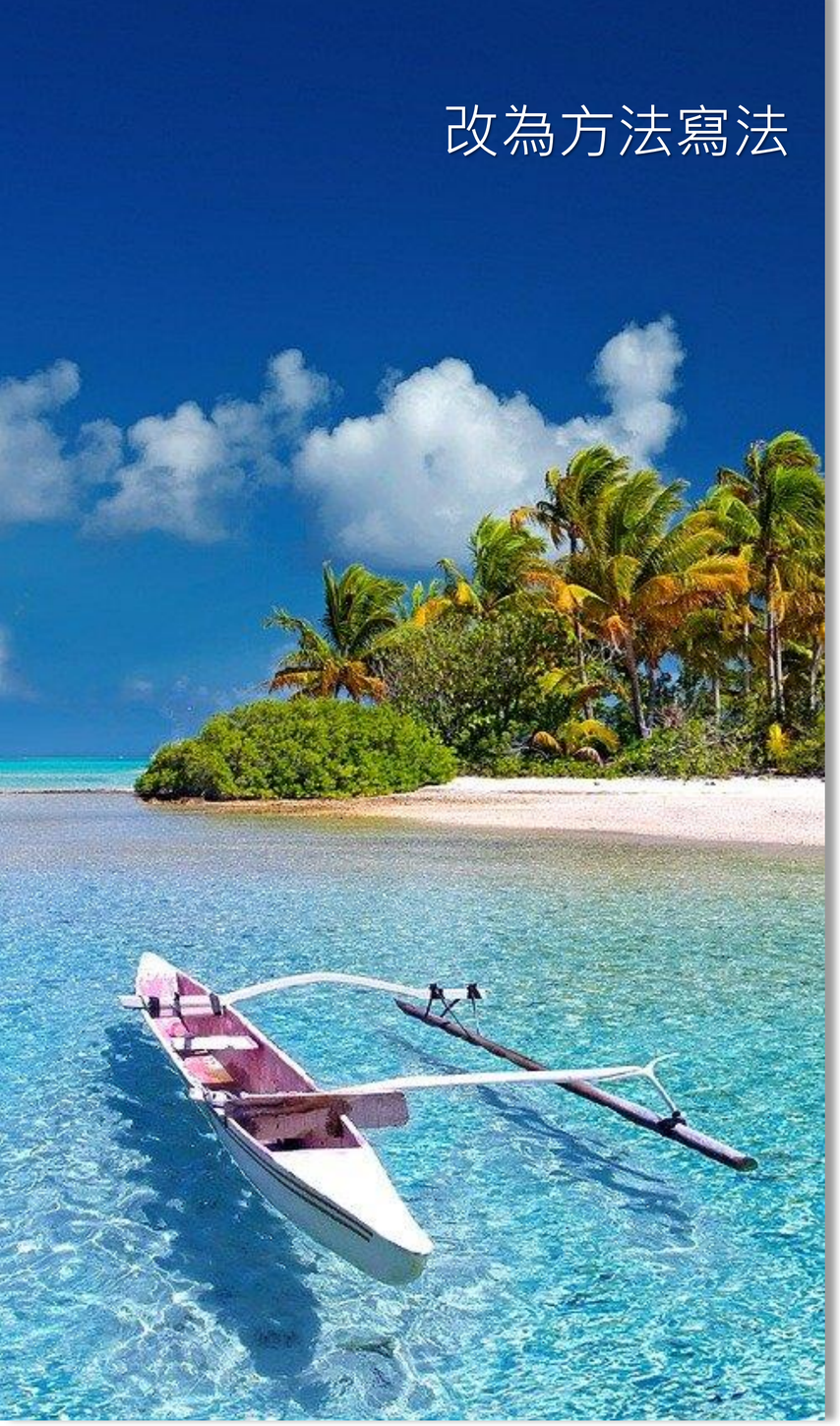
- 將程式中需重複執行的程式敘述獨立出來成為一個方法，在程式執行時可重複呼叫，可以精簡程式撰寫的長度也可以使程式具結構化
- 將具有特定功能的程式片段寫在獨立的方法中，可以方便程式的維護與除錯，也可提供其他程式呼叫執行



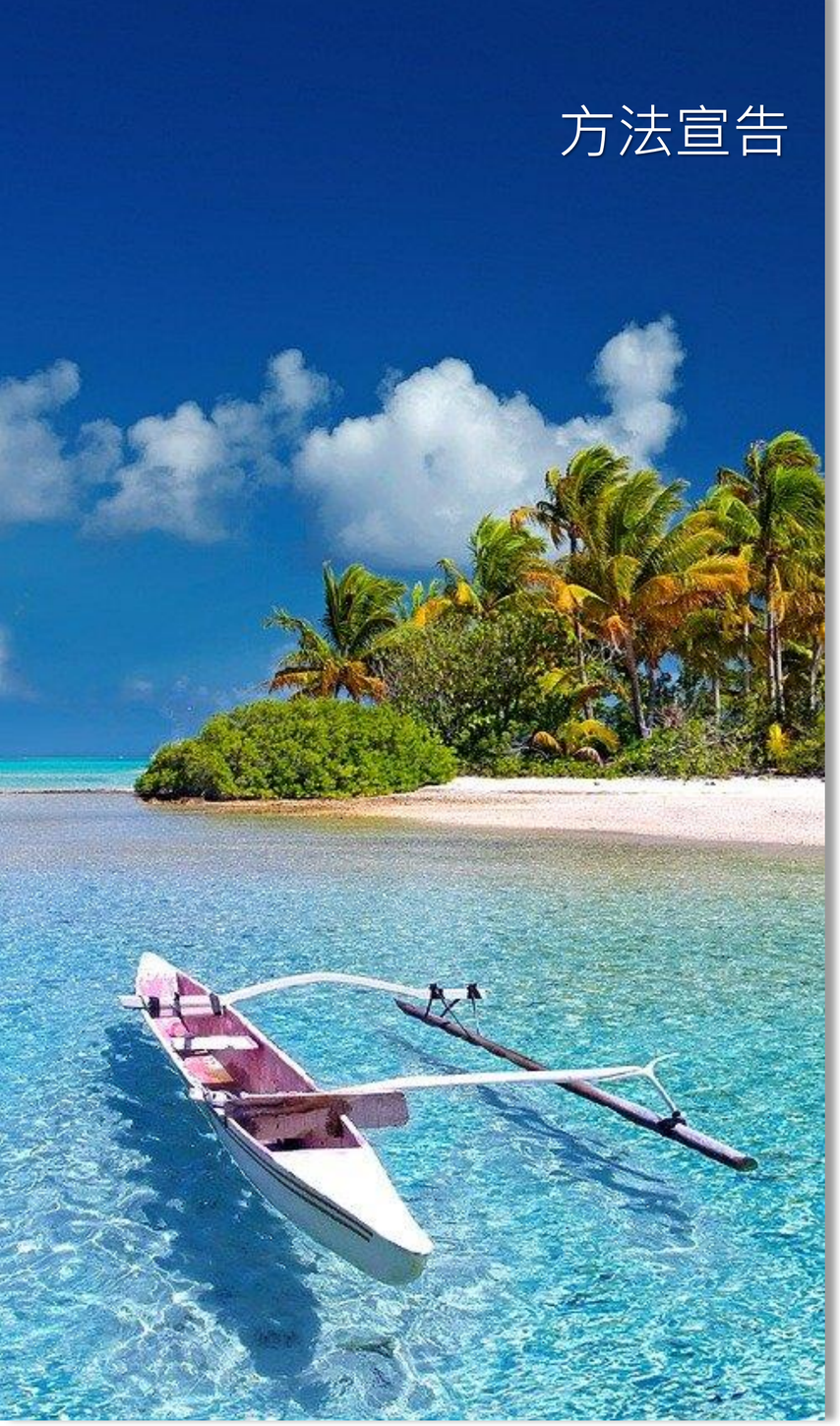
傳統寫法

```
int sum = 0;
for (int m = 1; m <= 100; m++) {
   sum += m;
}
System.out.println("1+2+3+....+100=" + sum);
//=============================================
sum = 0;
for (int m = 1; m <= 200; m++) {
   sum += m;
}
System.out.println("1+2+3+....+200=" + sum);
//=============================================
sum = 0;
for (int m = 1; m <= 300; m++) {
   sum += m;
}
System.out.println("1+2+3+....+300=" + sum);
```

改為方法寫法



```
System.out.println("1+2+3+....+100="+cal(100));

System.out.println("1+2+3+....+200="+cal(200));

System.out.println("1+2+3+....+300="+cal(300));


public static int cal(int max)   {

      int sum=0;

      for(int m=1;m<=max;m++)

         sum+=m;

      return sum;

   }
```



方法宣告

方法是由回傳值型別、方法名稱、參數與方法本體所組成

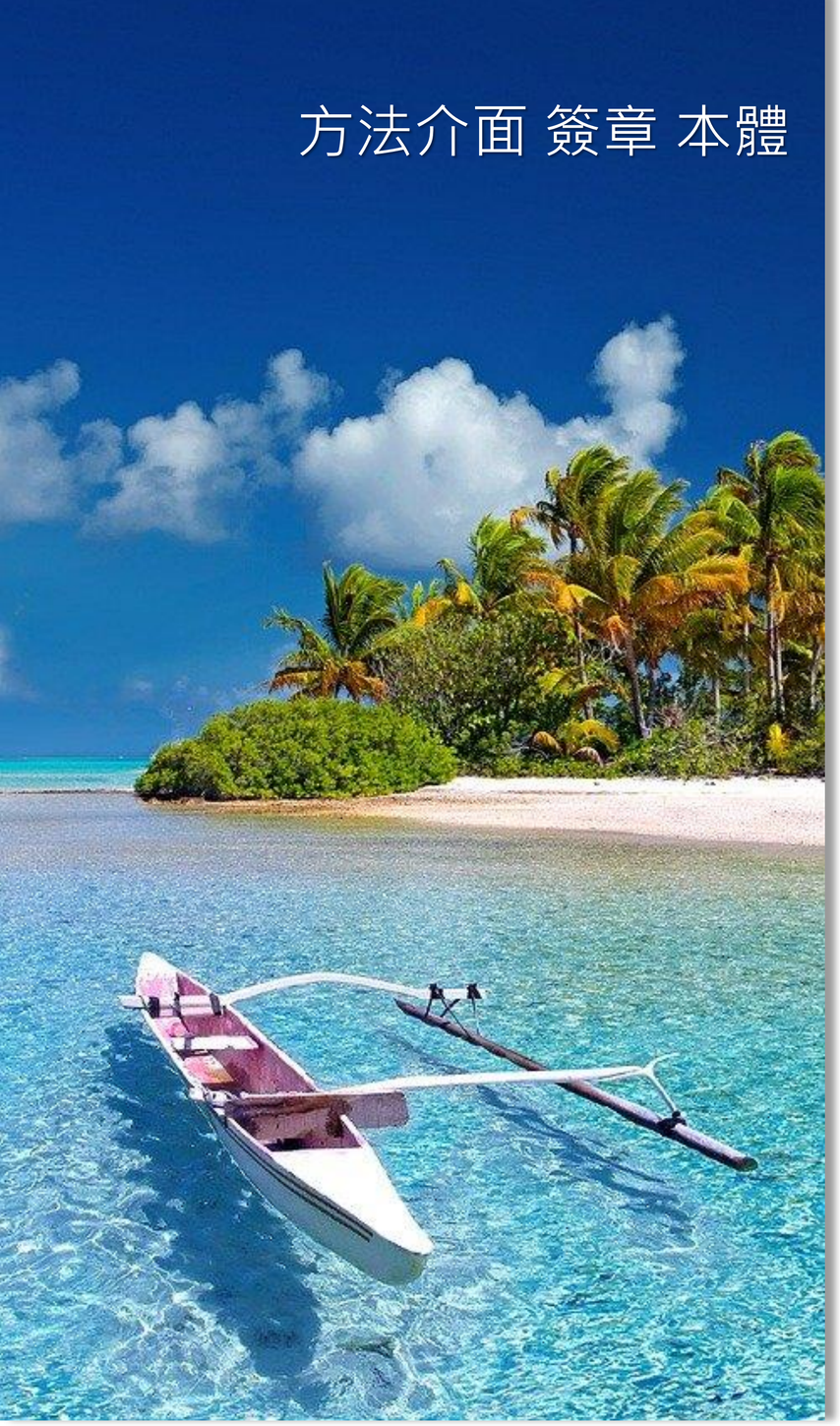
語法

```
[存取修飾字] [修飾語] 回傳值型別 方法名稱(參數列){
   //程式區段
   [return 運算式;]
}
```

```
01 public static int cal(int max)
02 {
03     int sum = 0;
04     for (int i=1; i<=max; i++)
05     {
06         sum+=i;
07     }
08     return sum;
09 }
```

- 傳回值型態需與方法區段內 return 的資料型態相符
  - 方法如沒有傳回值，傳回值型態為 void
- 傳入參數列 (Arguments List)
  - 格式為 Type Name
  - 可有 0~N 個，超過一個，用，隔開

方法介面 簽章 本體



- 方法的常見名詞
  - 方法介面
  - 方法本體
  - 方法簽章

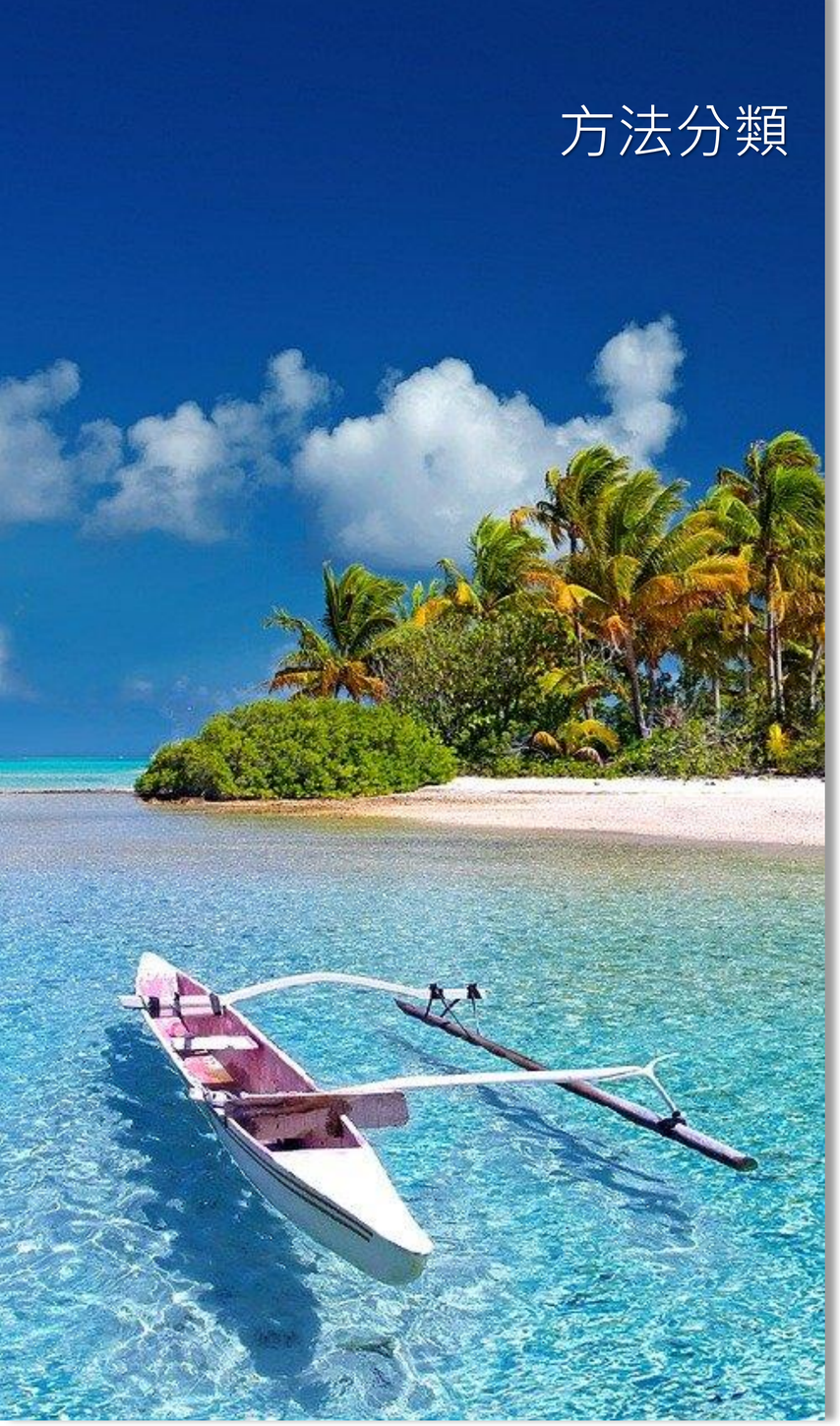
方法簽章

```
public void setTarget(double newTarget)
{
        target = newTarget;
        surplus = amount  -  target;
}
```

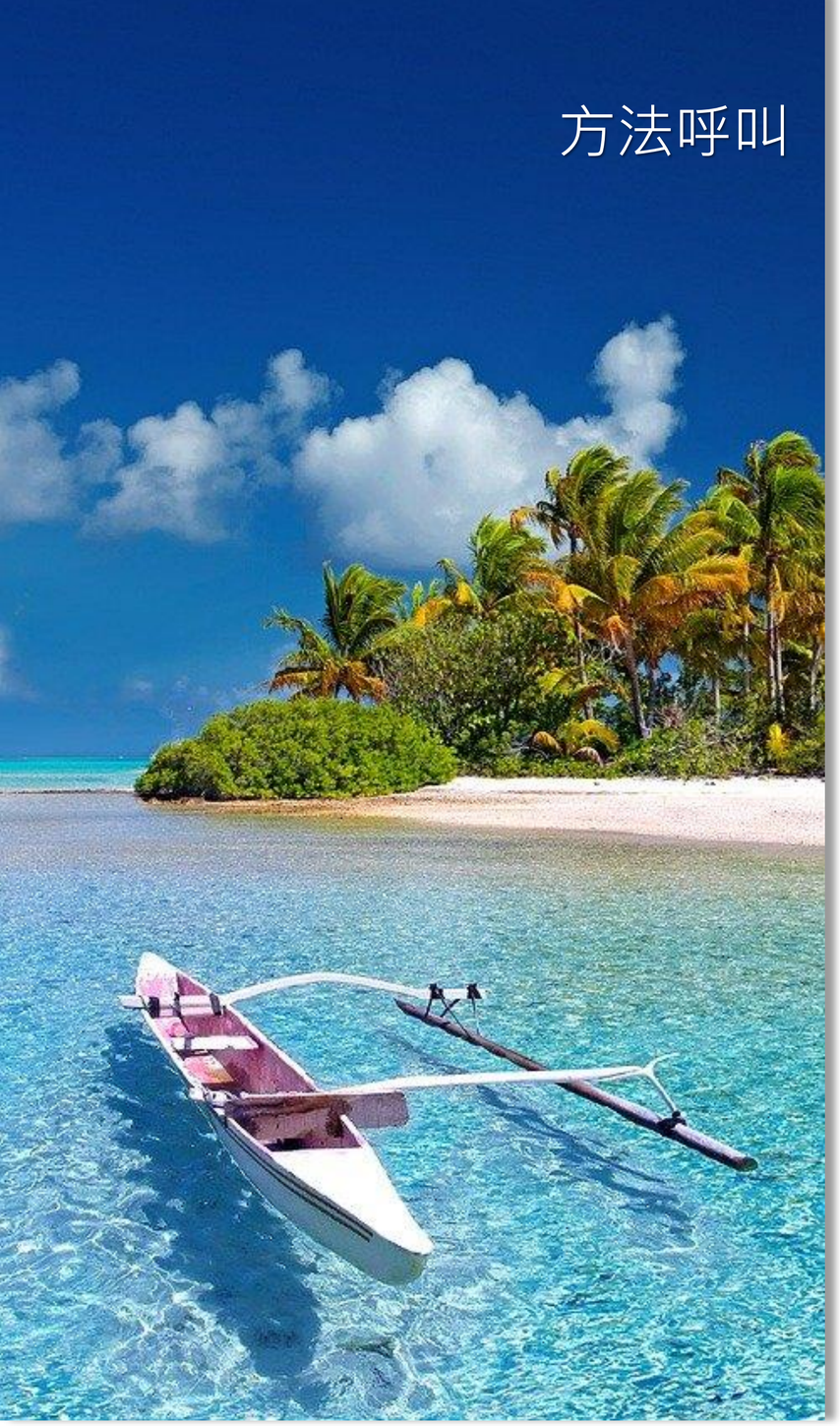
方法介面

方法本體

方法分類

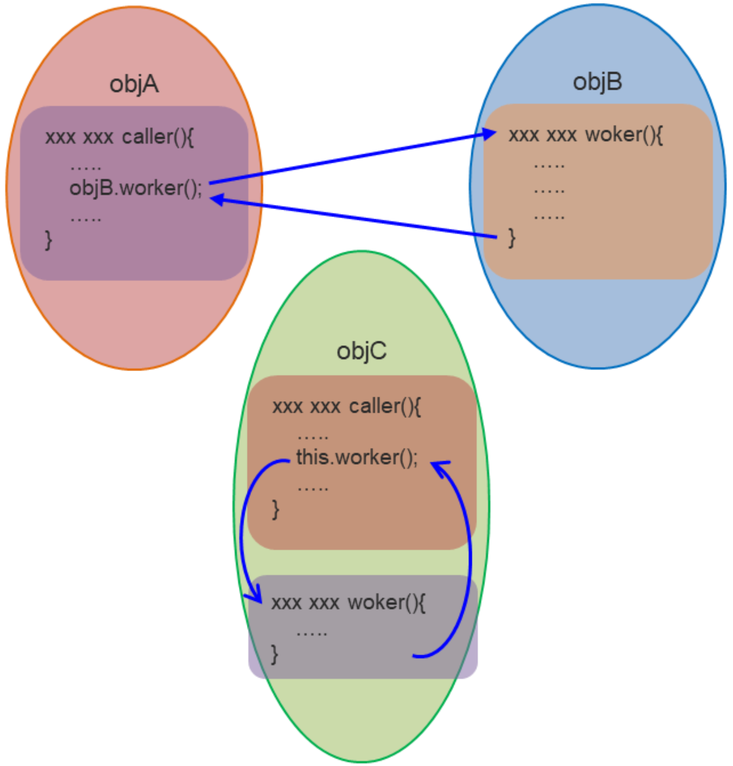


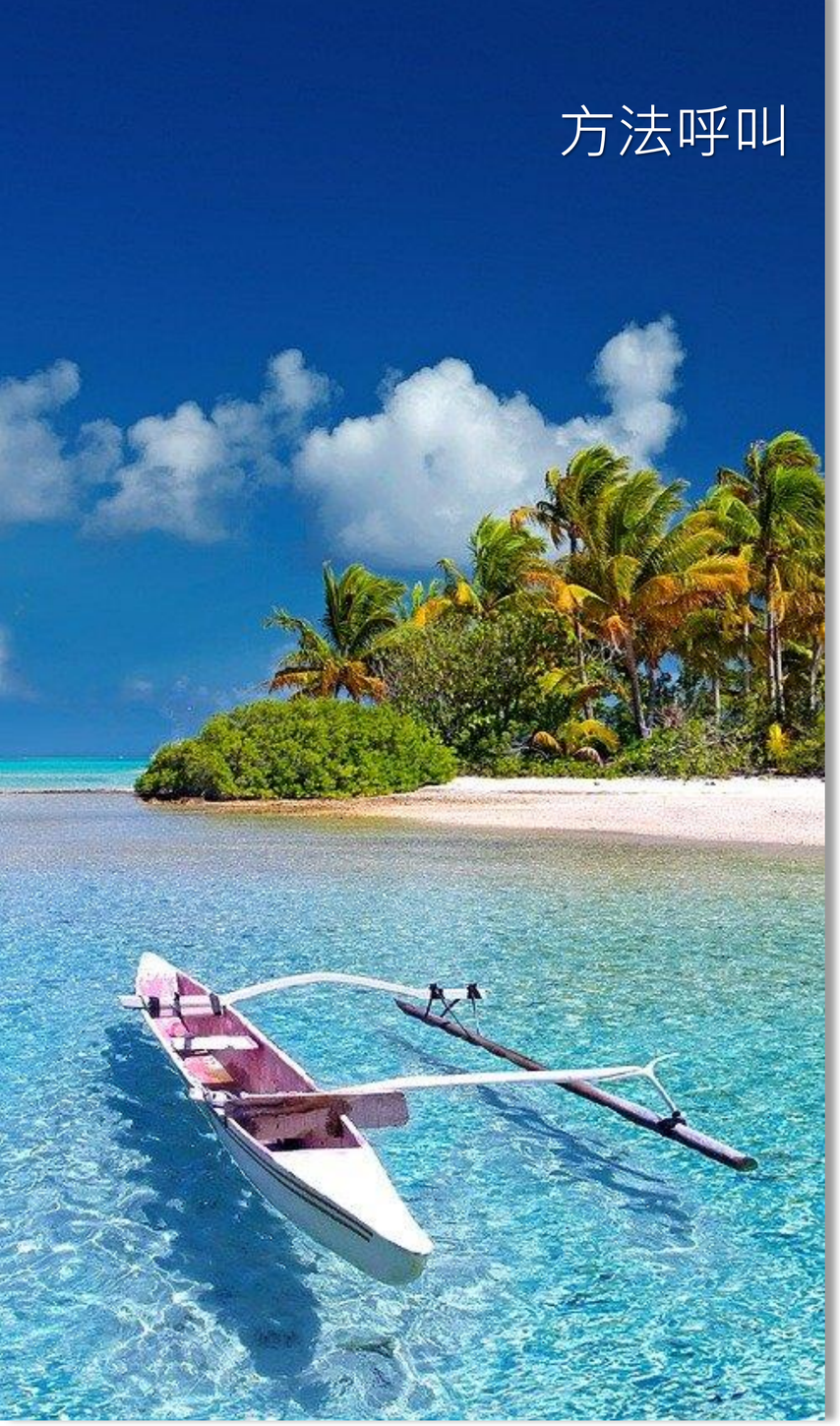
static	有加static為類別方法	沒加static為物件方法
	無回傳值	有回傳值
無形式參數	void 方法名( ){ }	返回值資料型別 方法名( ){ return }
有形式參數	void 方法名(形參列表){ }	返回值資料型別 方法名(形參列表){return}



方法呼叫

- 方法呼叫
  - 類別方法(static)
    - 可以直接另一類別方法
      - 類別方法();
    - 物件方法需實作物作才可呼叫
      - 物件參考.方法( );
  - 物件方法
    - 可以直接呼叫類別方法或物件方法
    - 呼叫物件自身的其他方法
      - this.方法( );
      - this 表示物件自身參考，可省略





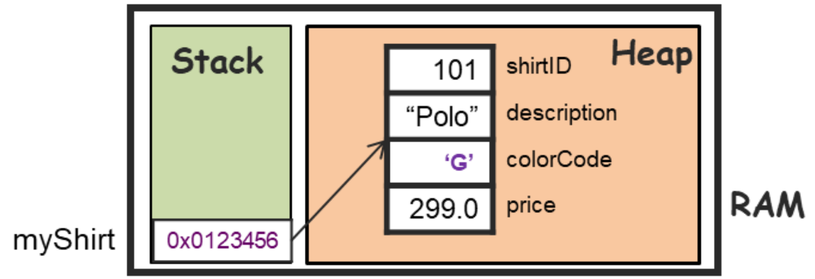
方法呼叫

物件參考.物件方法

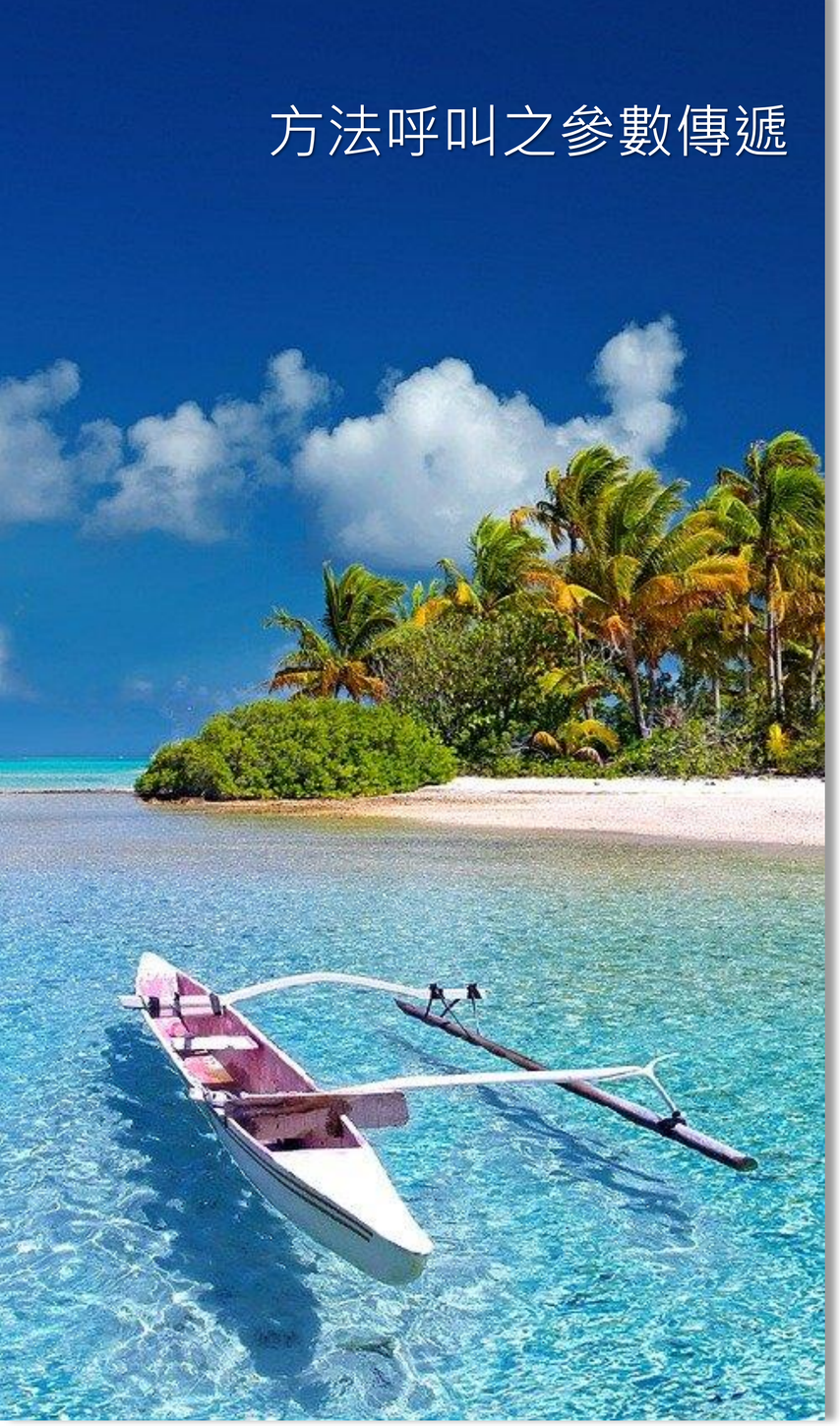
```
public class Shirt {
  public int shirtID = 101;
  public String description = "Polo Shirt";
  public char colorCode = 'R';
  public double price = 299.0;

public void displayInformation( ) {
    System.out.println("Shirt ID: " + shirtID);
    System.out.println("Description: " + description);
    System.out.println("Color Code: " + colorCode);
    System.out.println("Shirt Price: " + price);
  }
}
```

```
public class TestShirt {

  public static void main(String[ ] args) {
      Shirt myShirt = new Shirt( );
      myShirt.colorCode = 'G';
      myShirt.displayInformation();

  }
}
```

在螢幕上印
Shirt ID: 101
Description: Polo
Color Code: G
Shirt Price: 299.0

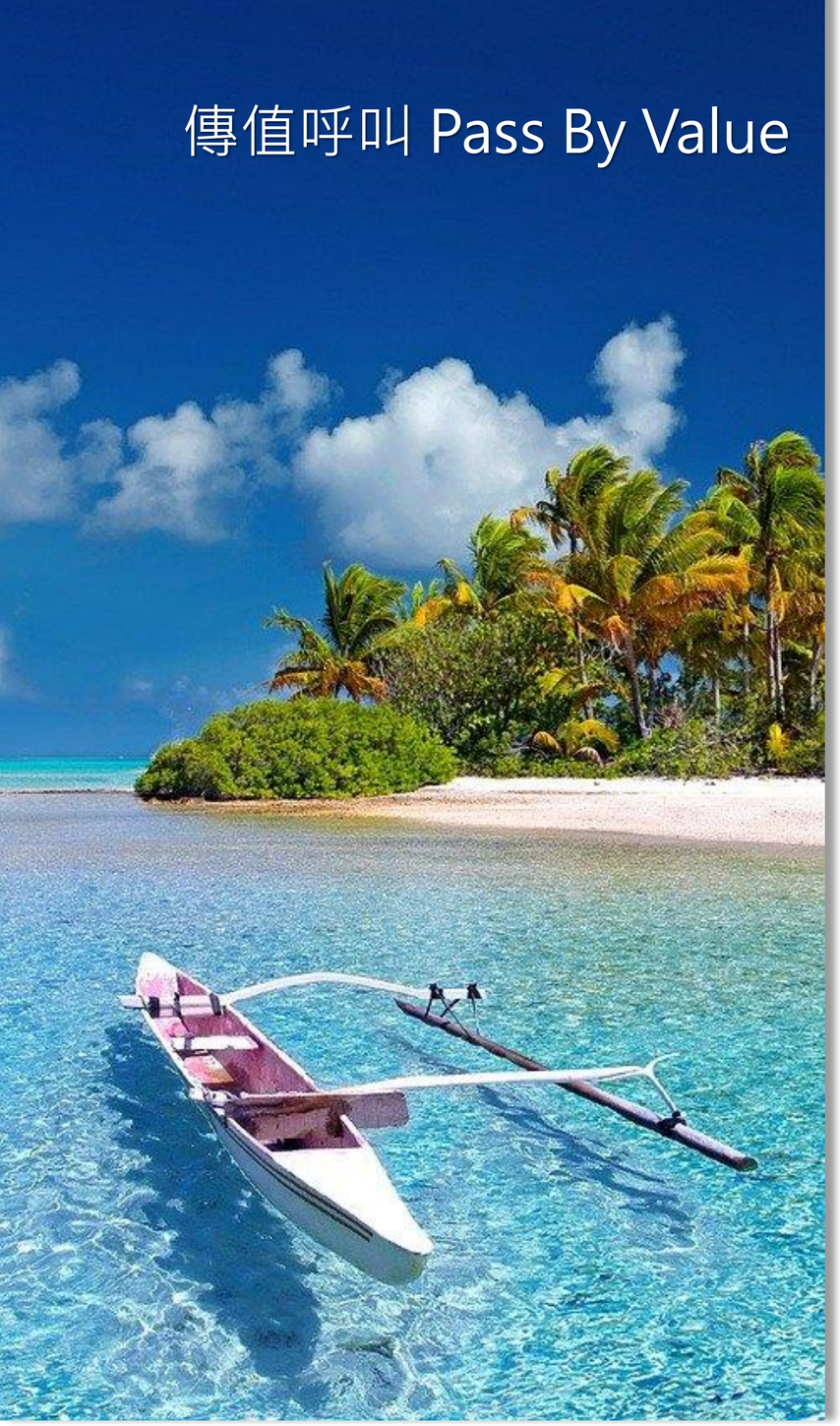


方法呼叫之參數傳遞



- 方法呼叫參數
  - Actual Argument (實際引數)：方法呼叫時呼叫端引數
  - Formal Parameter (形式參數 / 虛引數)：被呼叫端宣告的參數
- 方法呼叫參數傳遞方式
  - 傳值呼叫：實引數與虛引數使用不同記憶體空間，各自獨立，不互相影響
  - 傳址呼叫：實引數與虛引數使用相同的記憶體空間，使得彼此互相影響

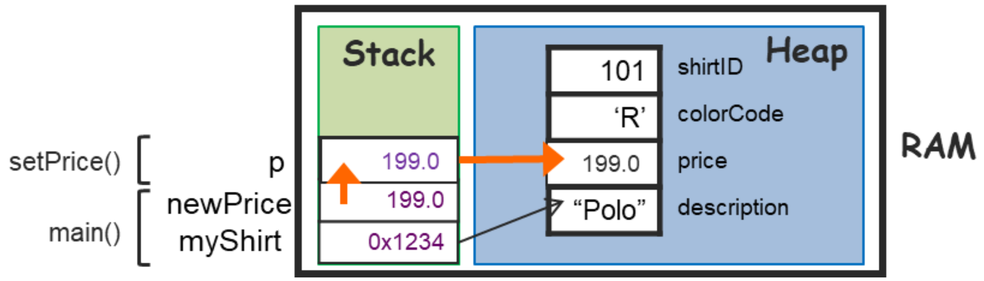
傳值呼叫 Pass By Value



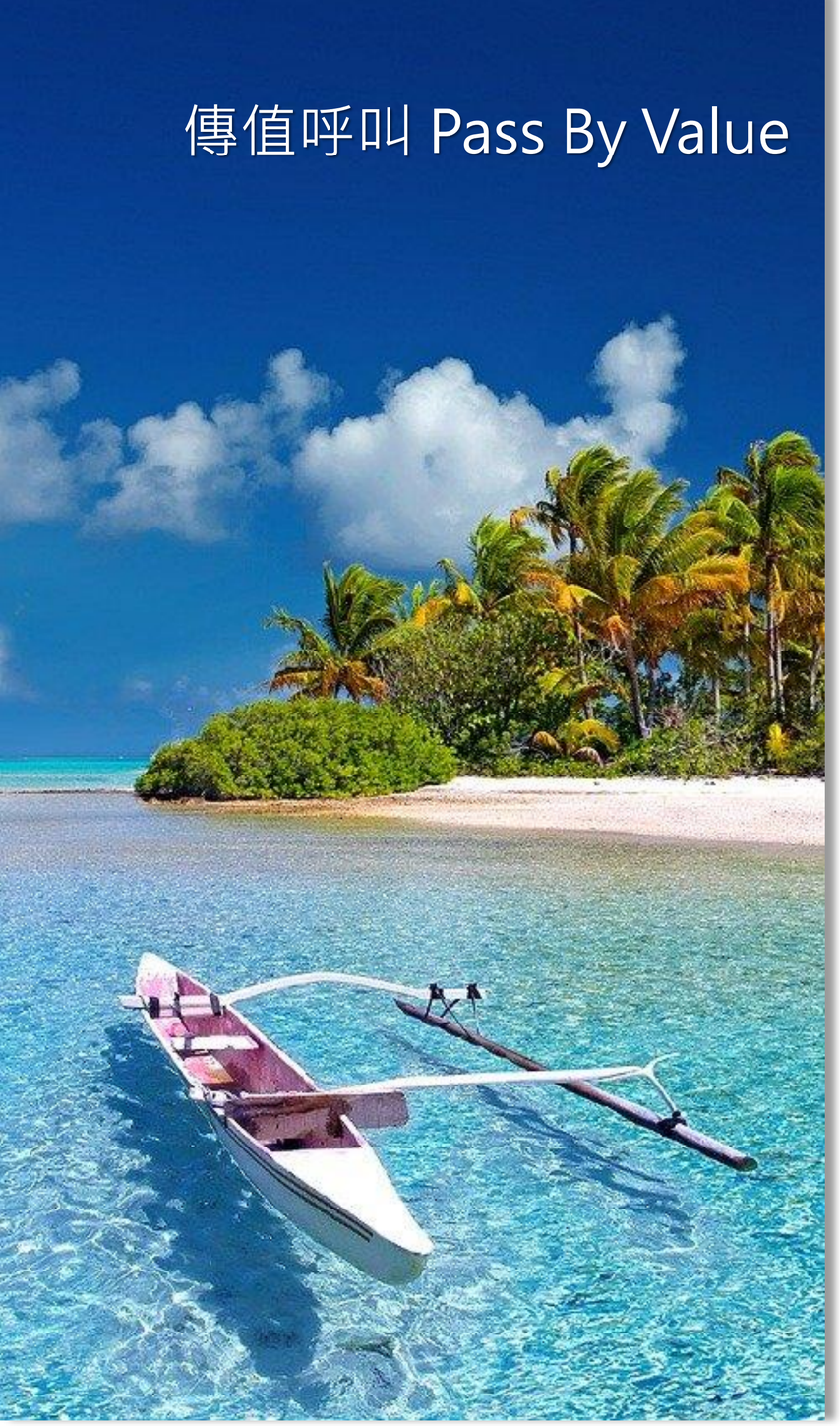
- Java 中參數的指派是傳遞目前 stack 中的內容
  - primitive type - 內容值

```
public class Shirt {
   public int shirtID = 101;
   public char colorCode = 'R';
   public double price = 299.0;
   public String description = "Polo Shirt";
   public void setPrice(double p) {
      price = p;
   }
}
```

```
public class TestShirt {

   public static void main(String[ ] args) {
      Shirt myShirt = new Shirt( );
      double newPrice = 199.0;
      myShirt.setPrice(newPrice);
      ..........
   }
}
```

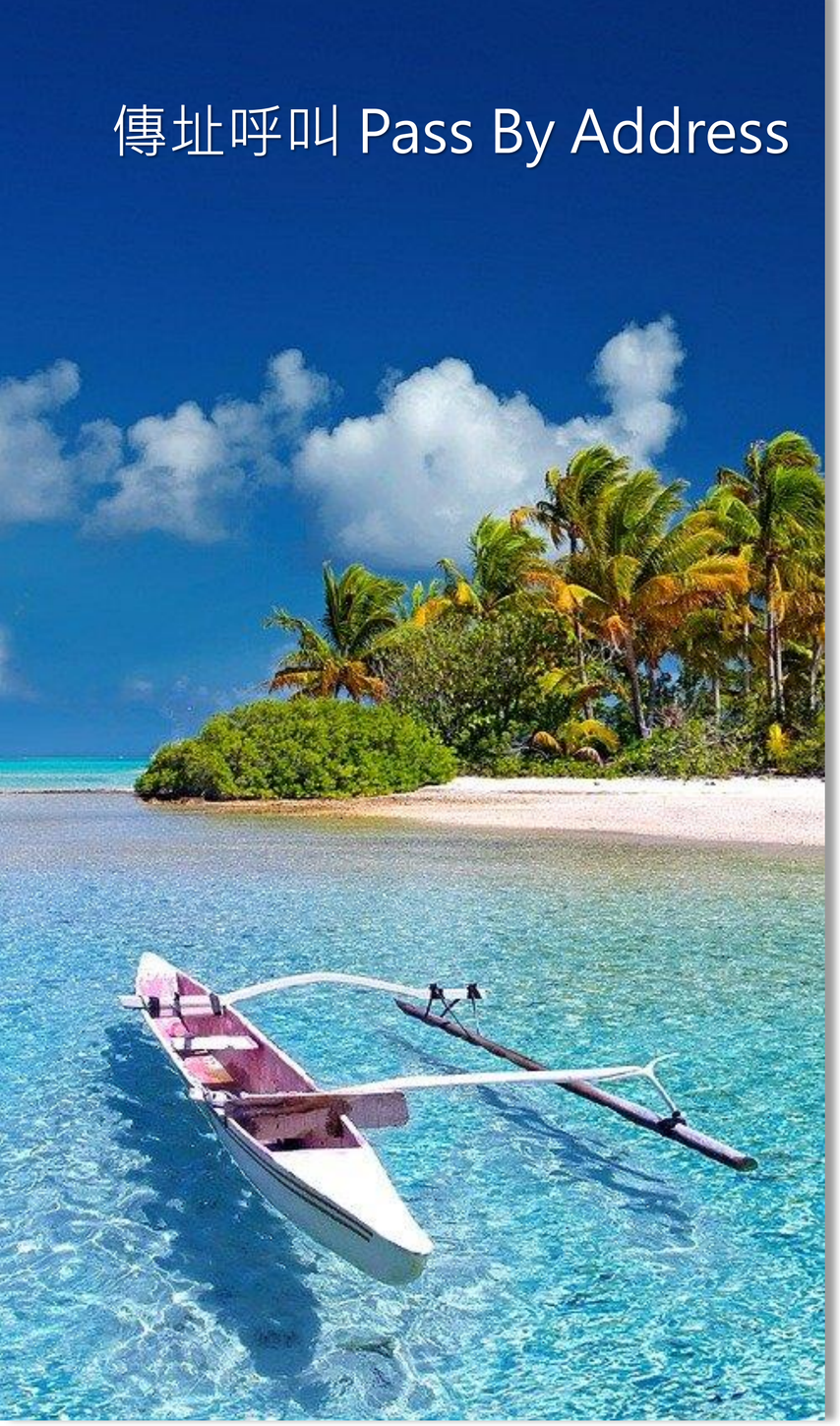


傳值呼叫 Pass By Value



```
public static void main(String[] args) {
        int m=1;
        System.out.println("呼叫changeValue()方法前m的值為:"+m);
        changeValue(m);
        System.out.println("----------------------------------------n");
        System.out.println("呼叫changeValue()方法後主方法中m的值為:"+m);
}
public static void changeValue(int m)
{
        System.out.println("changeValue方法內m的值為:"+m);
        m+=5;
        System.out.println("changeValue()方法內經過運算後之m的值為:"+m);
}
```

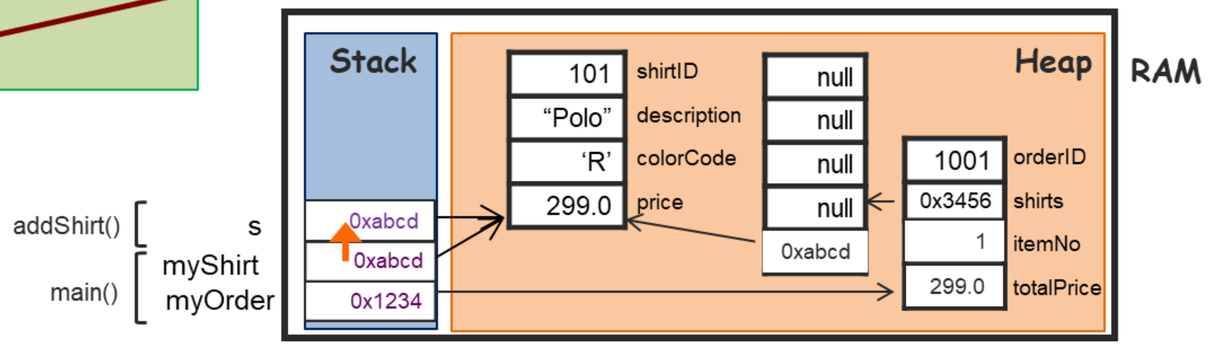
傳址呼叫 Pass By Address



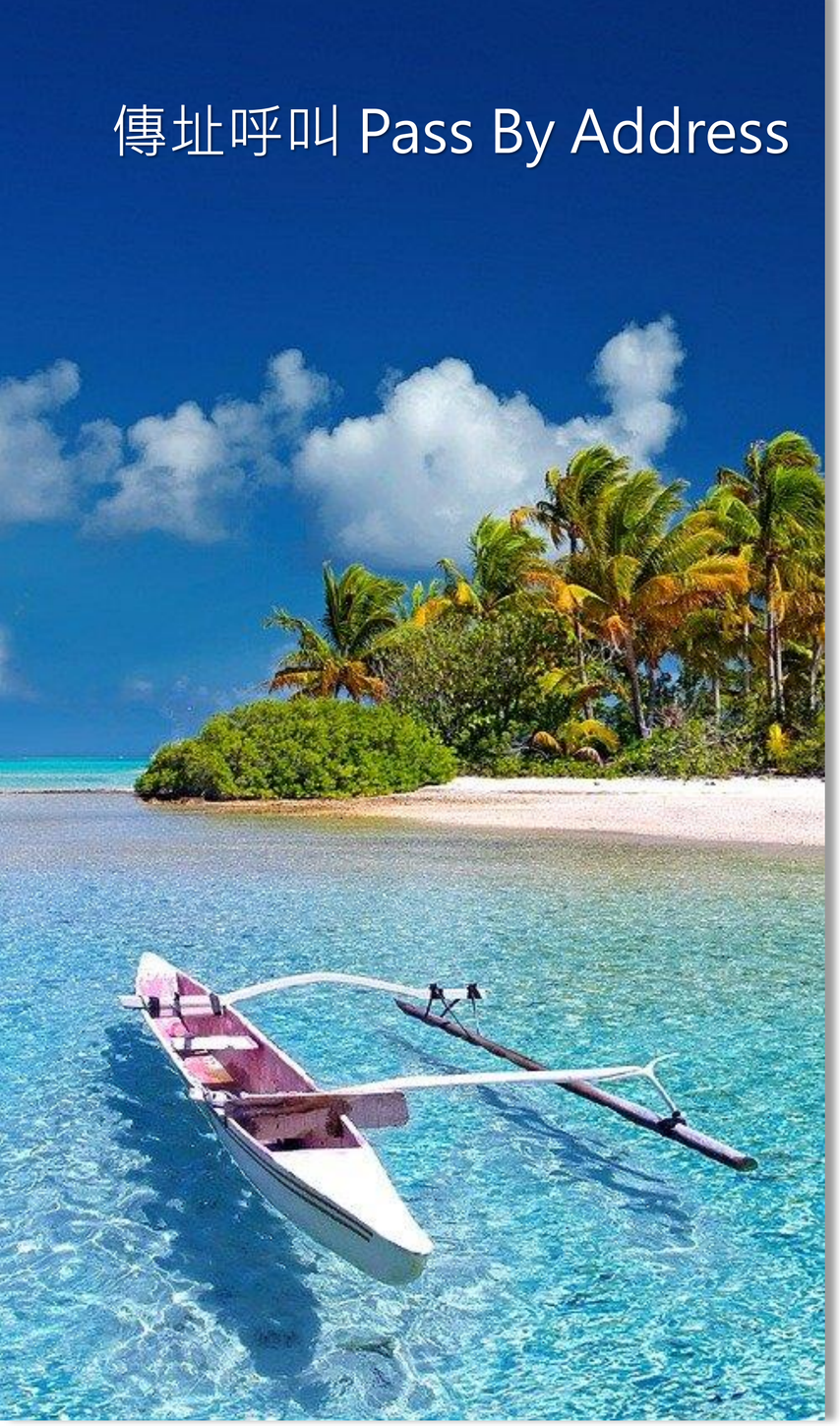
- Java 中參數的指派是傳遞目前 stack 中的內容
  - reference type - 參考值

```
public class Order {
   public int orderID = 1001;
   public Shirt[ ] shirts = new Shirt[5];
   public int itemNo = 0;
   public double totalPrice = 0.0;
   public void addShirt(Shirt s) {
      shirts[itemNo++] = s;
      totalPrice += s.price;
   }
}
```

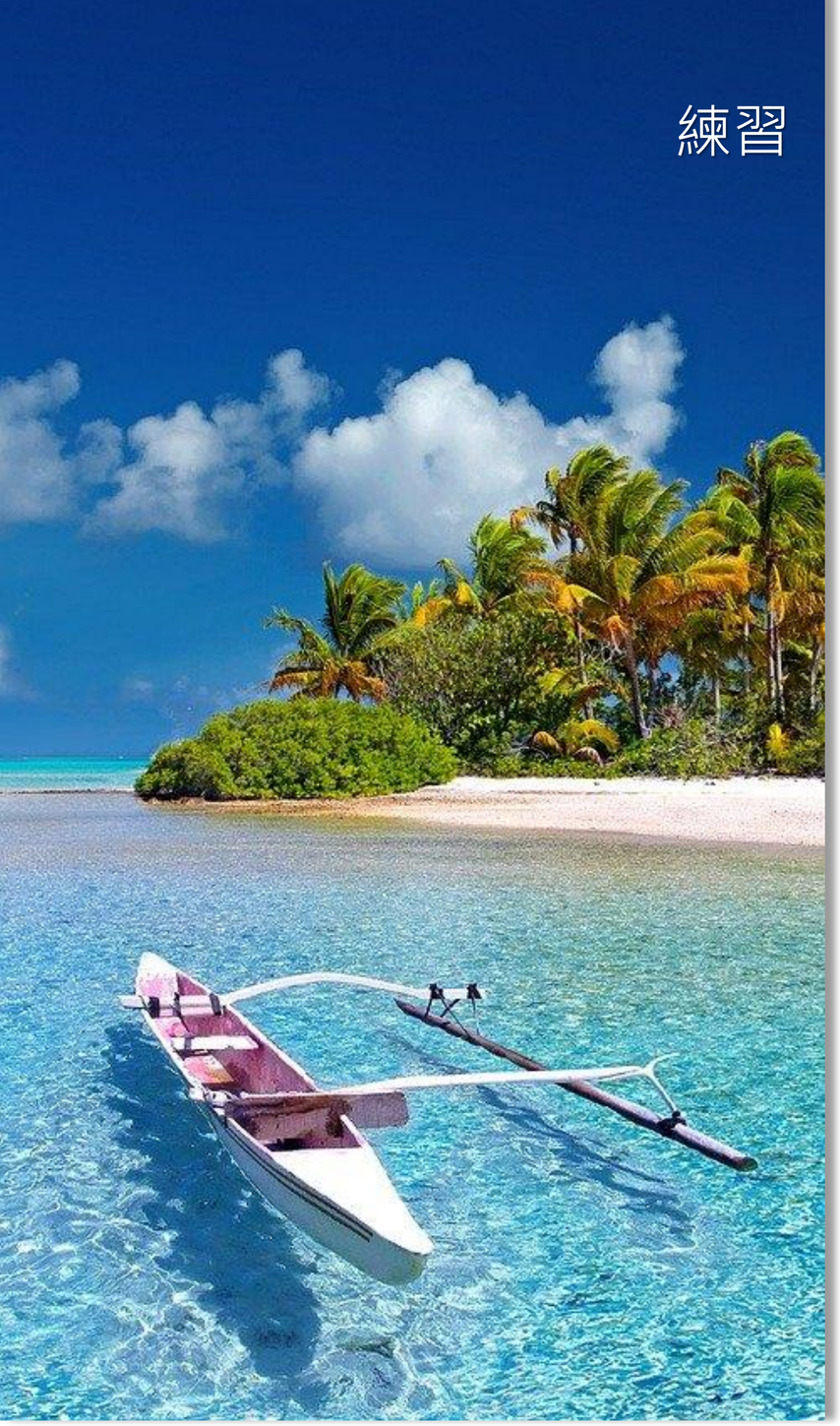
```
public class TestOrder {

   public static void main(String[ ] args) {
      Order myOrder = new Order( );
      Shirt myShirt = new Shirt();
      myOrder.addShirt(myShirt);
      ..........
   }
}
```



傳址呼叫 Pass By Address



```
public static void main(String[] args) {
      // TODO code application logic here
      int a[] = new int[]{20, 30, 11, 25, 8};
      System.out.println("呼叫方法前的陣列內容:");
      for (int x : a) {
         System.out.print(x + "\t");
      }
      callByRef(a);
      System.out.println("\n呼叫方法後的陣列內容:");
      for (int x : a) {
         System.out.print(x + "\t");
      }
   }

 public static void callByRef(int[] a) {
      System.out.println("\n方法內執行排序前之陣列內容");
      for (int x : a) {
         System.out.print(x + "\t");
      }
      Arrays.sort(a);
      System.out.println("\n方法內執行排序後之陣列內容");
      for (int x : a) {
         System.out.print(x + "\t");
      }
   }
}
```



練習

```
public class Pass{
    static public void main(String [] args) {
        int x = 5;
        Pass p = new Pass();
        p.doStuff(x);
        System.out.print( " main x =  " + x );
    }

    void doStuff(int x) {
        System.out.print( " doStuff x =  " + x++);
    }
}
```

A. doStuff x =6 main x =6

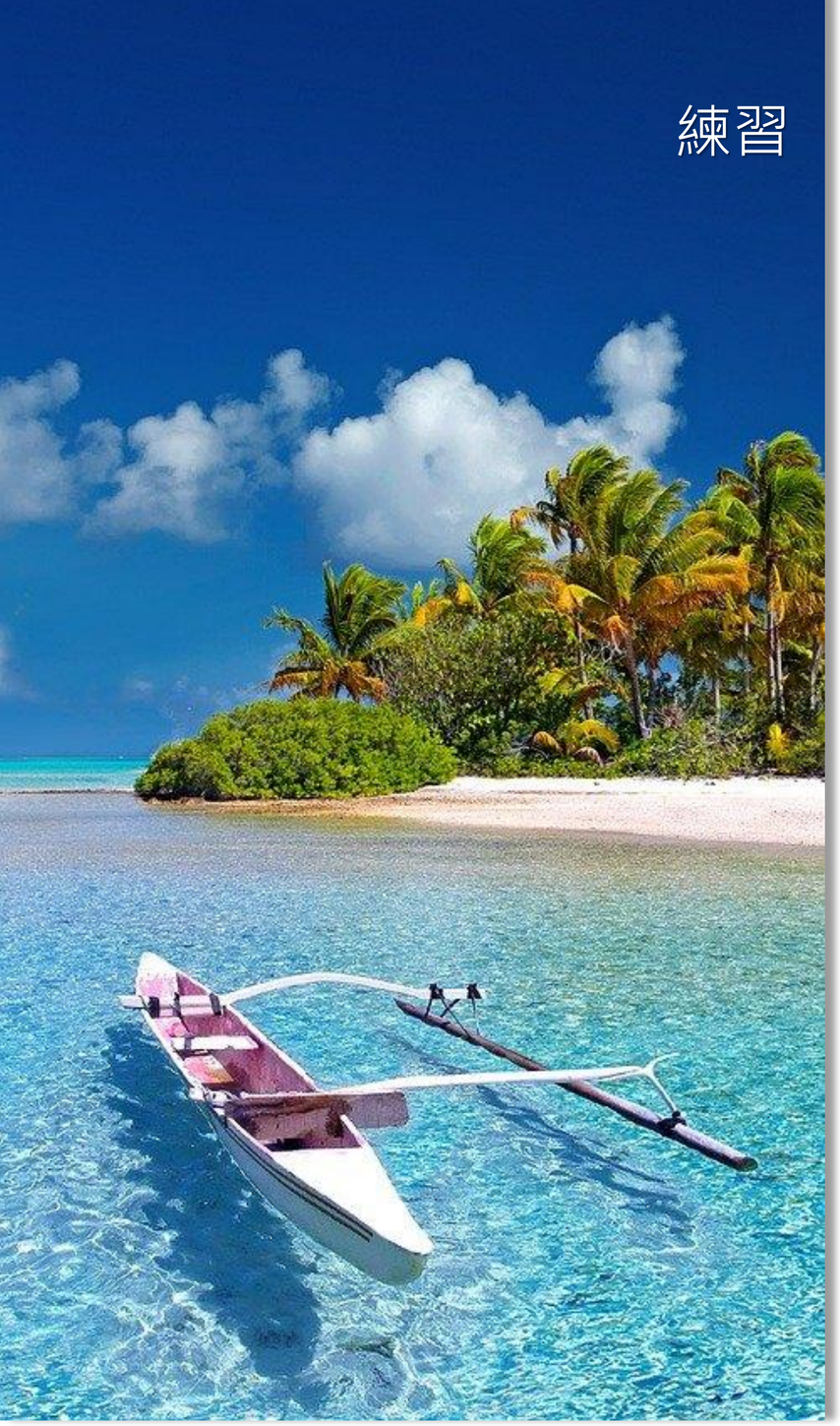
B. doStuff x = 5 main x = 5

C. doStuff x = 5 main x = 6

D. doStuff x = 6 main x = 5

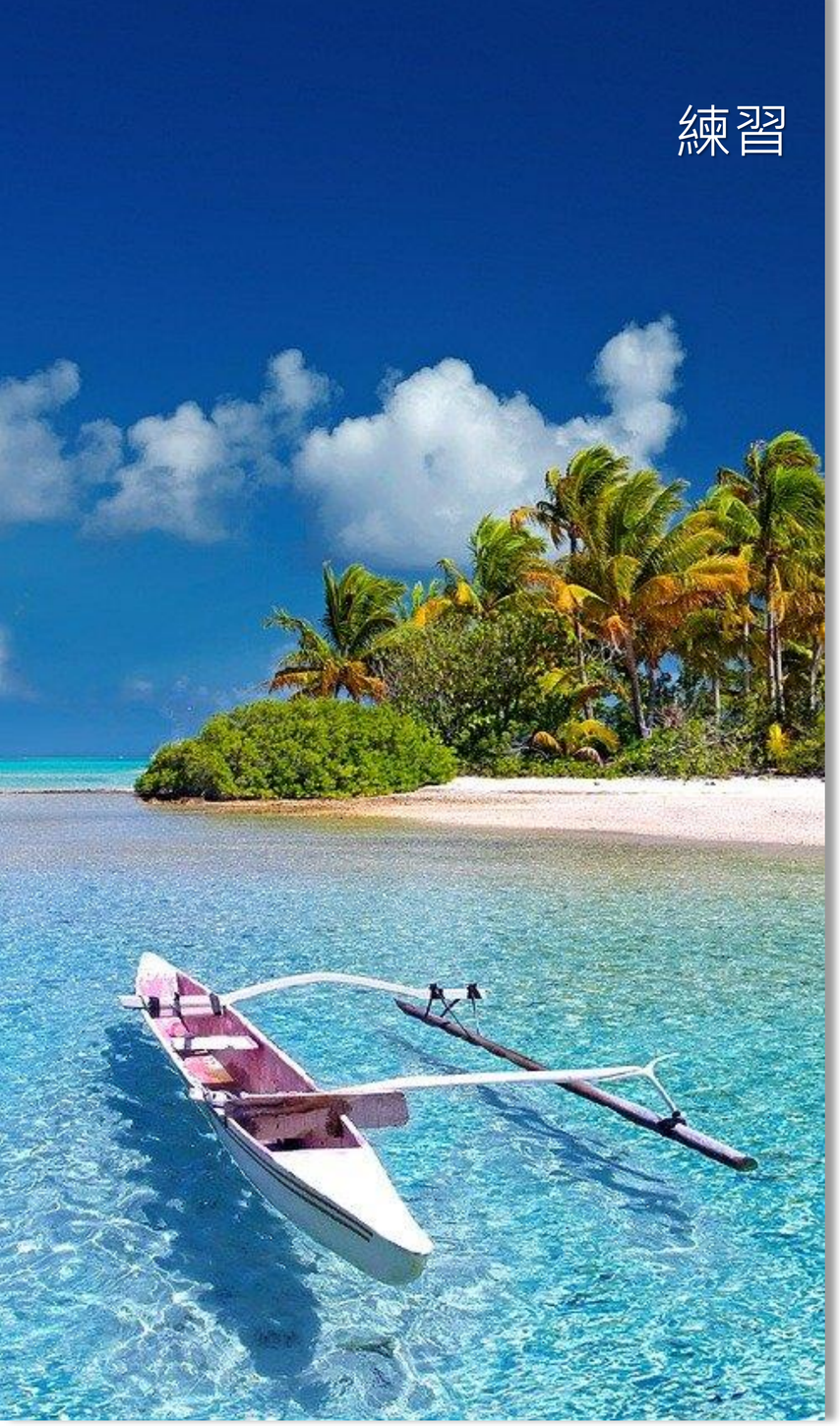
E. 編譯錯誤

F. 執行時拋出一個錯誤



練習

```
public static void main(String[] args) {
 new ThirteenthClass().first();
   }
   void first() {
      int i = 5;
      Value v = new Value();
      v.i = 25;
      second(v,i);
      System.out.println(v.i);
   }



void second(Value v, int i) {
      i = 0;
      v.i = 20;
      Value val = new Value();
      v = val;
     System.out.println(v.i + " " + i);
   }
}

class Value {
   int i = 15;
}
```



練習

```
class Value {
   public int i = 15;
}
public class Test {
   public static void main(String argv[]) {
      Value v = new Value();
      Test t = new Test();
      t.first(v);
      System.out.println("Main v.i=" + v.i);
   }
   public void first(Value v) {
      v = new Value();
      v.i = 25;
      System.out.println("First v.i=" + v.i);
    }
}
```

A. First v.i=25 Main v.i=25

B. First v.i=15 Main v.i=15

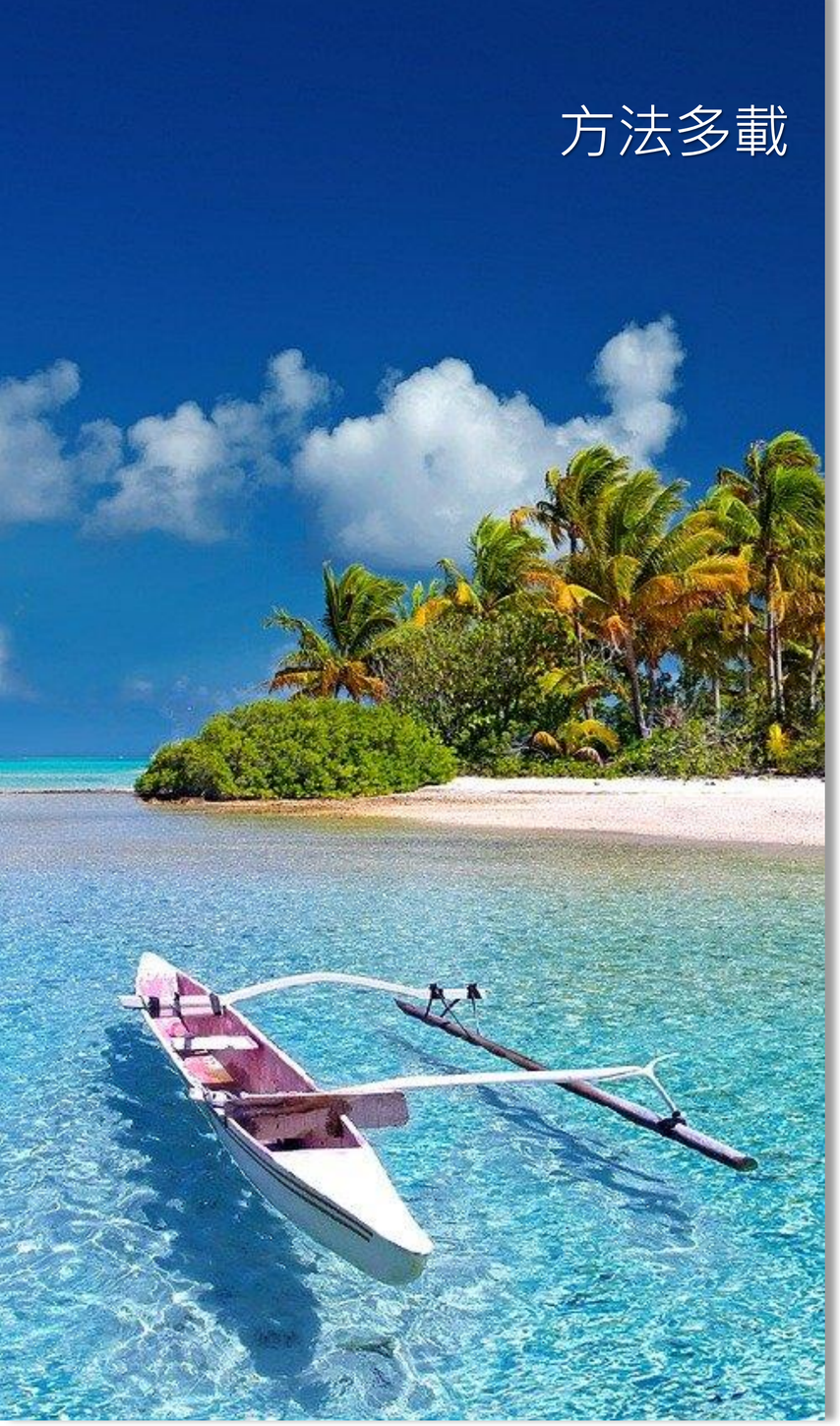
C. First v.i=25 Main v.i=15

D. First v.i=15 Main v.i=25

E. 編譯錯誤

F. 執行時拋出一個錯誤

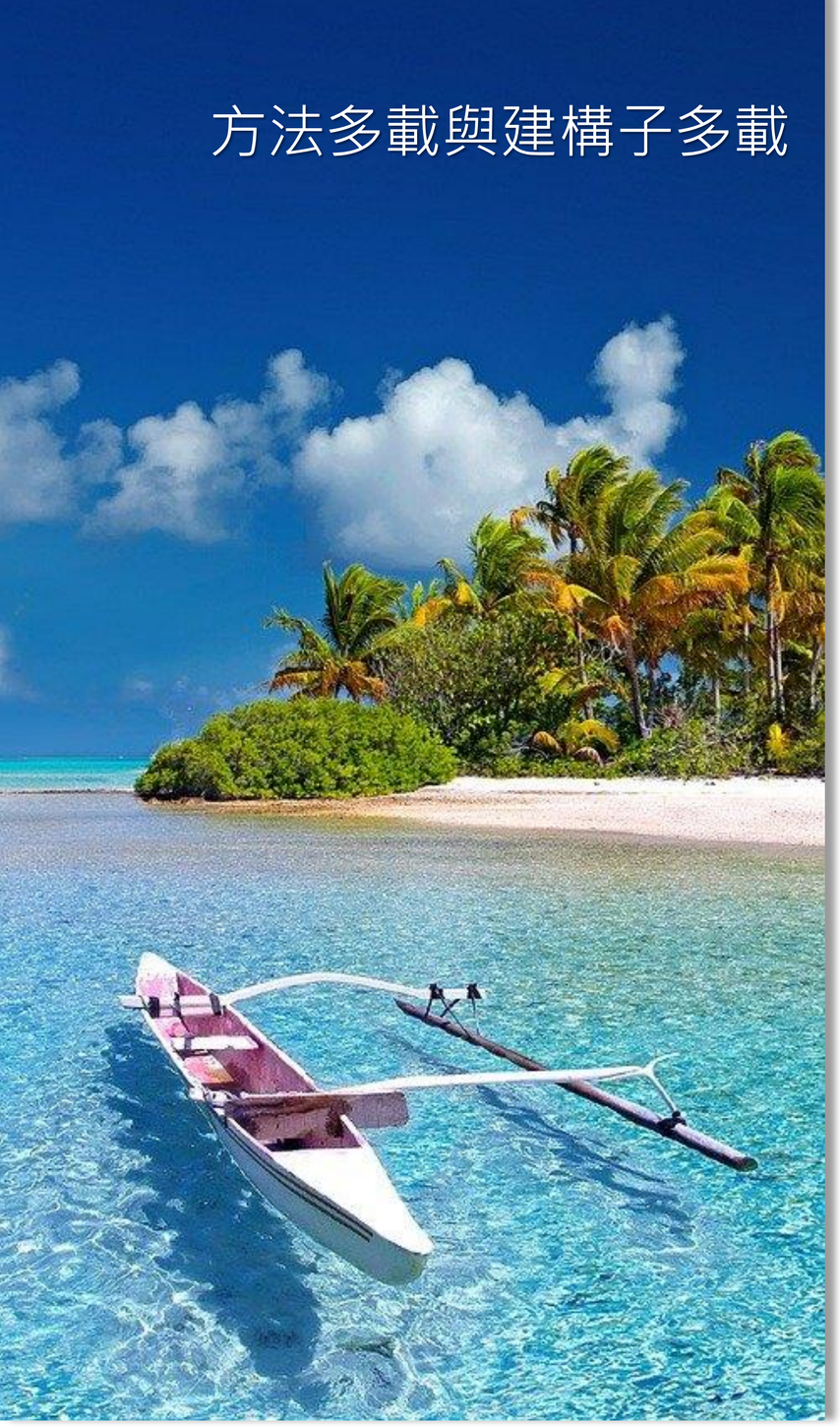
多載 OVERLOADING



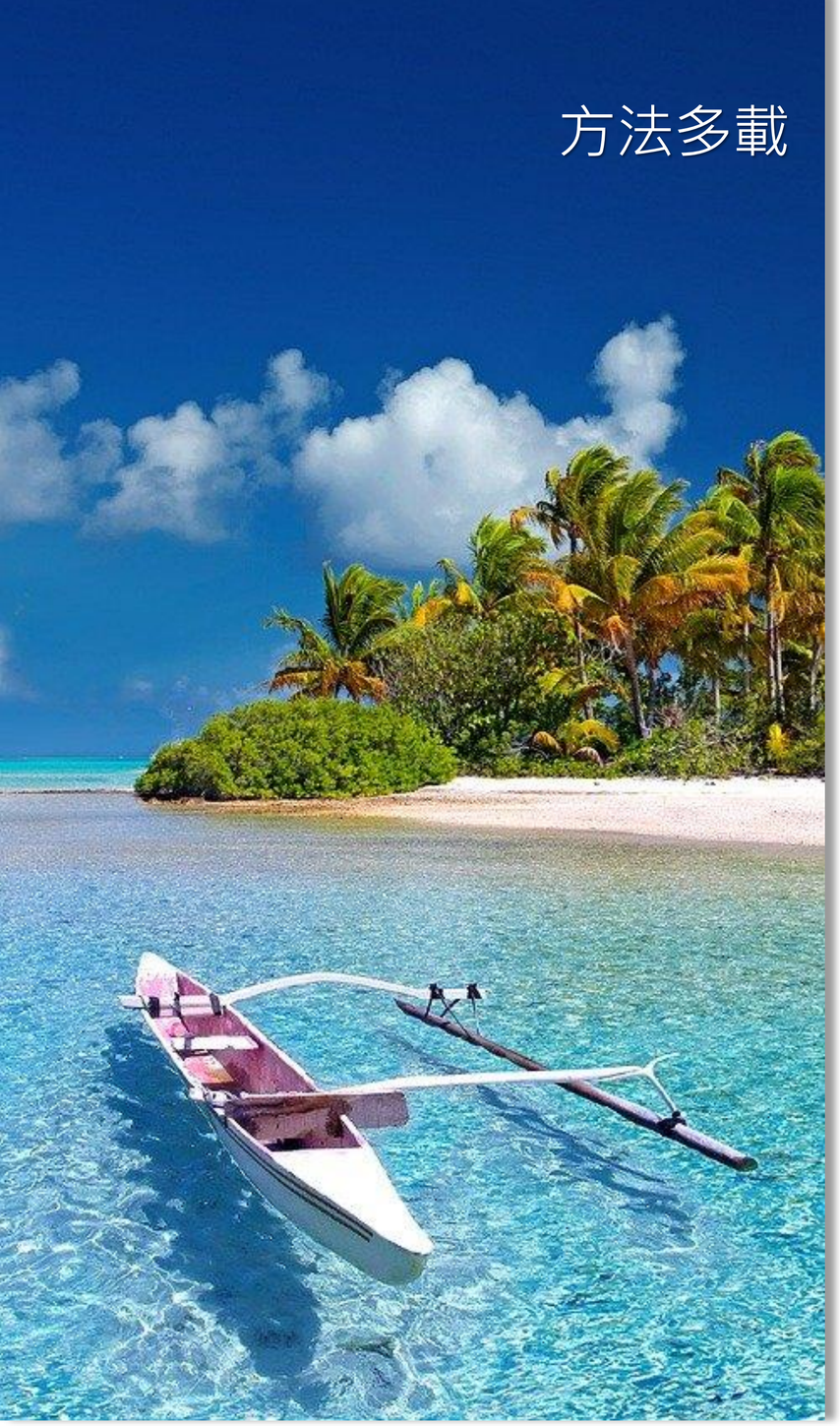
方法多載

- 在類別中可以宣告多個名稱相同，功能相同，但參數型別不同或參數數量不同的方法
- 注意事項
  - 必須使用不同的參數，參數數量相同時，參數型別需不同，參數型別皆相同時，參數的數量需不同，同時需要注意晉升轉型的規則
  - 方法多載的宣告與方法回傳型別無關，即void跟int有回傳值視為同一方法，無法多載
  - 參數列的數量相同，型別相同，但位置不同也可多載

方法多載與建構子多載



方法多載(method overloading)	建構子多載（constructor overloading)
public void aMethod( ){...}	public Base( ){.....}
public void aMethod(int i){...}	public Base(int i){.....}
public void aMethod(String s , int i){...}	public Base(String s , int i){.....}
public void aMethod(int i , String s){...}	public Base(int i , String s){.....}



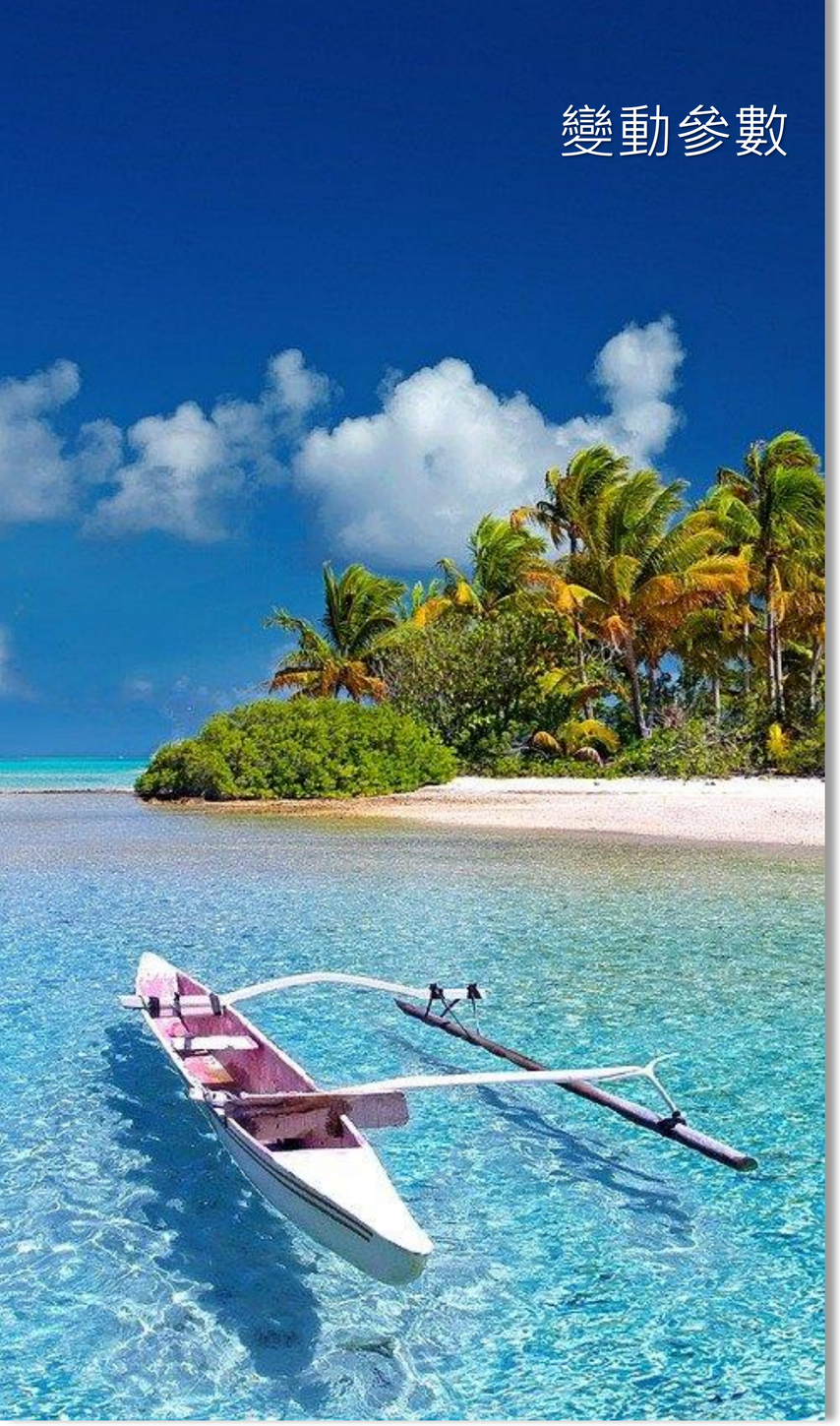
方法多載

```
01 public static void main(String[] args)
02 {
03     System.out.println("1+2="+add(1,2));
04     System.out.println("1.5+2.5="+add(1.5,2.5));
05     System.out.println("1+2+3="+add(1,2,3));
06 }
07
```

```
run:
1+2=3
1.5+2.5=4.0
1+2+3=6
BUILD SUCCESSFUL (total time: 0 seconds)
```

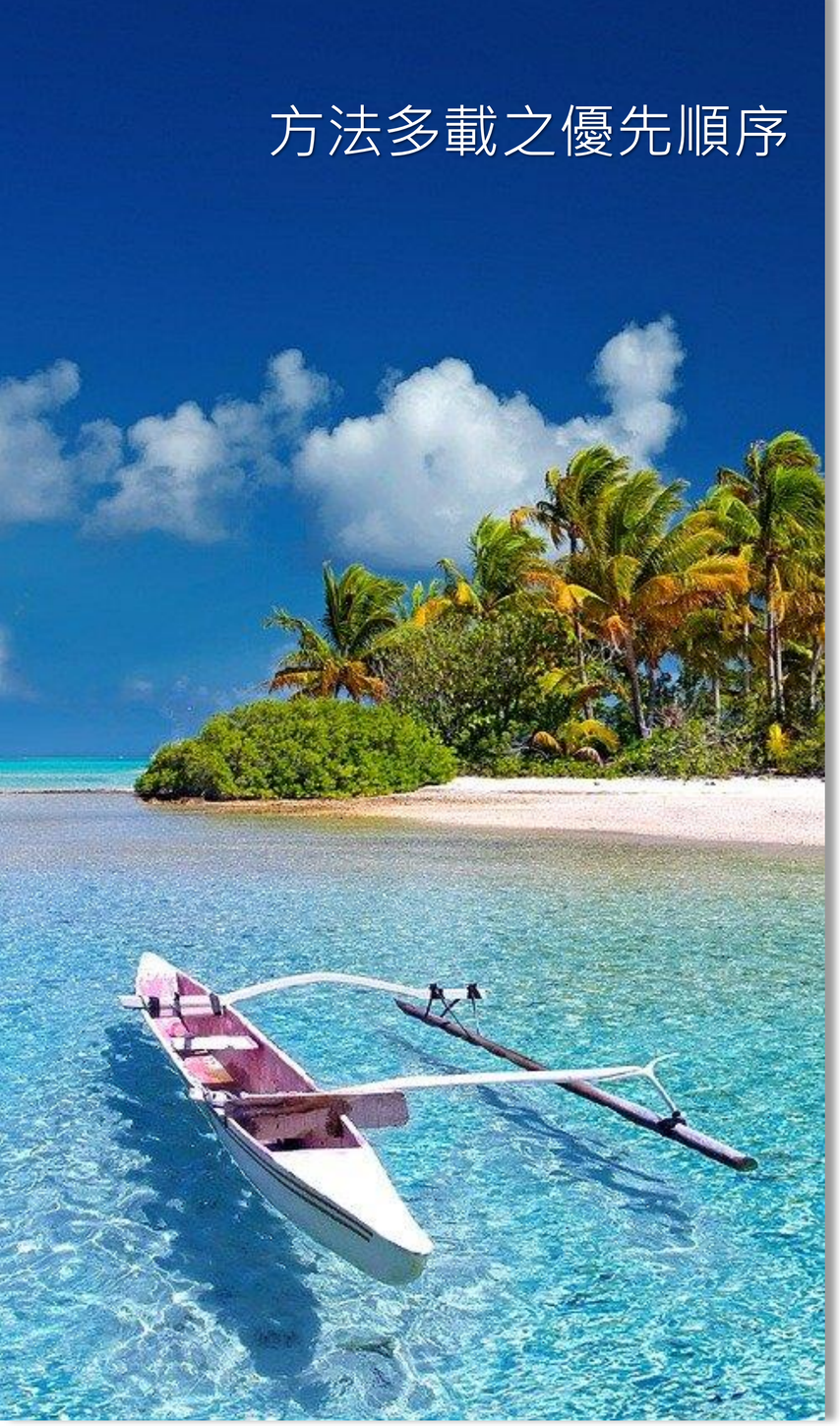
```
01 public static int add(int a, int b)
02 {
03     return a + b;
04 }
05
06
07 public static double add(double a, double b)
08 {
09     return a + b;
10 }
11
12
13 public static int add(int a, int b, int c)
14 {
15     return a + b + c;
16 }
```

變動參數



- 方法名稱相同，但無法確定呼叫方法時會傳遞幾個引數時，可以使用變動參數，編譯器在編譯時，會將變動參數視為陣列
  - [存取修飾字]　[修飾語]　回傳值型別　方法名稱（型別 ...參數）{
  - //程式區段
  - [return 運算式;]
  - }
- 注意事項
  - 其中...只能三個，不能多也不能少，且變動參數一定要寫在最後
  - 一個方法最多只能有一個省略符號
  - 當使用省略符號做為方法中唯一的參數列時，呼叫端不傳進引數也是被允許的
  - 要存取變動參數，因其視為陣列，所以要存取變動參數，需以陣列方式來存取

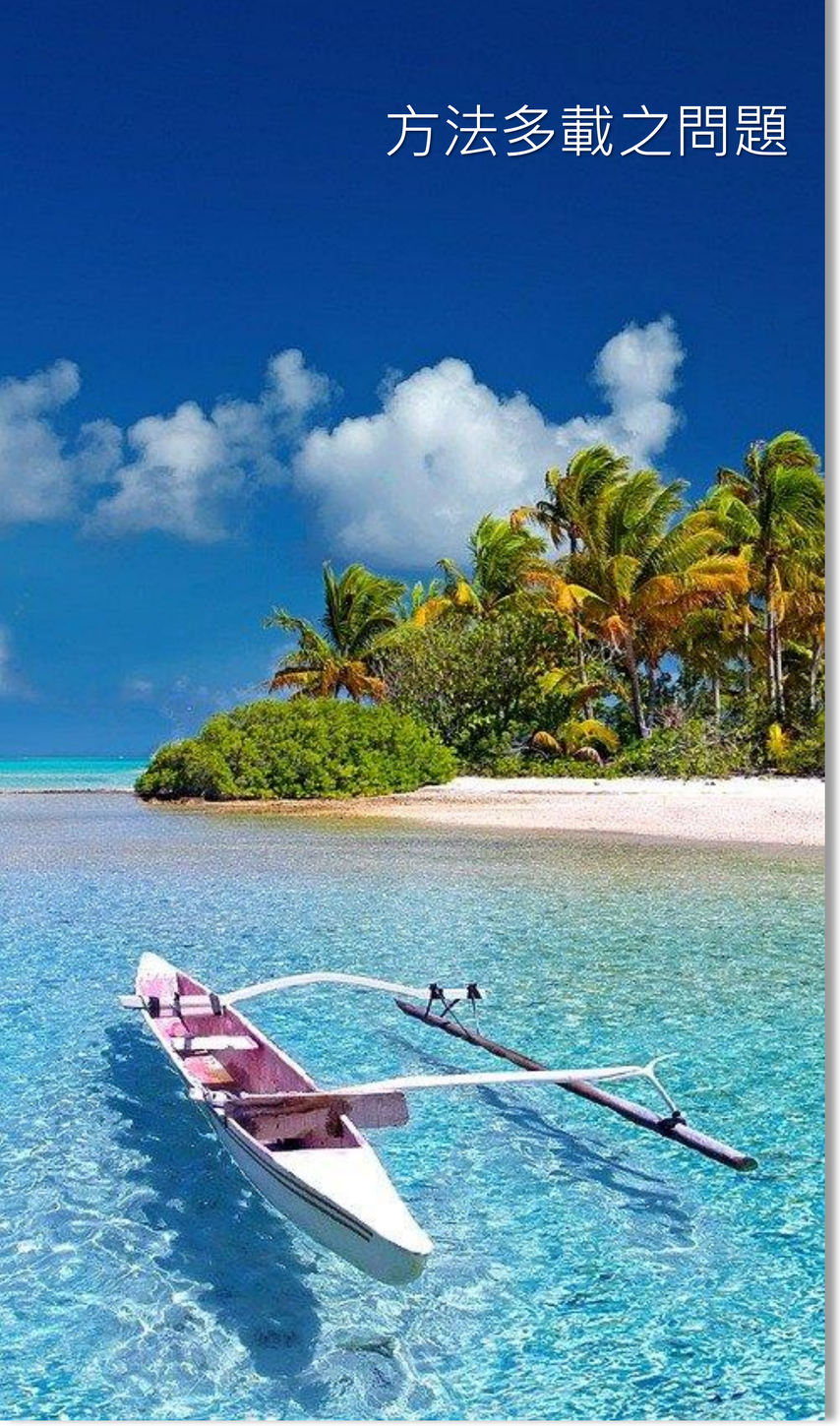
方法多載之優先順序



- 相同型態>自動轉型>自動裝箱>變動參數

```
public static String overLoading(int a) {
        return "intA";
    }
    public static String overLoading(double a) {
        return "doubleA";
    }
    public static String overLoading(Integer a) {
        return "IntegerA";
    }
    public static String overLoading(int...a) {
        return "variableA";
    }
```

方法多載之問題



呼叫多載

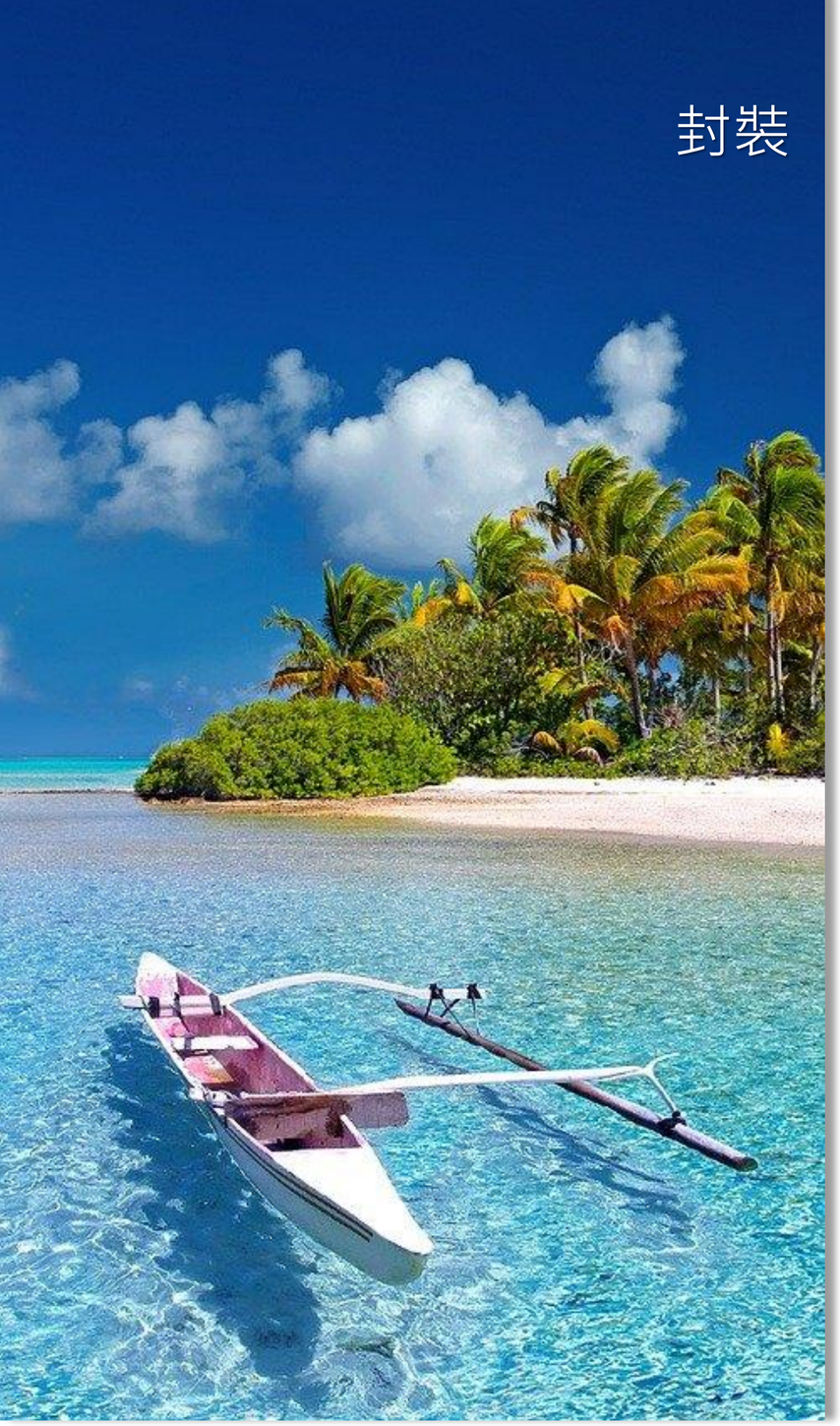
```
System.out.println(overLoading(10,20));
```

方法多載

```
public static String overLoading(double a,int b) {
        return "DoubleA";
 }
public static String overLoading(int a,double b) {
        return "DoubleA";
}
```

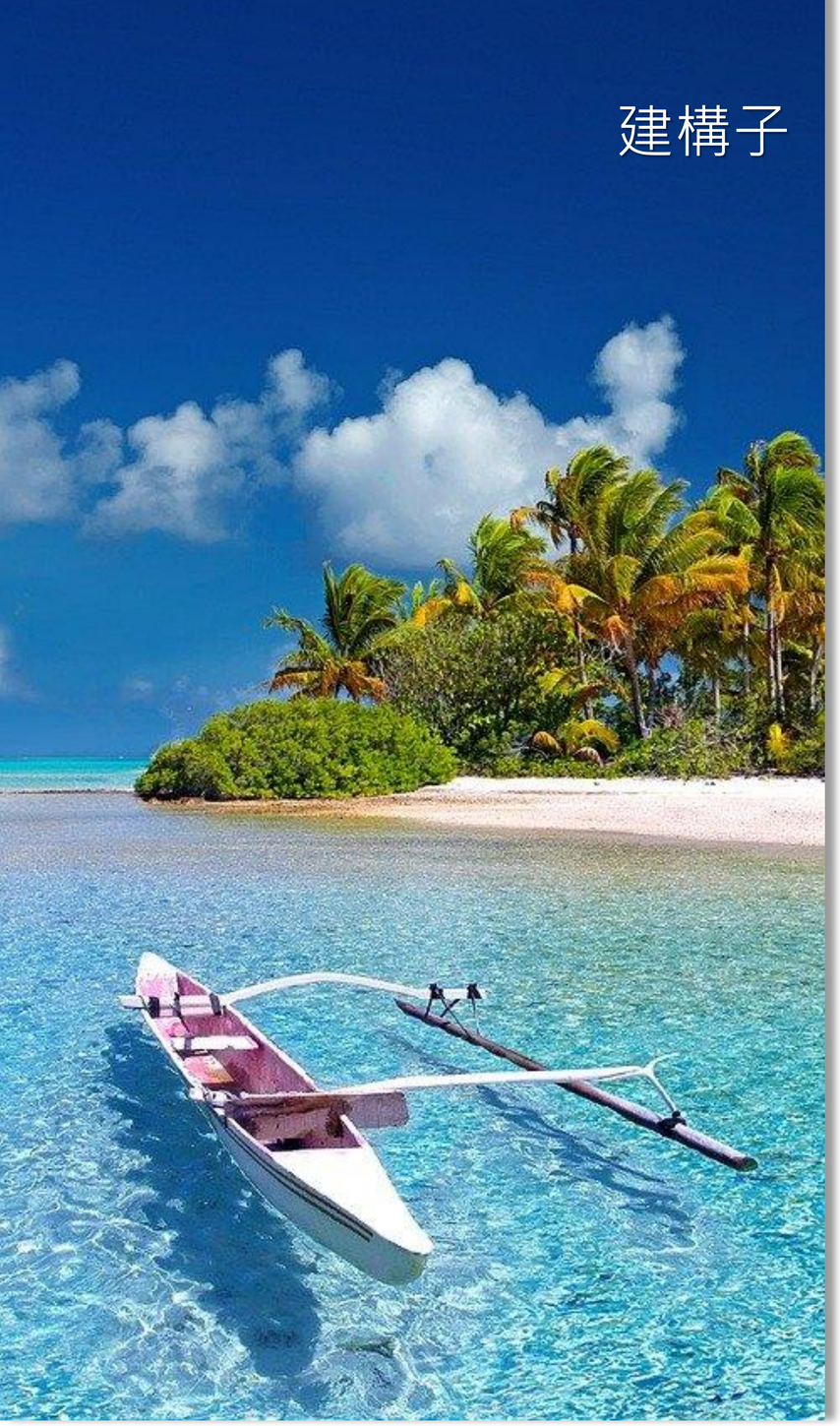
封裝 ENCAPSULATION

封裝



- 將類別內的屬性或方法隱藏，外界無法直接操作使用，僅能透過提供的方法介面來取得類別中不允許被外界存取的內部資料，其目的在維護資料的完整性與存取安全
- 物件導向程式設計的封裝是將屬性和方法放在同一個類別內，並使用存取權限修飾字(public private protected 或 default)來定義內部的屬性與方法的存取限制
- 設值與取值
  - 因為變數設為private所以外界無法存取，此時可利用公有的方法讓外界可以存取變數setXXX與getXXX其中XXX為變數名稱

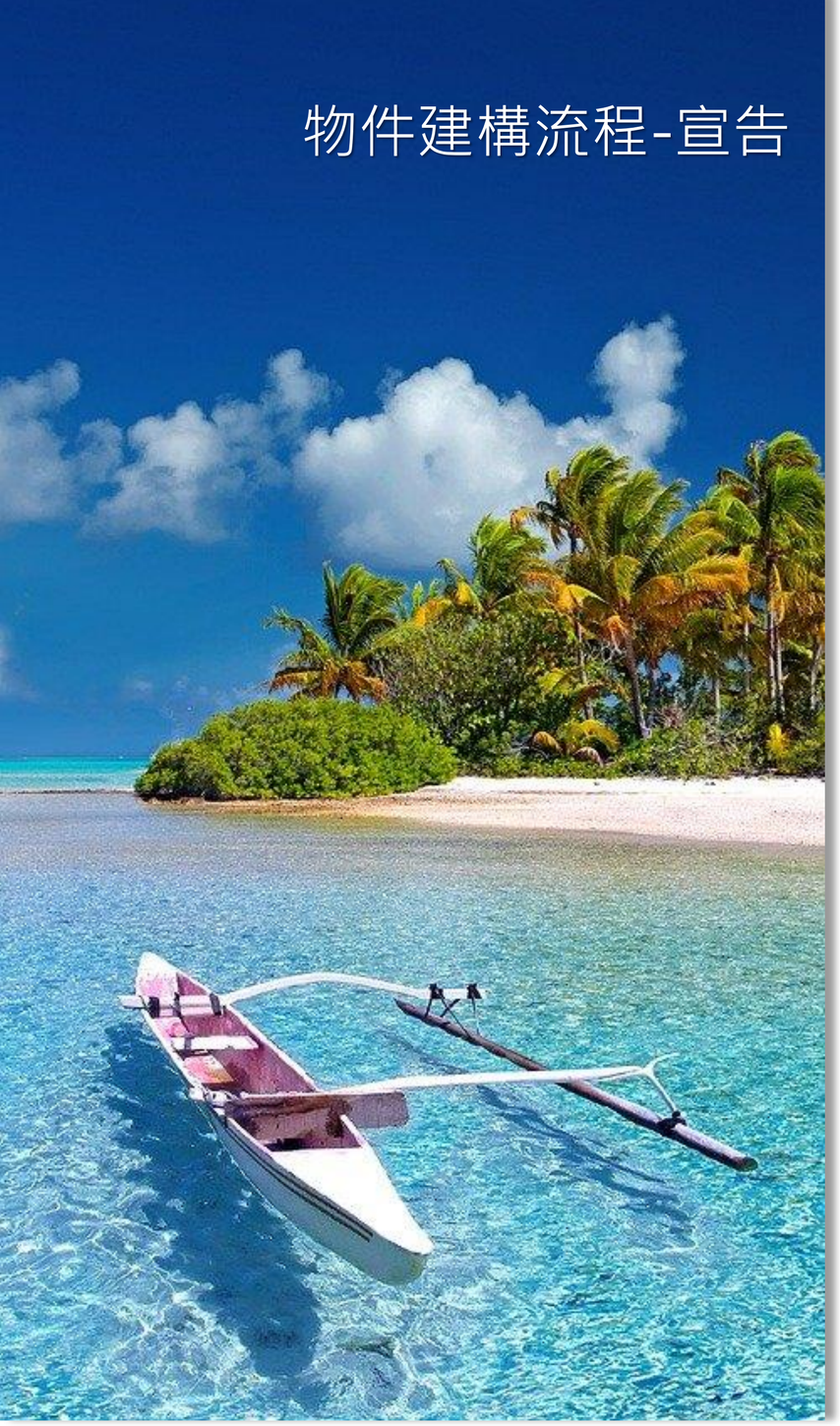
建構子 CONTRUCTOR



建構子

- 物件建立時，可以透過建構子來動態指定物件變數初始值
- 建立物件語法
  - 類別名稱 物件參考名稱=new 類別名稱( );
- 注意事項
  - 建構子名稱必須與類別名稱相同，物件建立時會同時執行建構子
  - 類別中若無定義建構子，編譯器在編譯時會加入預設建構子，預設建構子沒有參數列與程式敘述句
  - 建構子和方法類似，但建構子沒有型別與回傳值
  - 在類別提供多組不同建構子來初始化，稱為建構子多載（Constructor Overloading）
  - 每一個 class最多只能有一個預設建構子
  - 預設建構子的存取權限與該類別的存取權限相同
  - 除了初始成員變數或繼承時 super() 的定義外，預設建構子基本上是沒有其它的程式敘述
  - 在預設建構子或其它建構子(有不同參數)要在呼叫另一個建構子,不能直接寫建構子名稱,會出現cannot find symbol,因為會被當成呼叫另一方法,所以要改為this(參數)

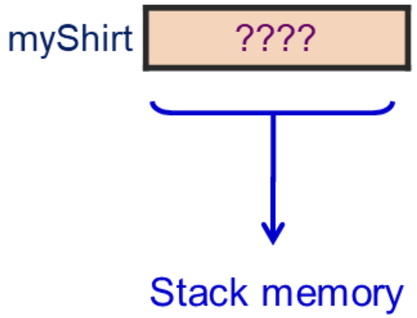
物件建構流程-宣告



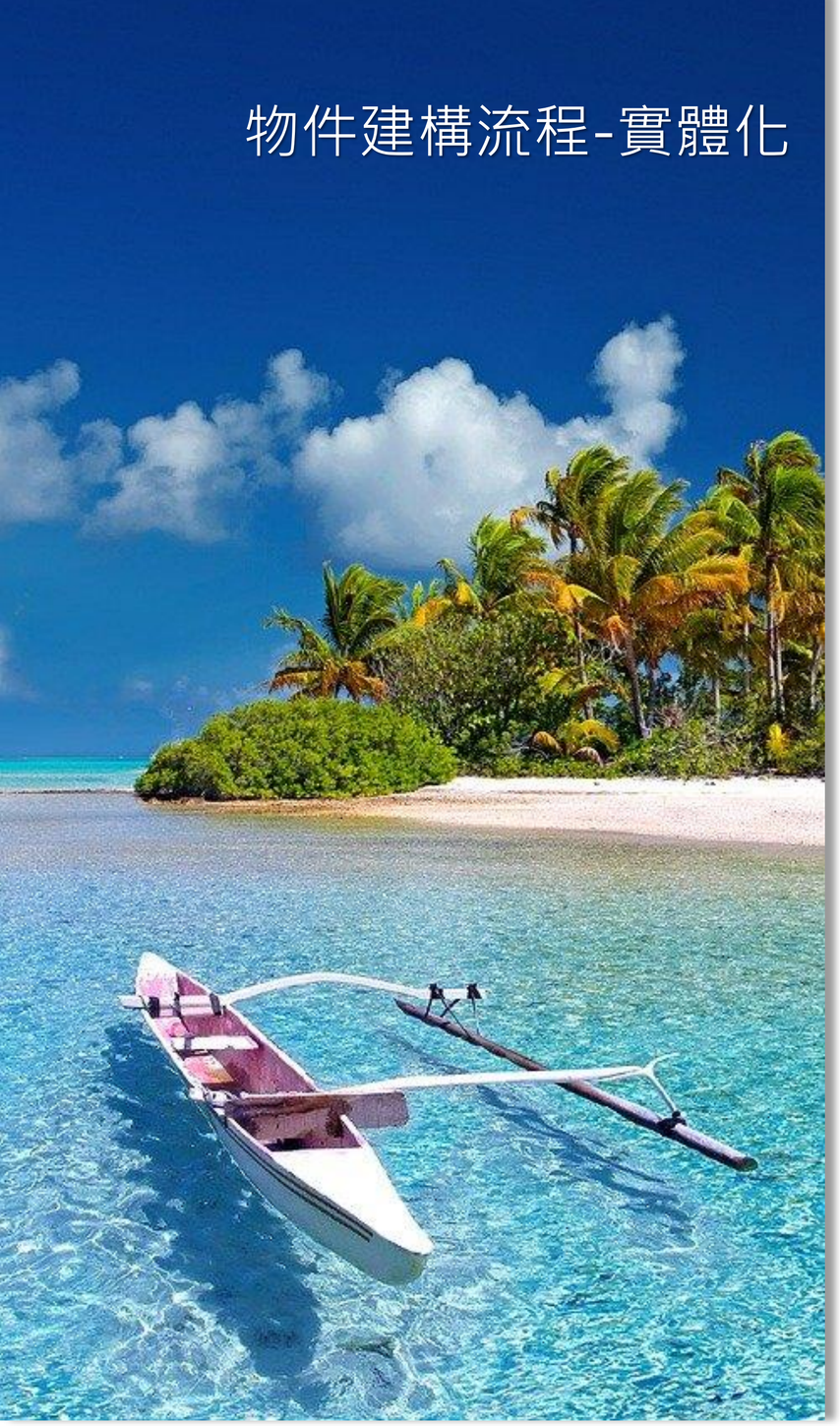
```
public class Shirt {
   private int shirtID = 101;
   private char colorCode = 'R';
   private double price = 299.0;
   private String description = "Polo Shirt";

   public Shirt(char c, double p, String d){
      colorCode = c;
      price = p;
      description = d;
   }

}
```

Shirt myShirt;

myShirt = new Shirt('G', 199.0 , "T-Shirt");

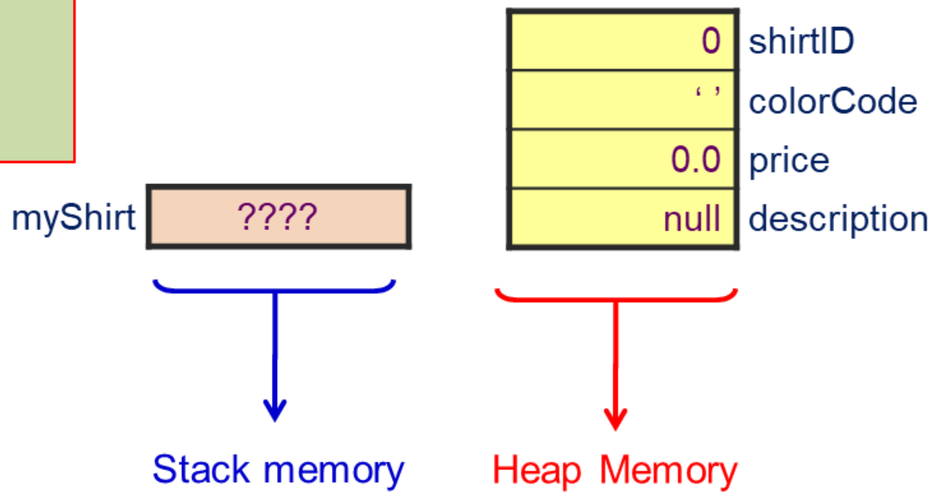


物件建構流程-實體化

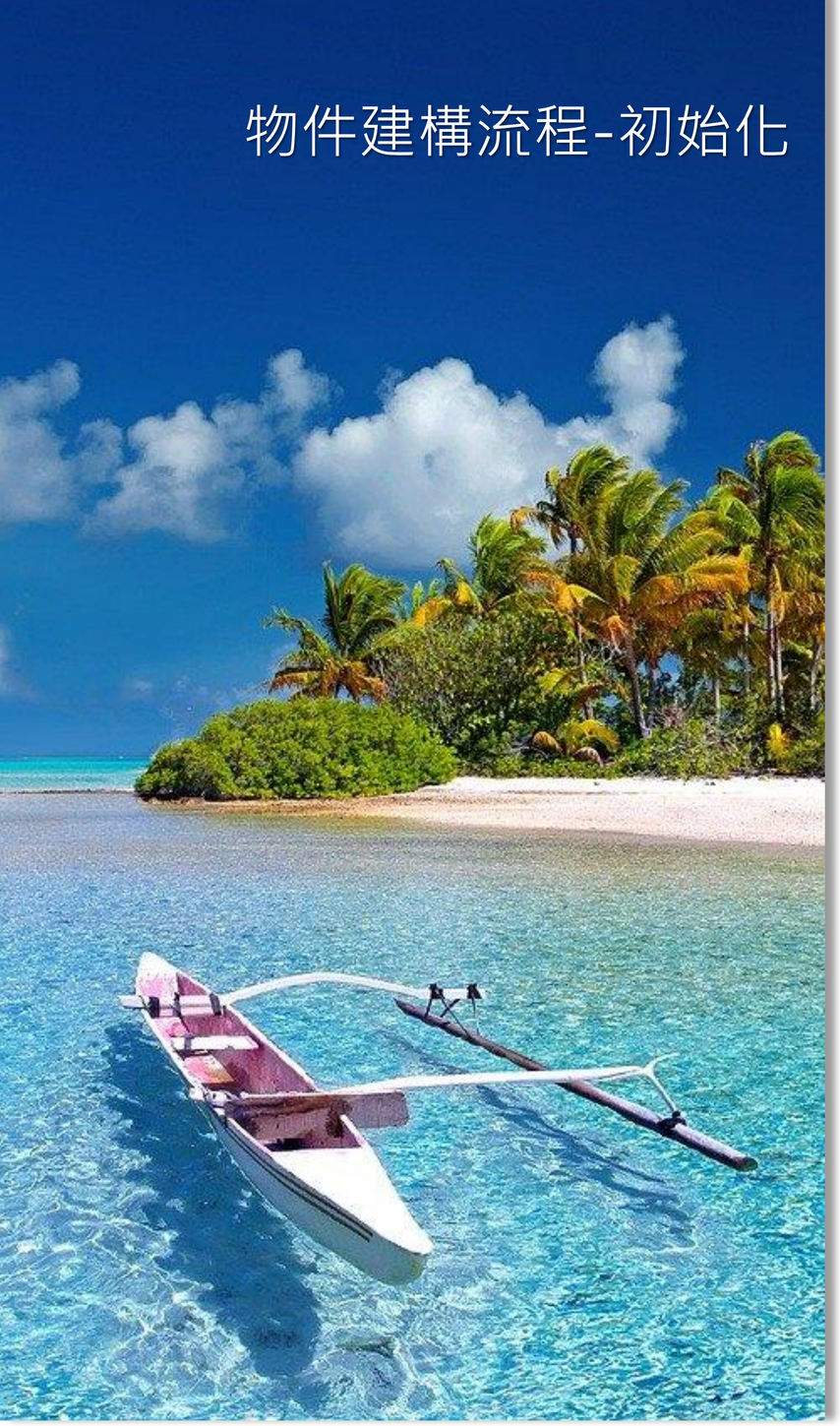


```
public class Shirt {
   private int shirtID = 101;
   private char colorCode = 'R';
   private double price = 299.0;
   private String description = "Polo Shirt";

   public Shirt(char c, double p, String d){
      colorCode = c;
      price = p;
      description = d;
   }

}
```

```
Shirt myShirt;
myShirt = new Shirt ('G', 199.0, "T-Shirt" );
```

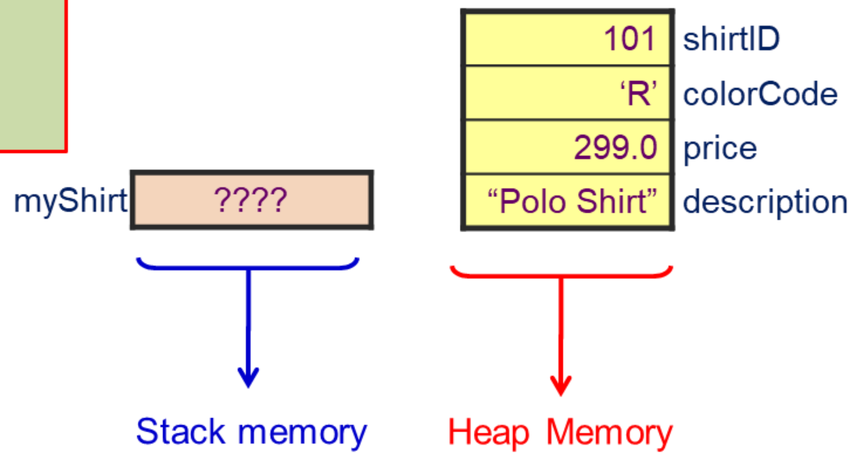


物件建構流程-初始化

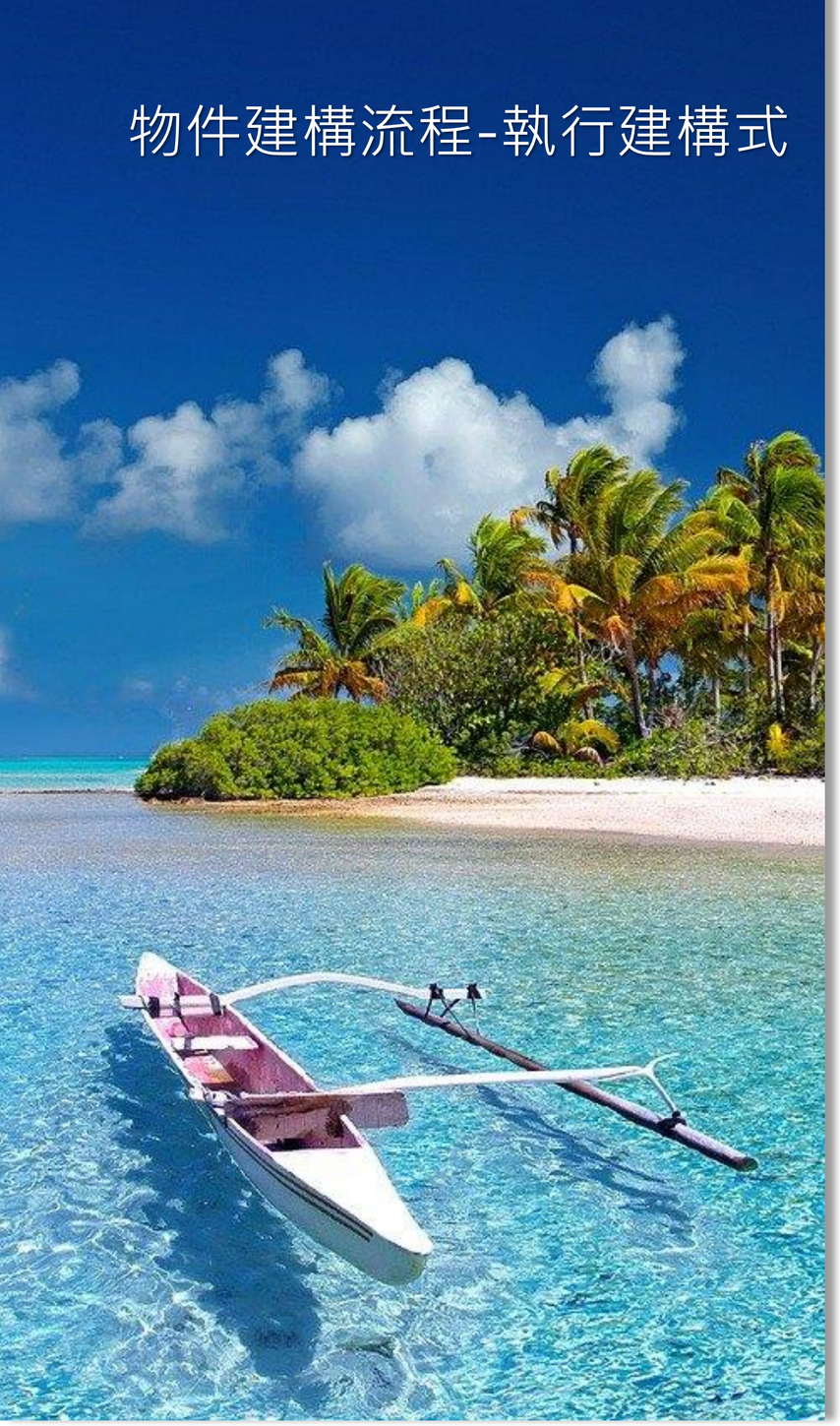


```
public class Shirt {
   private int shirtID = 101;
   private char colorCode = 'R';
   private double price = 299.0;
   private String description = "Polo Shirt";

   public Shirt(char c, double p, String d){
      colorCode = c;
      price = p;
      description = d;
   }

}
```

```
Shirt myShirt;
myShirt = new Shirt ('G', 199.0, "T-Shirt" );
```

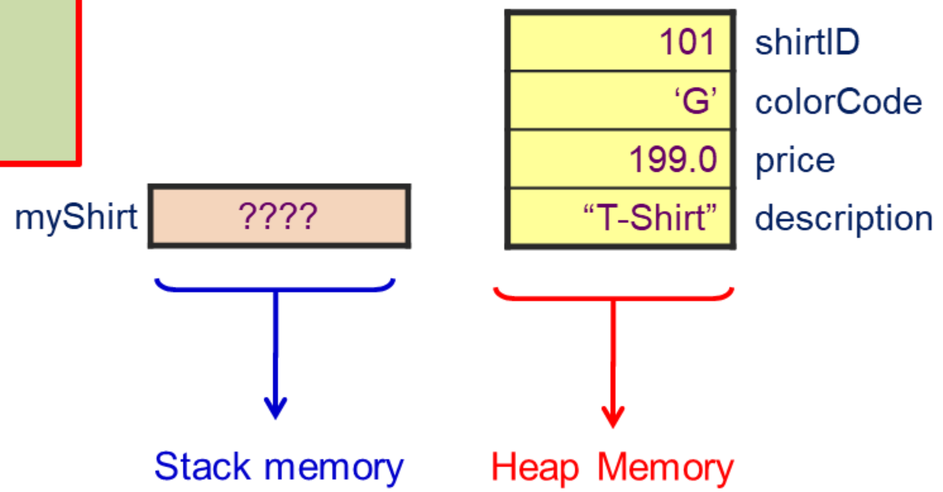


物件建構流程-執行建構式

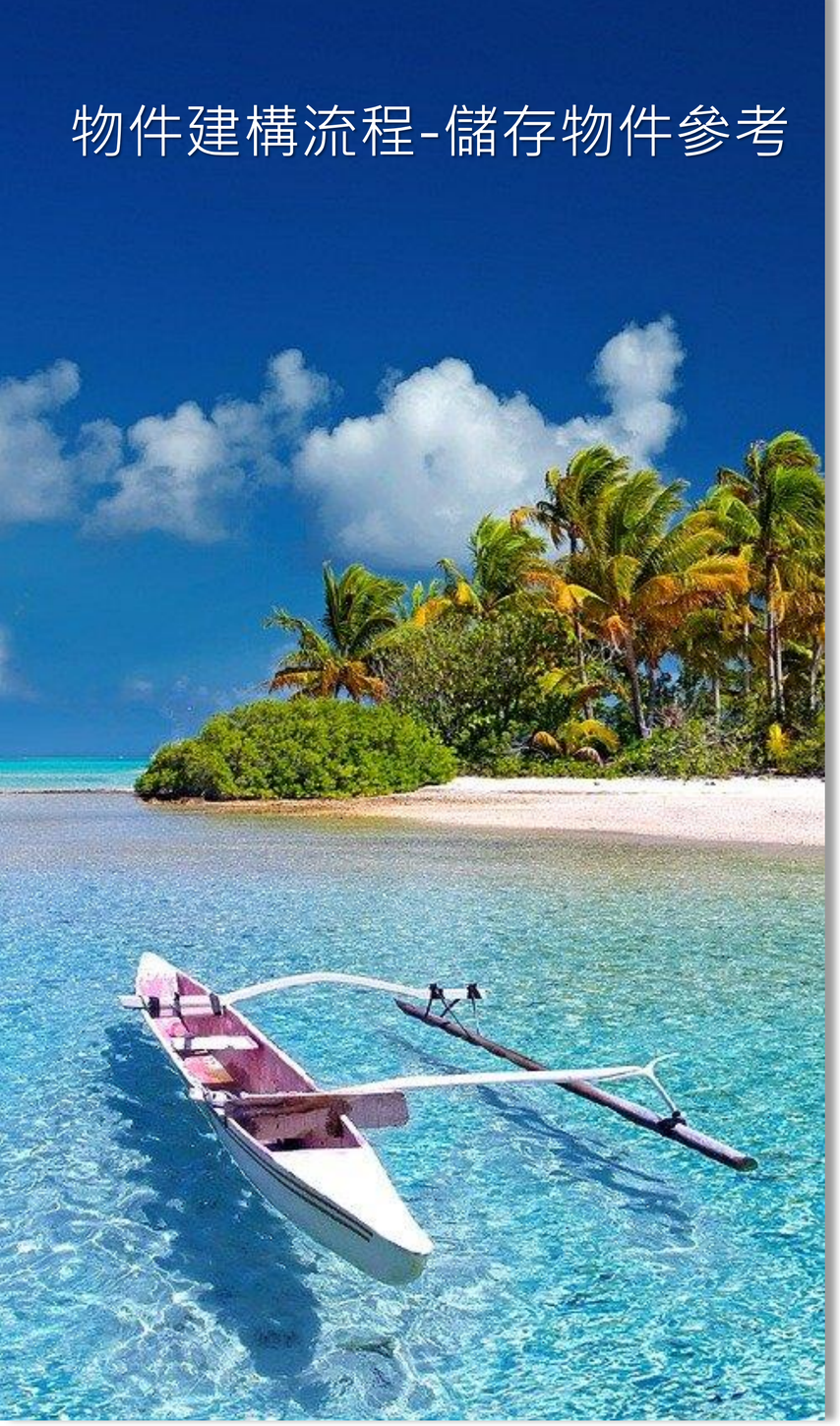


```
public class Shirt {
   private int shirtID = 101;
   private char colorCode = 'R';
   private double price = 299.0;
   private String description = "Polo Shirt";

   public Shirt(char c, double p, String d){
      colorCode = c;
      price = p;
      description = d;
   }

}
```

```
Shirt myShirt;
myShirt = new Shirt ('G', 199.0, "T-Shirt" );
```



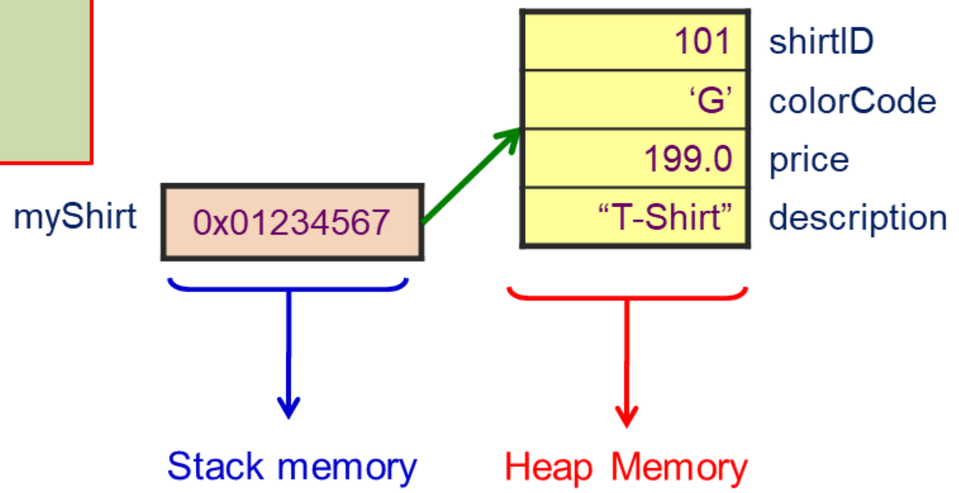
物件建構流程-儲存物件參考



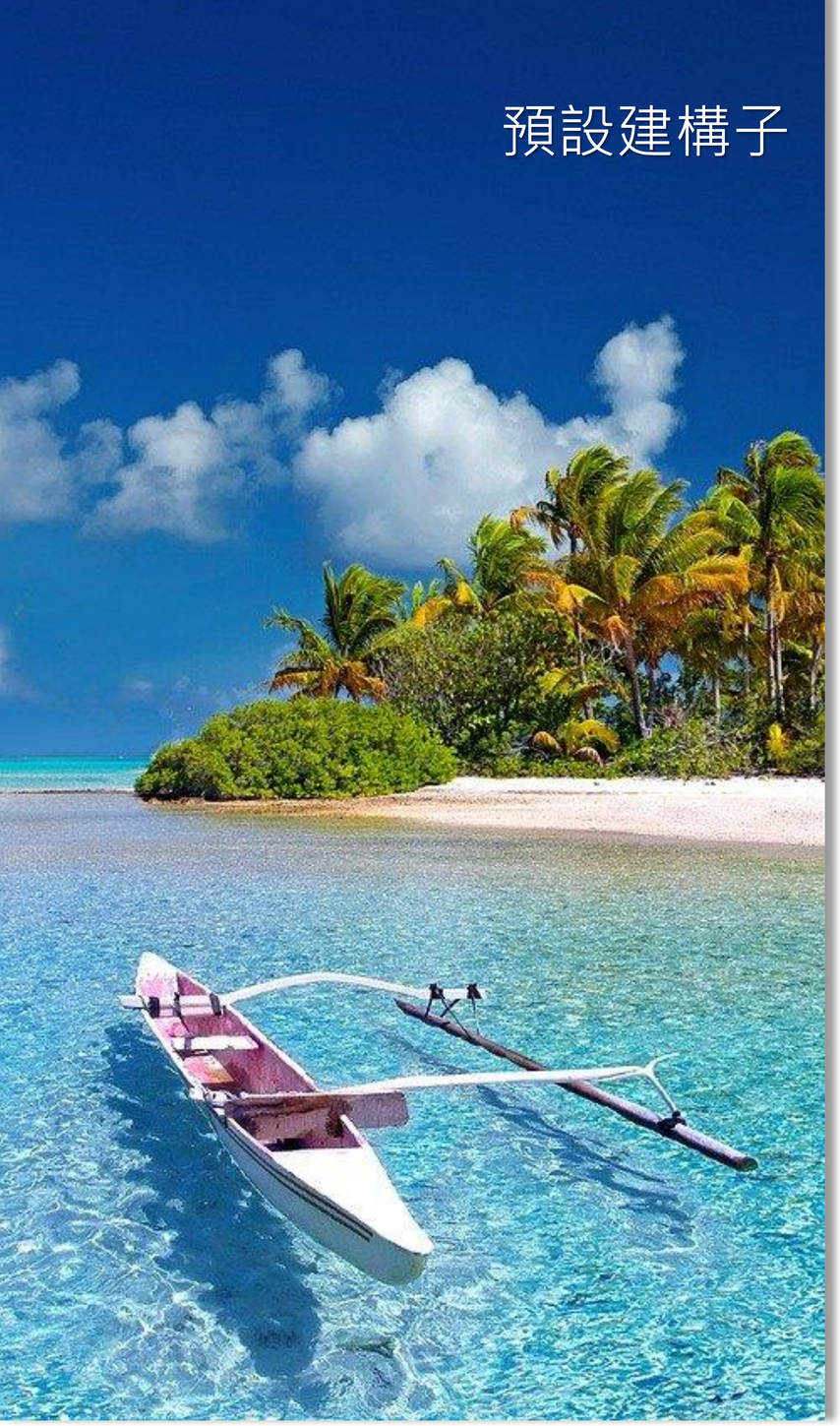
```
public class Shirt {
   private int shirtID = 101;
   private char colorCode = 'R';
   private double price = 299.0;
   private String description = "Polo Shirt";

   public Shirt(char c, double p, String d){
      colorCode = c;
      price = p;
      description = d;
   }

}
```

Shirt myShirt;

myShirt = new Shirt('G', 199.0, "T-Shirt");

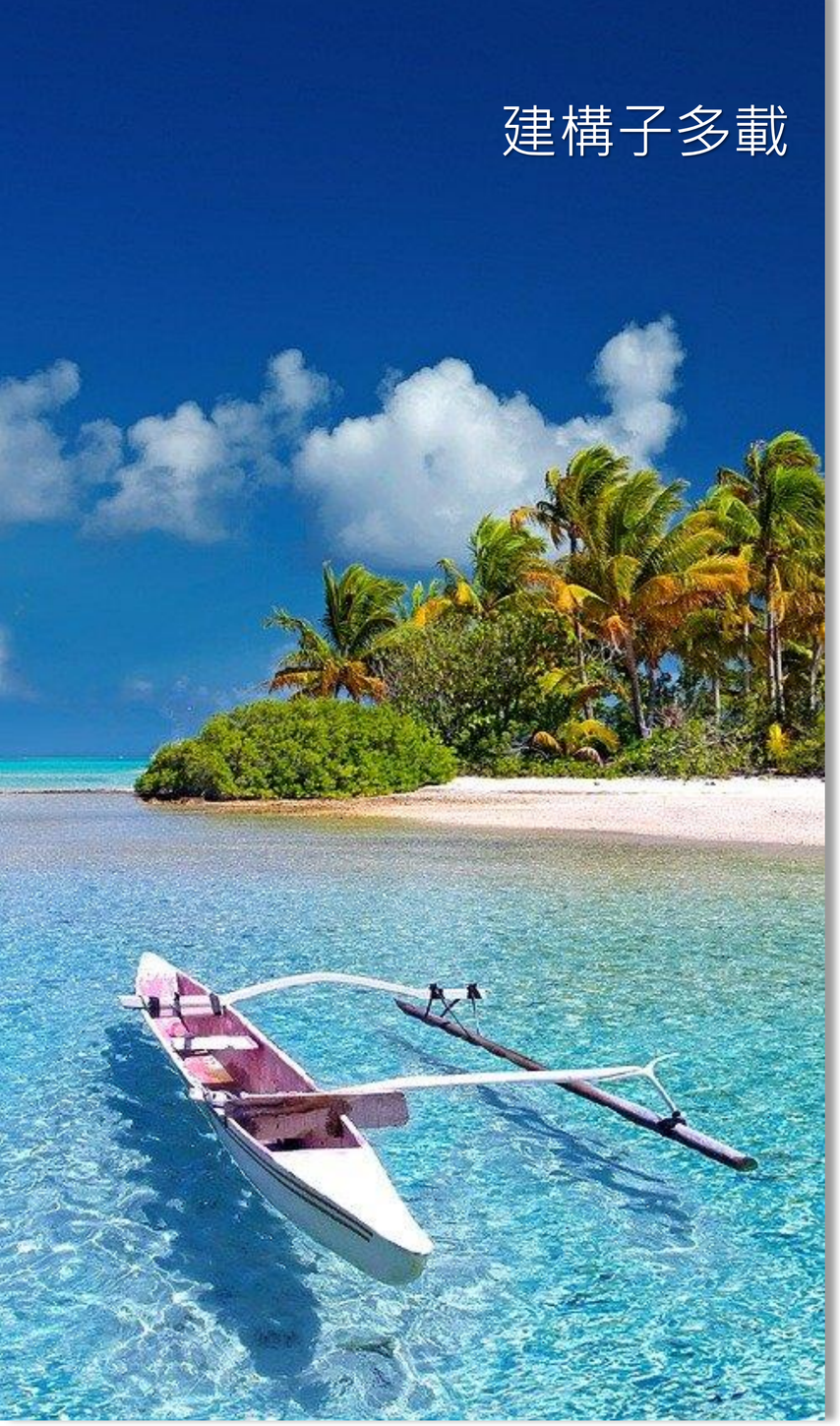


預設建構子



- 類別中若沒有定義建構子，編譯時期自動加入預設構構子
  - 預設建構子沒有參數列 (no arguments)
  - 存取權限與類別權限相同
  - 預設建構子沒有其他的程式敘述 (no body statement)
    - 除了繼承時隱含的 super( ) 呼叫之外
  - 自行定義建構子，即無預設建構子

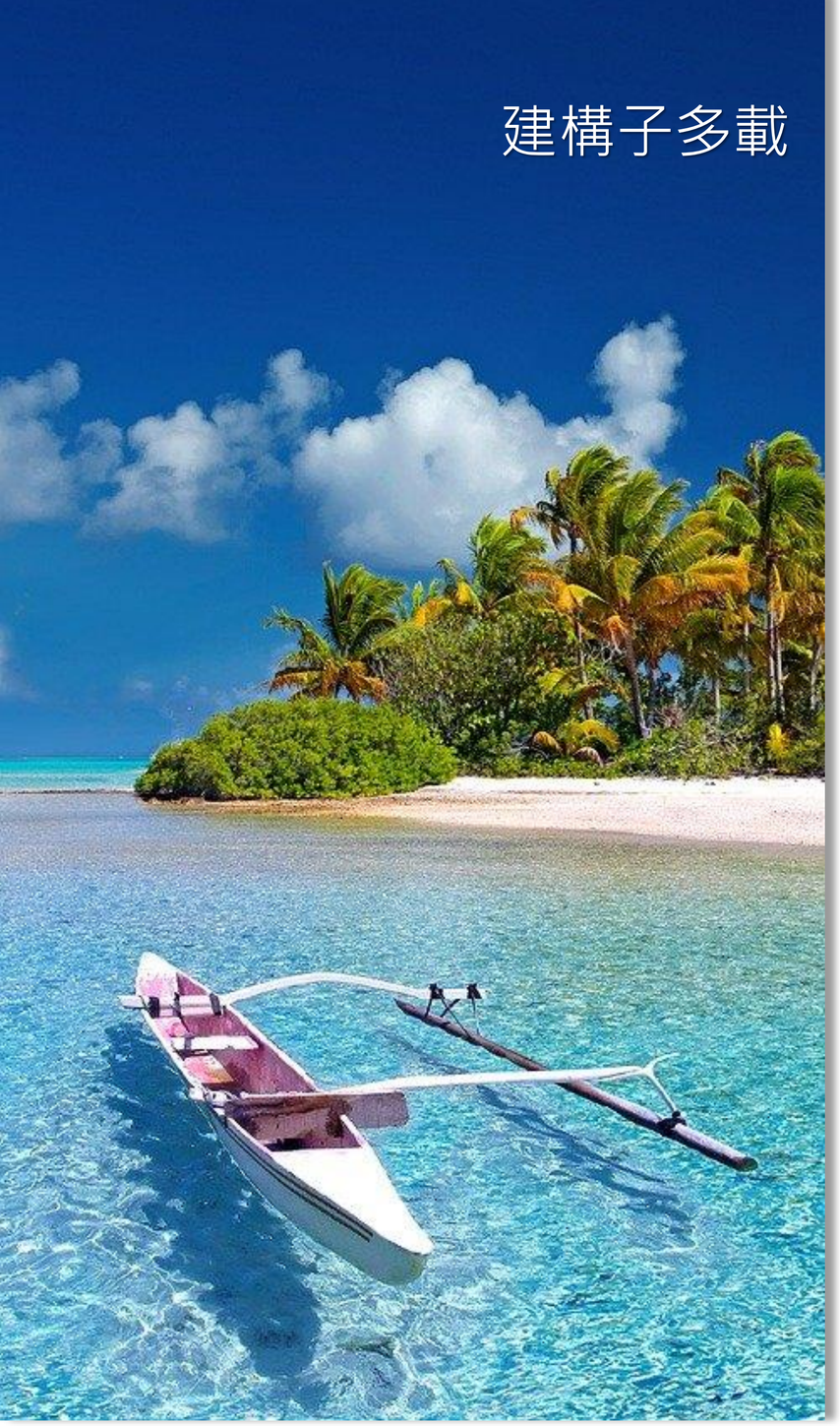
建構子多載



一

- 建構子多載 Constructors overloading
  - 物件的屬性欄位
    - 必要性欄位
    - 選擇性欄位
  - 提供多組建構子為物件設定初值
  - 傳入參數數量或型態不同

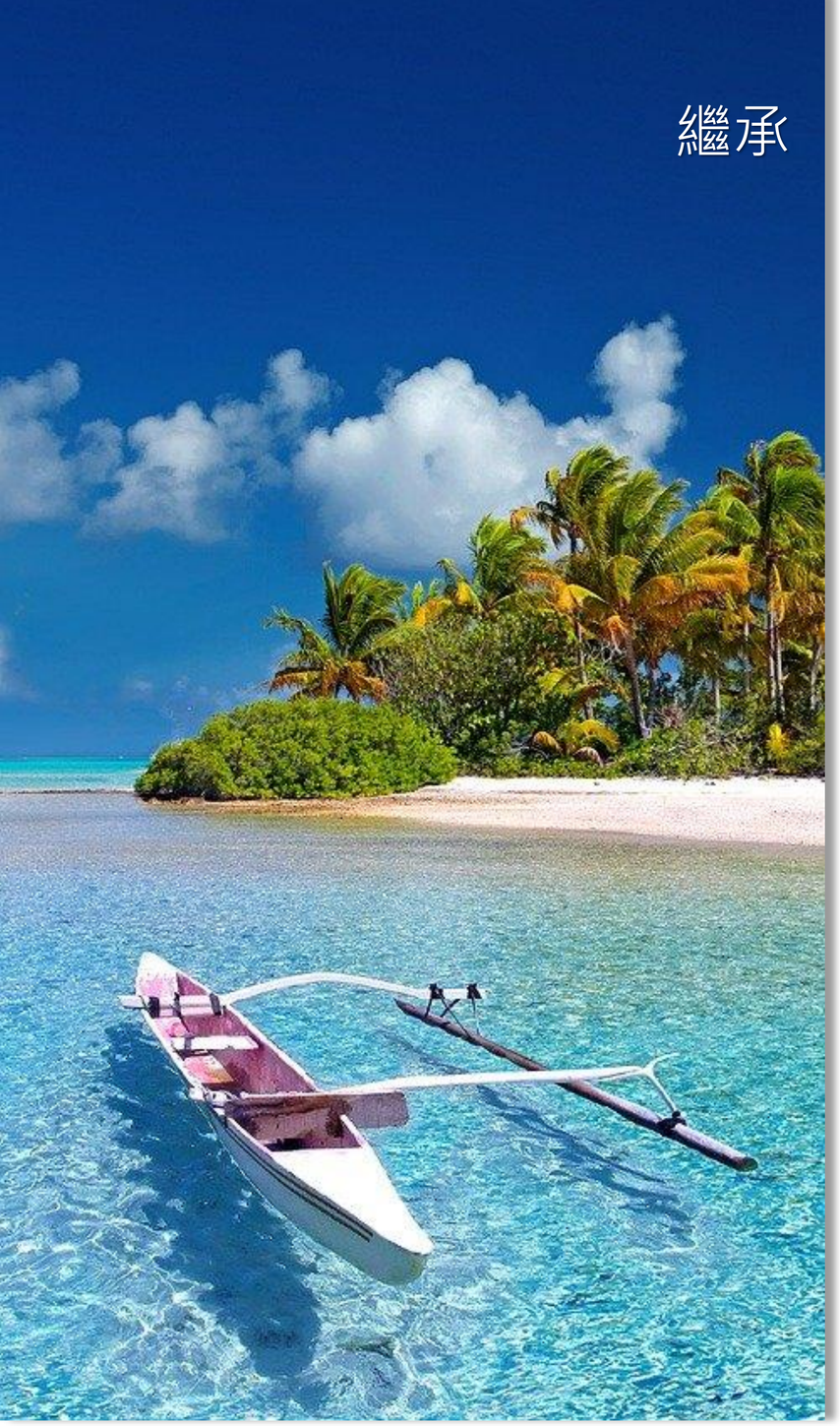
建構子多載



```
01  public class Shirt {
02      private int shirtID = 101;
03      private char colorCode = 'R';
04      private double price = 299.0;
05      private String description = "Polo Shirt";
06
07      public Shirt(int id) {
08          shirtID = id;
09      }
10      public Shirt(char color, double newPrice) {
11          colorCode = color;
12          price = newPrice;
13      }
14      public Shirt(char color, double newPrice, String desc) {
15          colorCode = color;
16          price = newPrice;
17          description = desc;
18      }
19  }
20
```

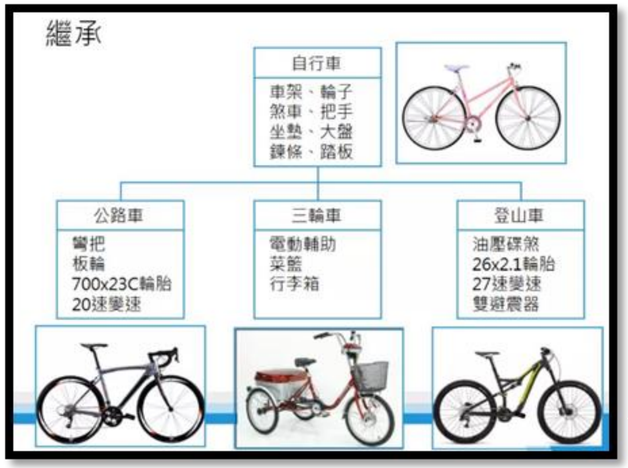
```
01  public class TestShirt {
02      public static void main(String[] args) {
03
04          Shirt s1 = new Shirt( );
05
06          Shirt s2 = new Shirt(101);
07
08          Shirt s3 = new Shirt('G', 600.0);
09
10          Shirt s4 = new Shirt('Y', 199.0, "T-Shirt");
11
12      }
13  }
```

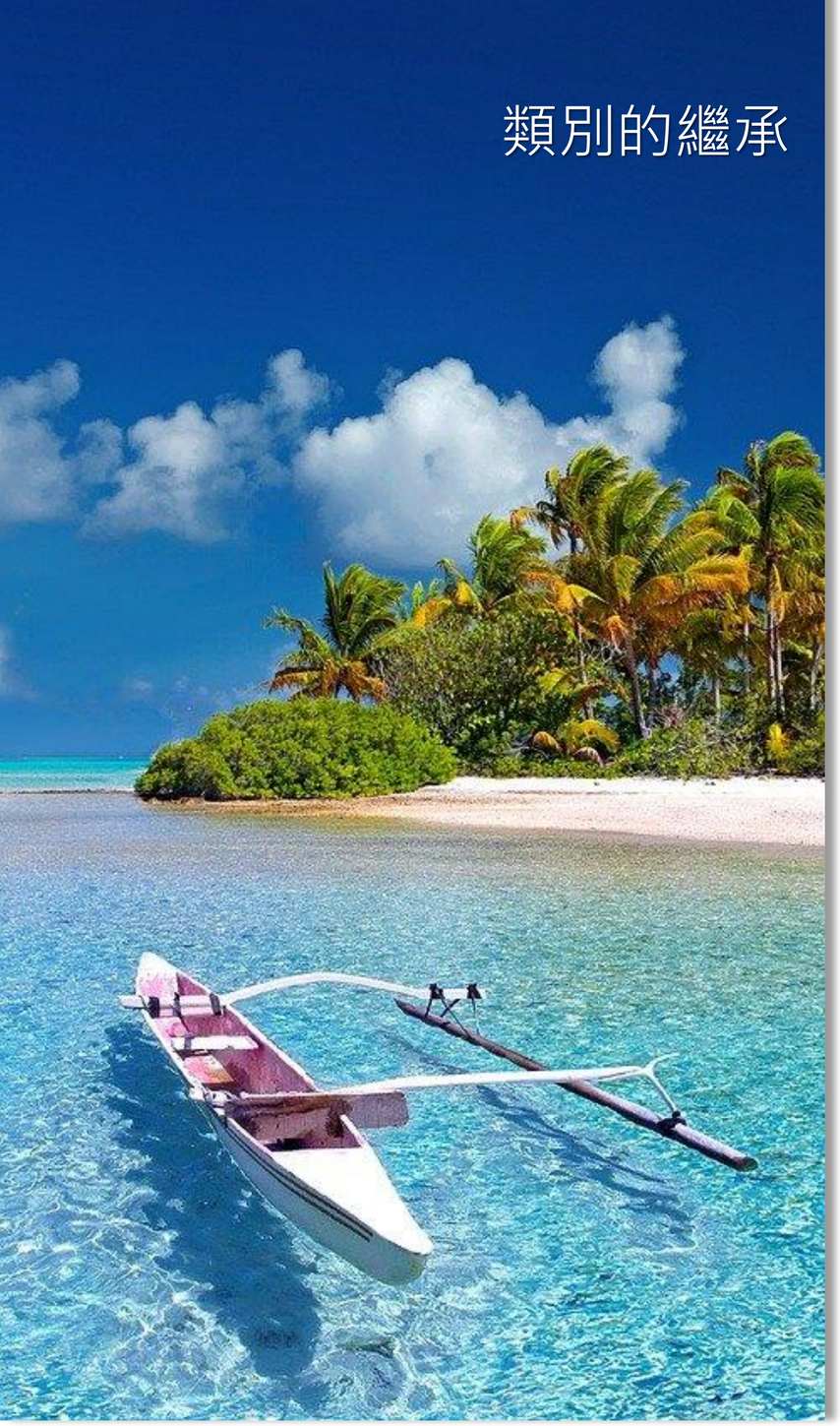
繼承 INHERITANCE



繼承

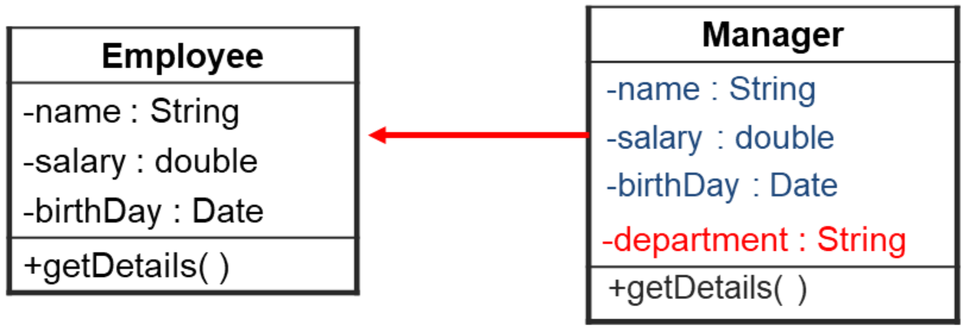
- 傳統宣告三個類別有共同的屬性如 車架，輪子，煞車……，若要修改煞車則需要三個類別都更改，如此很浪費時間，且程式也變得冗長
- 若改成下成之繼承，將相同的屬性獨立出來為父類別，而不同的屬性則變為子類別，如此若要更改煞車，只需更改父類別之屬性即可





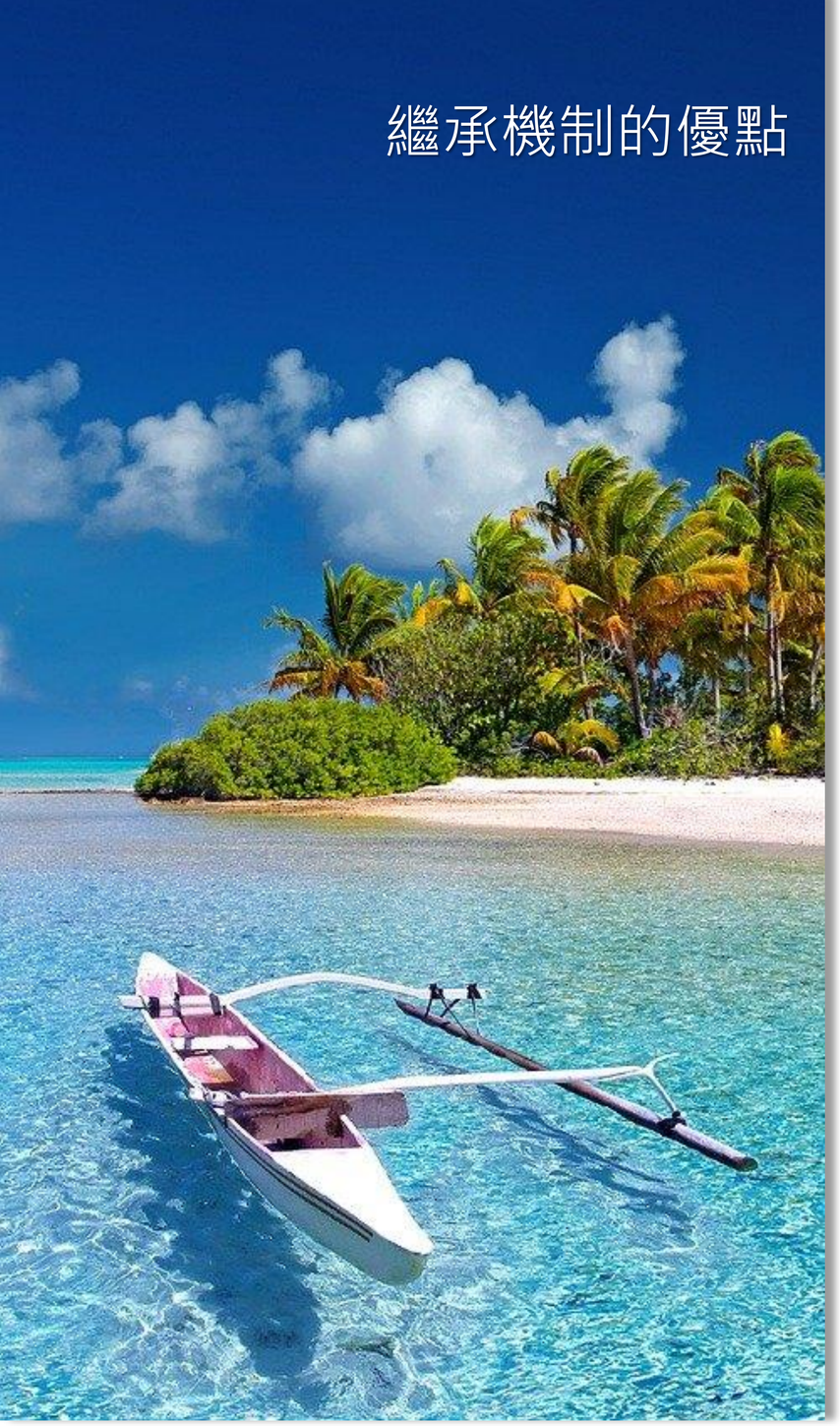
類別的繼承

- 繼承讓類別的程式碼可以延伸及重複使用
  - 父類別 base class/superclass
  - 子類別 derived class/subclass
  - 類別延伸 (extends class) 的關係是 " is a" 的概念。

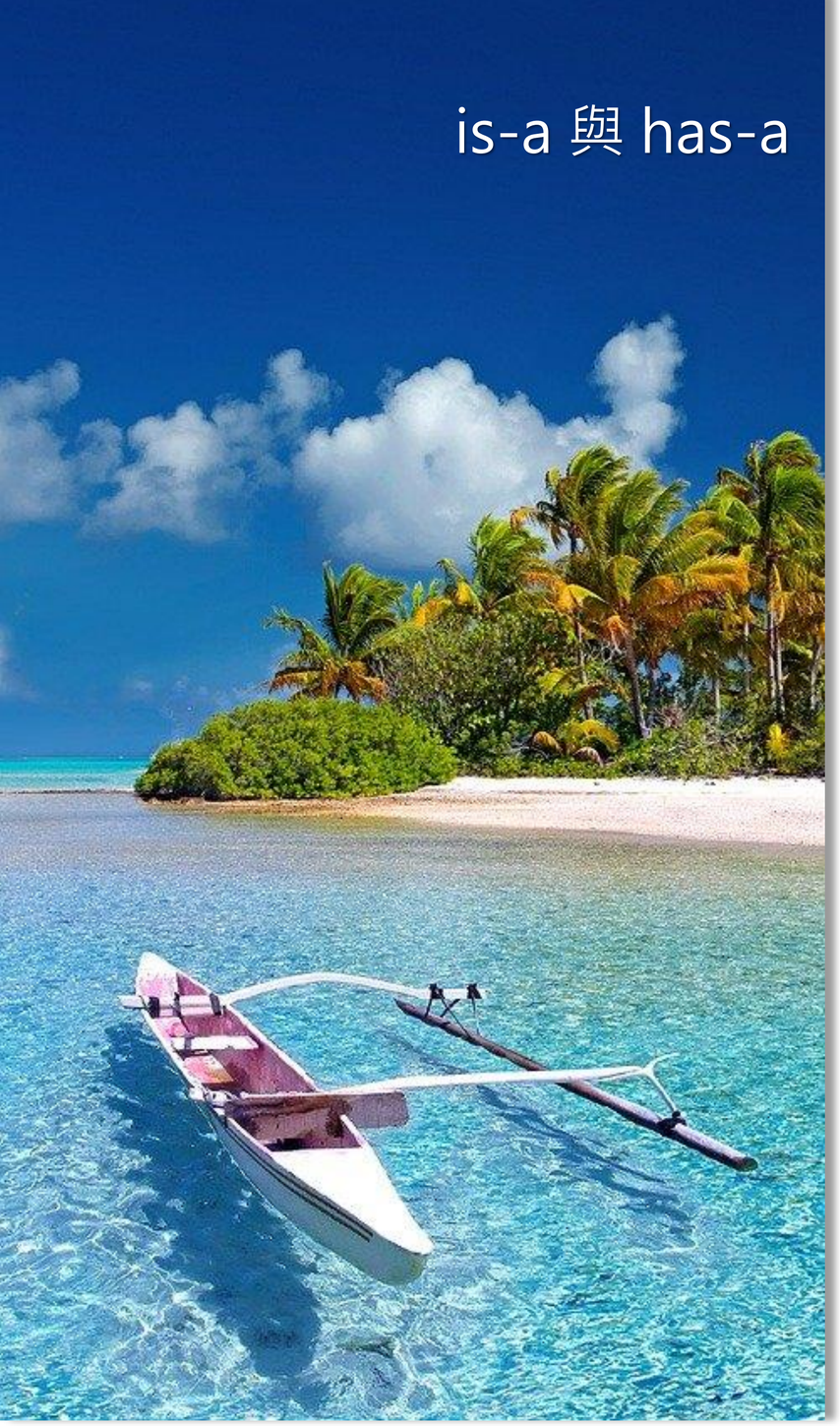


A Manager is a Employee

繼承機制的優點

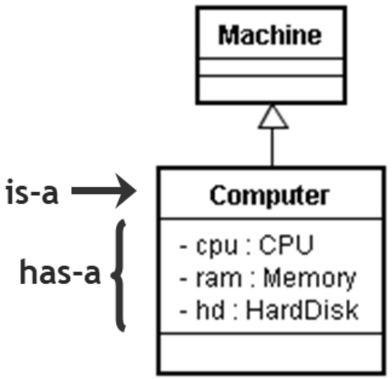


- 產生更特殊性的型別
- 消除重複的程式碼
- 提高程式的可維護性

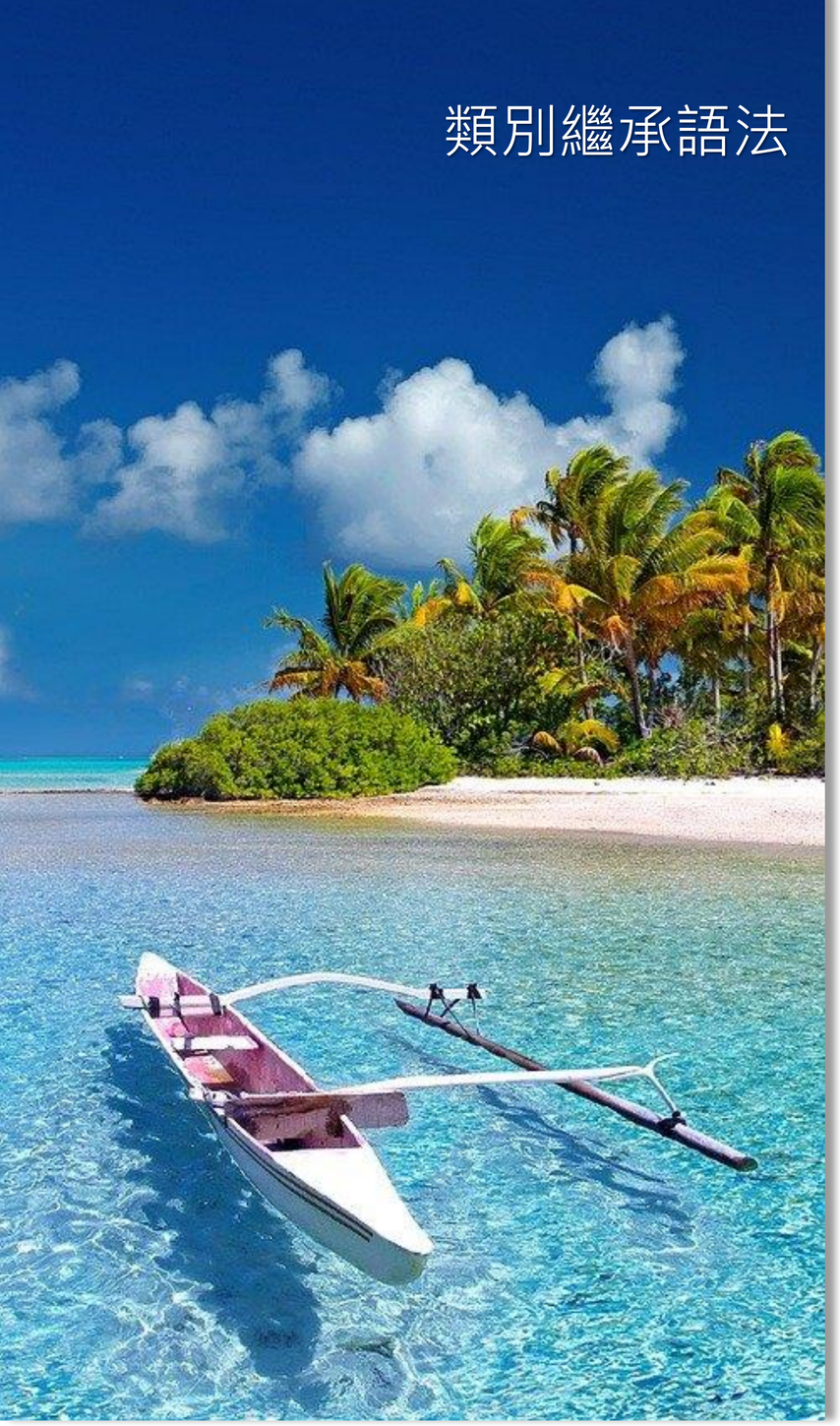


is-a 與 has-a

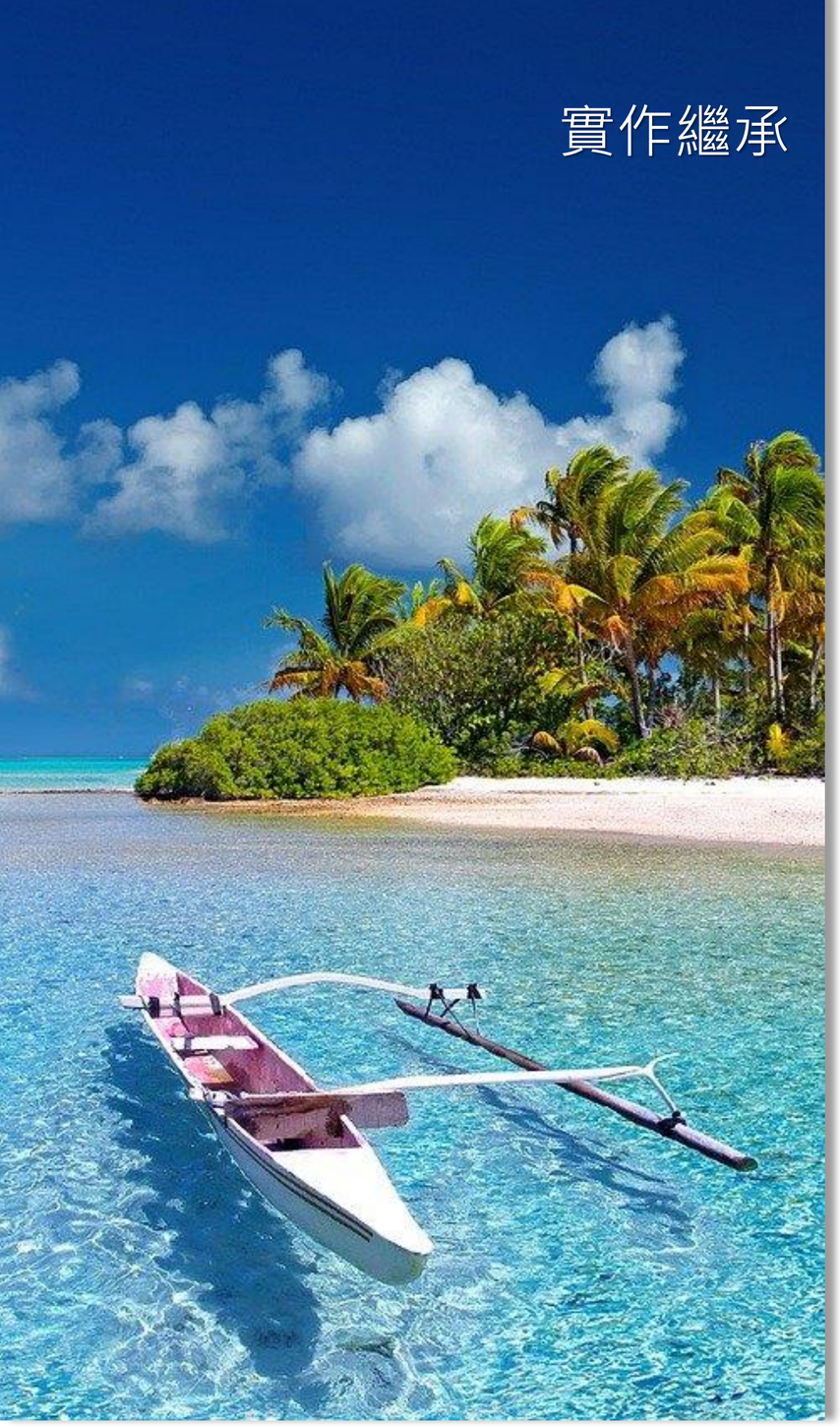
- is a (是一個)
  - 延伸的關係。
  - Java 語言利用 extends 關鍵字來實作
  - EX : Computer 是一種 Machine
    - Computer 類別繼承了 Machine 類別。
    - Machine 是父類別，而 Computer 則是子類別。
- has a (有一個)
  - 聚合的關係。
  - 類別中的成員變數 (member variable) 來表示。
  - EX : 電腦中有 CPU、256M RAM、40GB HD
    - CPU、RAM 與 HD 便成了電腦的成員變數



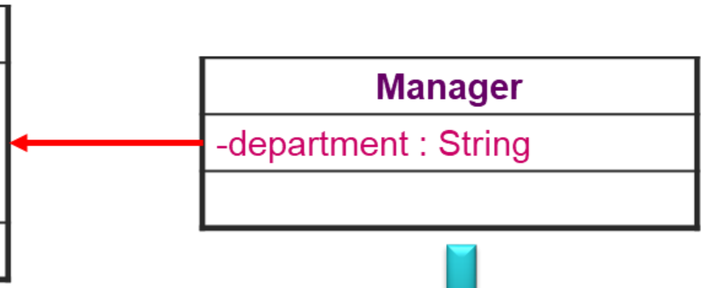
類別繼承語法



```
[modifiers] class 子類別名稱 extends 父類別名稱 {
        //類別內容
    }
```



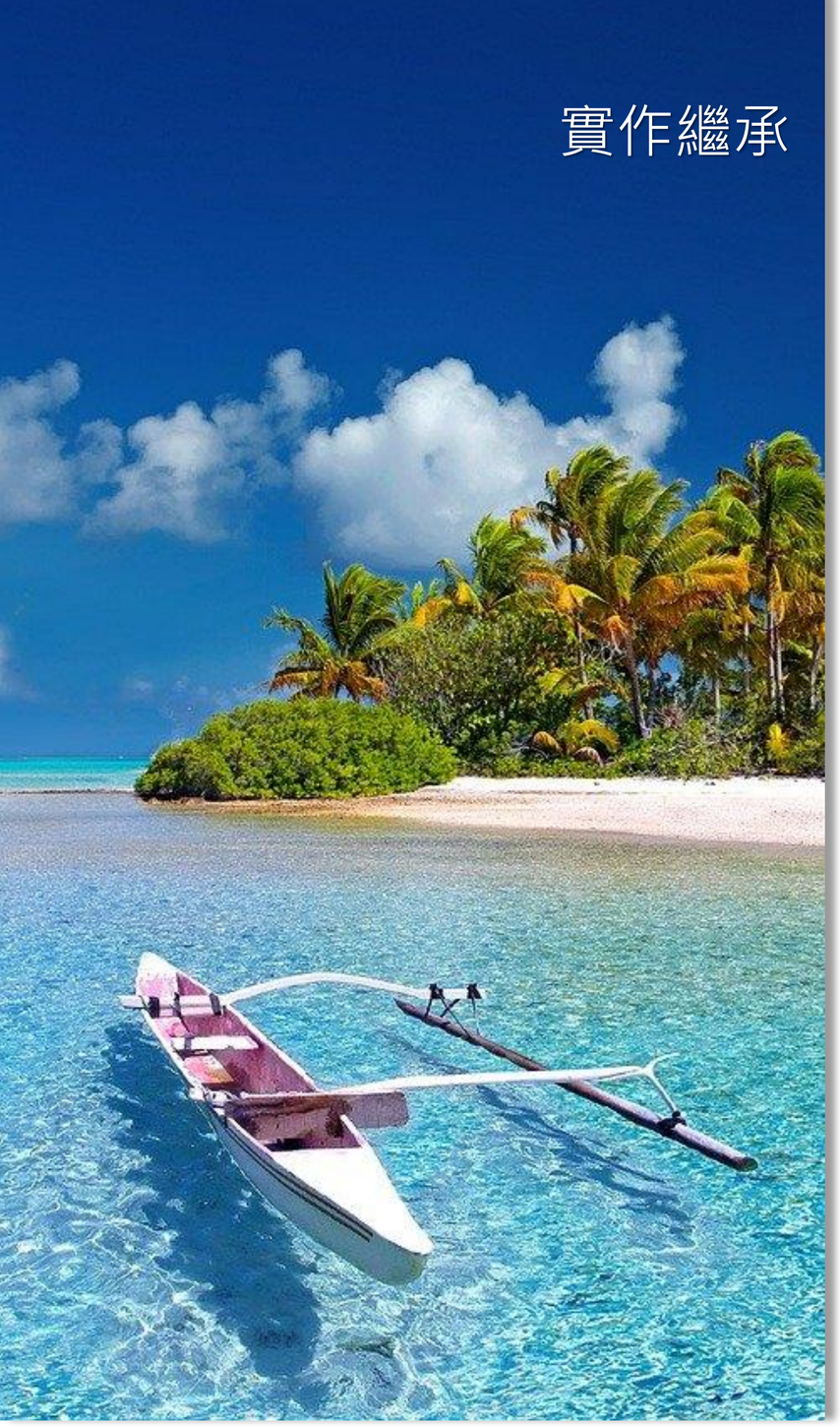
實作繼承



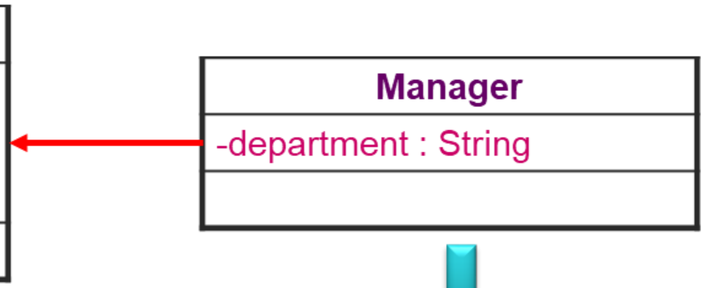

```
01  public class Employee {
02      private String name = “Sean”;
03      private double salary = 10000;
04      public void getDetails( ) {
05          System.out.println(“Name:” + name);
06          System.out.println(“Salary:” + salary);
07      }
08  }
```

```
01  public class Manager extends Employee {
02      private String dept = “EDU”;
03
04  }
05
```

Manager 繼承了 Employee 所有成員;
屬性:name, salary, birthDay
方法:getDetails( ) 方法



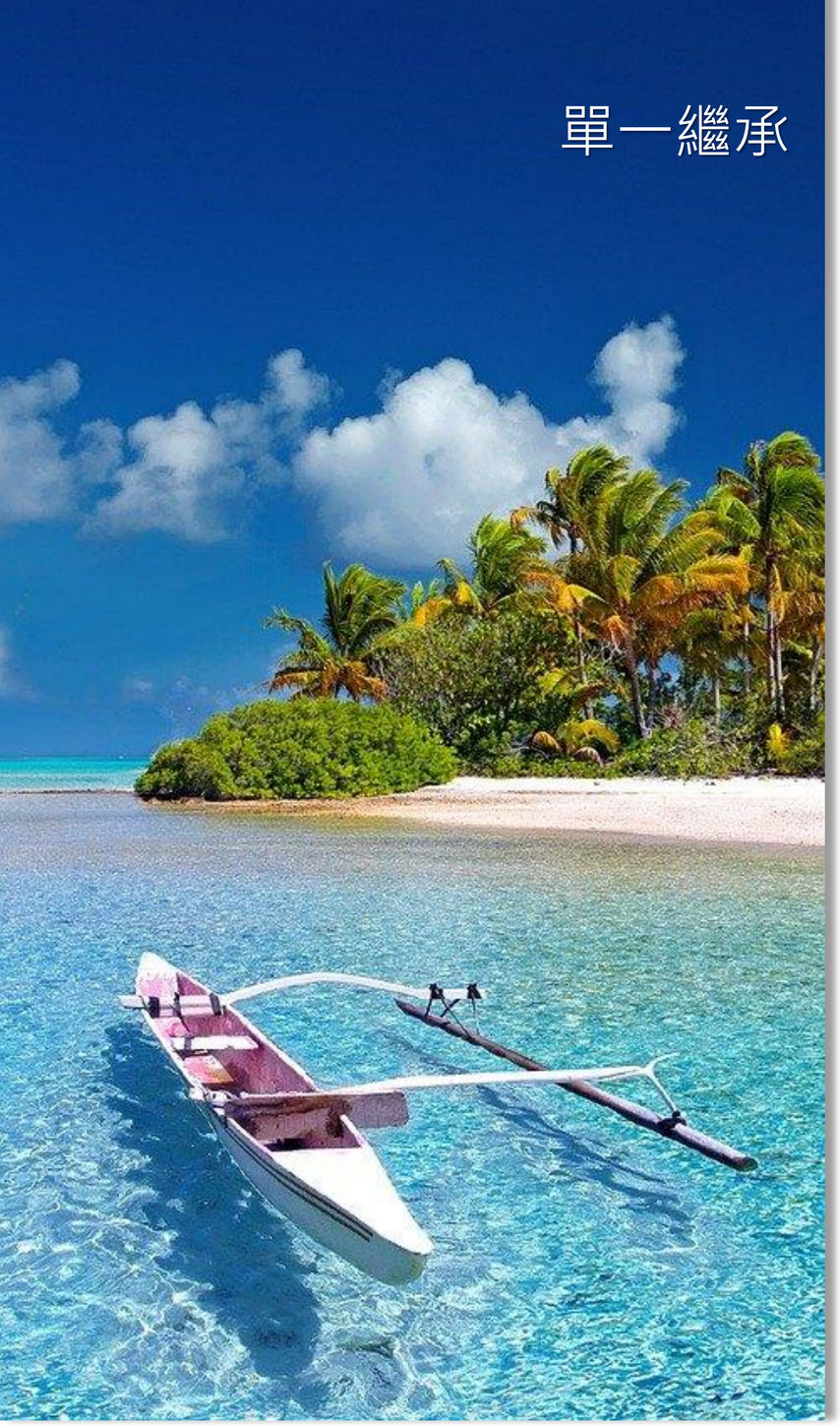
實作繼承




```
01  public class Employee {
02      private String name = “Sean”;
03      private double salary = 10000;
04      public void getDetails( ) {
05          System.out.println(“Name:” + name);
06          System.out.println(“Salary:” + salary);
07      }
08  }
```

```
01  public class Manager extends Employee {
02      private String dept = “EDU”;
03
04  }
05
```

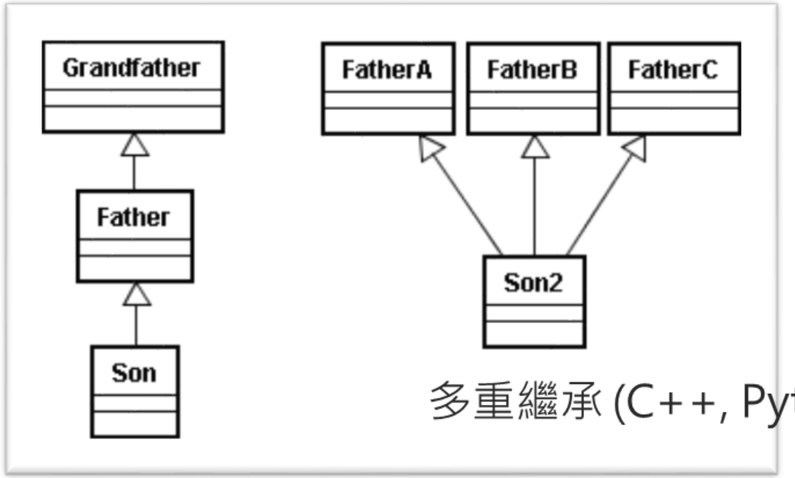
Manager 繼承了 Employee 所有成員;
屬性:name, salary, birthDay
方法:getDetails( ) 方法



單一繼承

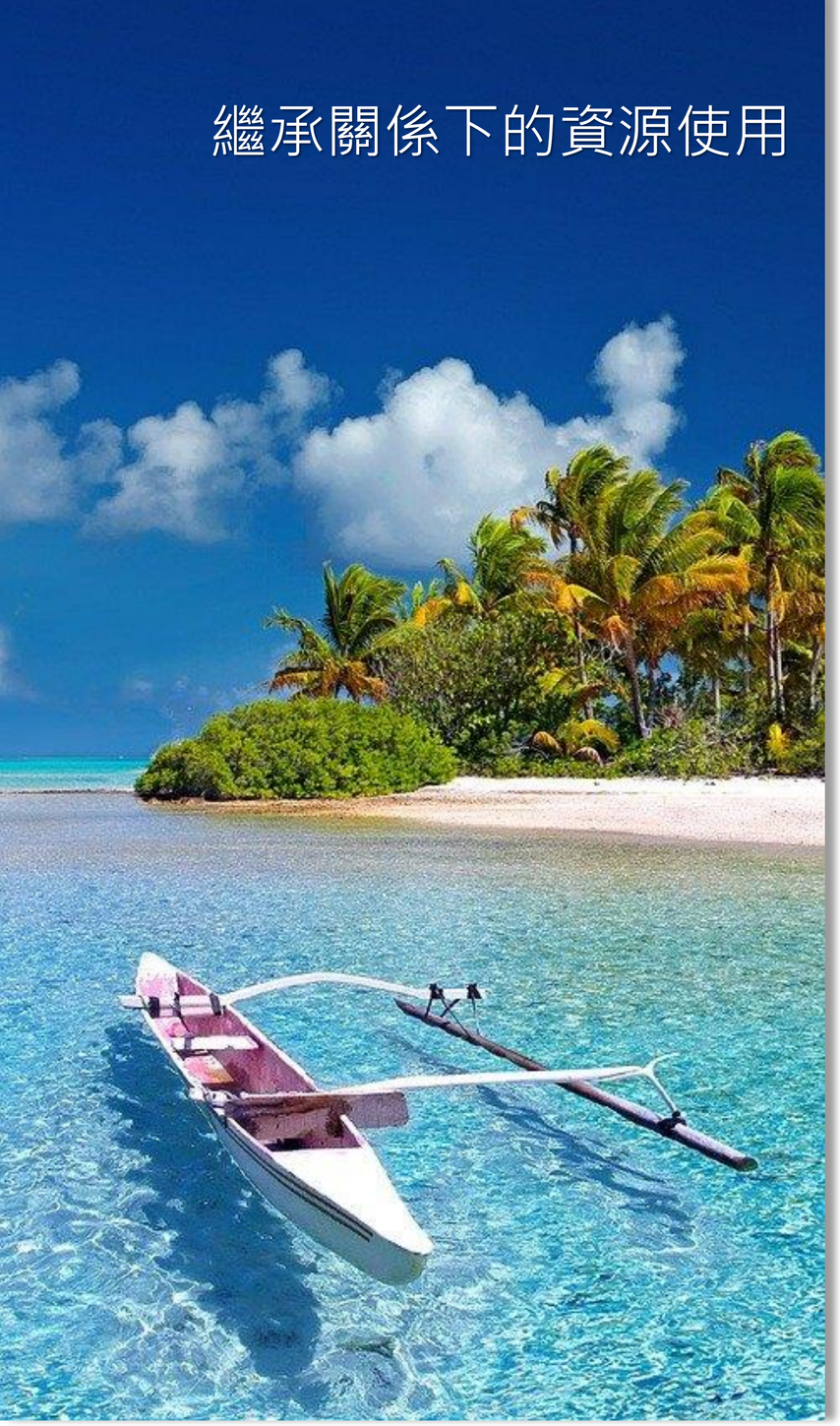
- Java 語言在繼承上只允許單一繼承 (Single Inheritance) 關係。
  - 子類別在定義繼承的關係時，只能針對單一父類別做延伸，不能同時使用來自多個父類別的資源。

單一繼承 (Java)

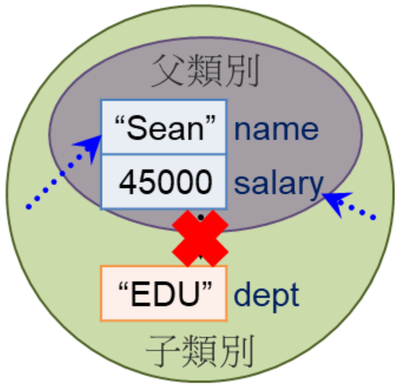


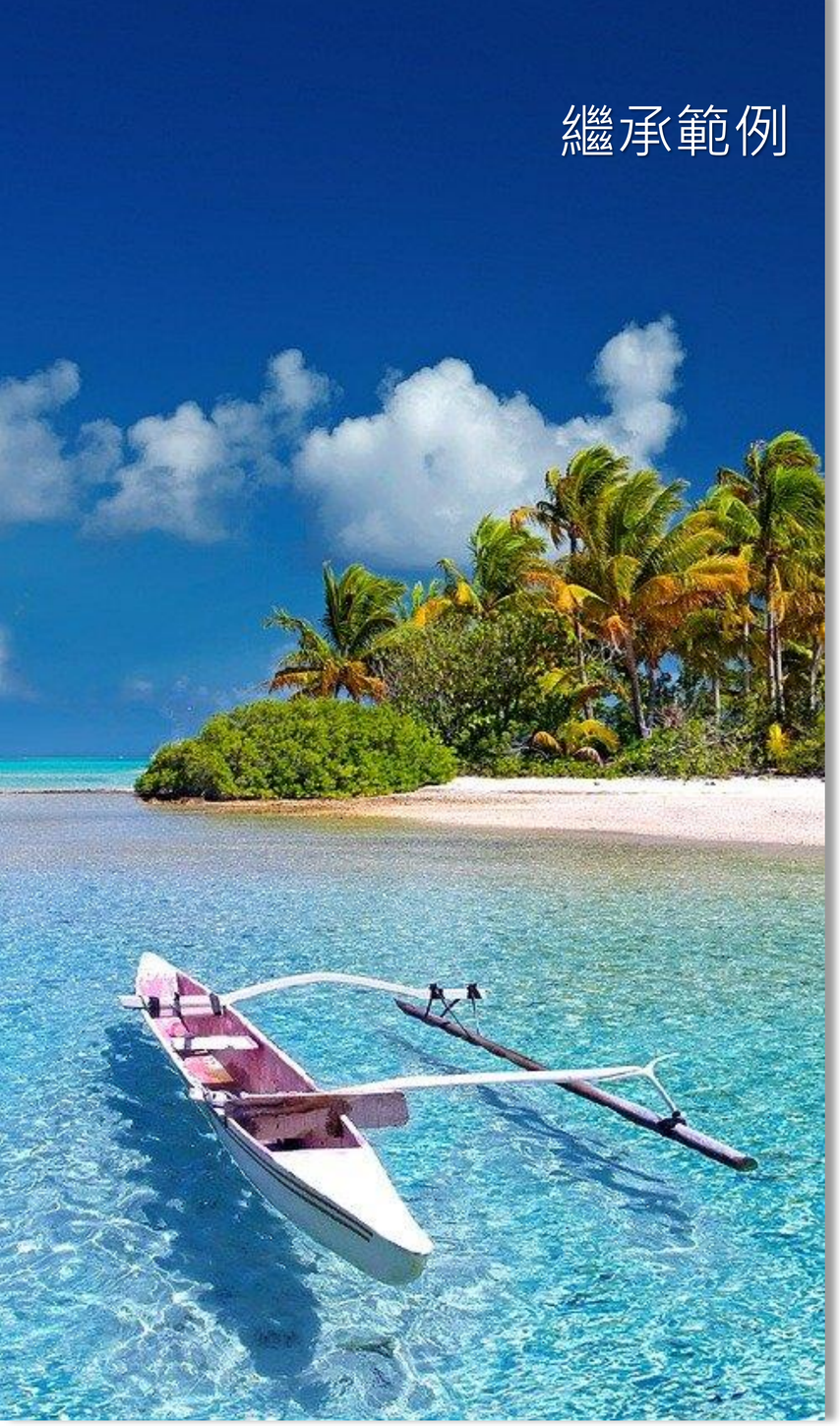
多重繼承 (C++, Python等)

繼承關係下的資源使用



- 當子類別繼承父類別，擁有父類別中的資源
  - 子類別繼承 (擁有) 父類別成員 (屬性和方法)
    - 實作：子類別中包含父類別
  - 父類別不擁有子類別屬性和方法
  - 建構子不會被繼承
- 子類別存取父類別資源
  - 存取父類別的屬性：直接使用屬性名稱;
  - 呼叫父類別的方法：直接使用方法名稱(參數列);
  - 但存取權限仍受限於父類別的存取修飾字





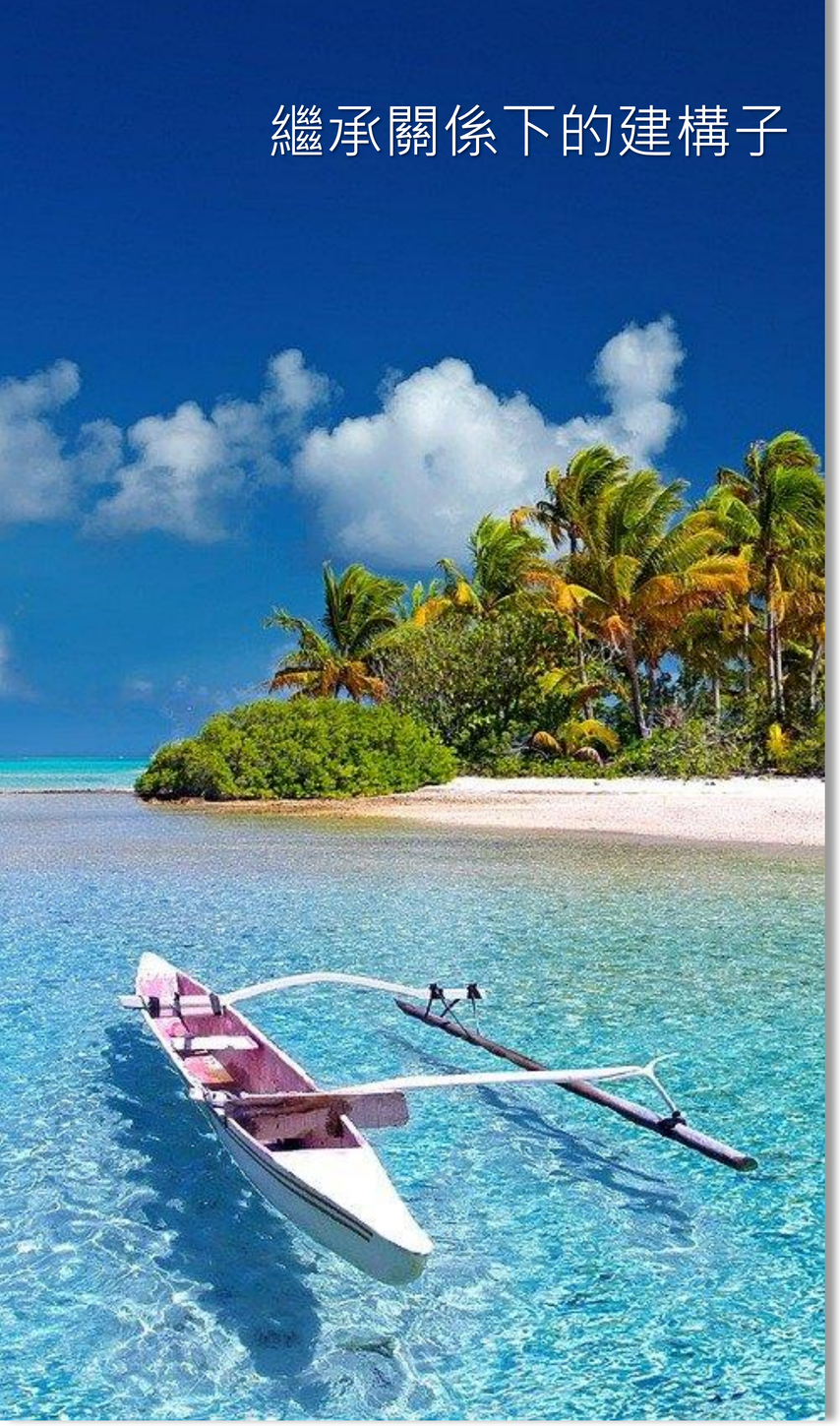
繼承範例

```
01  class Father {
02      public int money = 1000000;
03      public void undertaking( ) {
04          System.out.println("父親的事業");
05      }
06  }
```

```
01  class Son extends Father{
02
03  }
```

```
01  public class Extends {
02      public static void main(String[] args){
03          Son son = new Son( );
04          son.undertaking( );
05          System.out.println("金額:" + son.money);
06      }
07  }
```

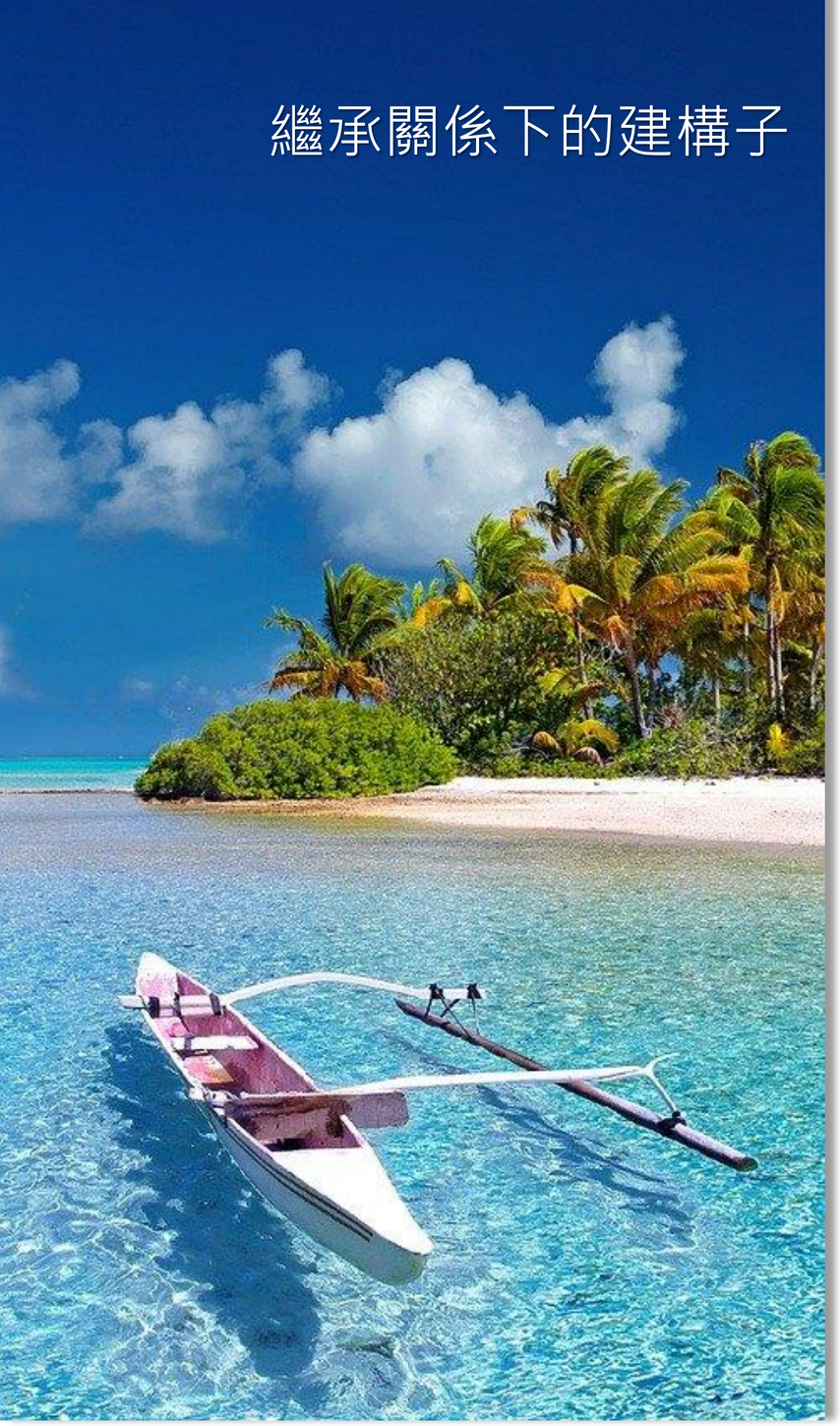
繼承關係下的建構子



一

- 子類別產生物件時,需先建構父類別物件
  - 先配置父類別所需記憶體，再配置子類別記憶體
  - 先初始化父類別的屬性，才執行子類別初始化
  - 先呼叫父類別建構式,再呼叫子類別建構式
- 預設建構子
  - 程式中若沒有定義建構子，在編譯時期會自動加入
  - 沒有參數列 (no arguments)
  - 包含 super( ) 語法，呼叫父類別無參數建構子

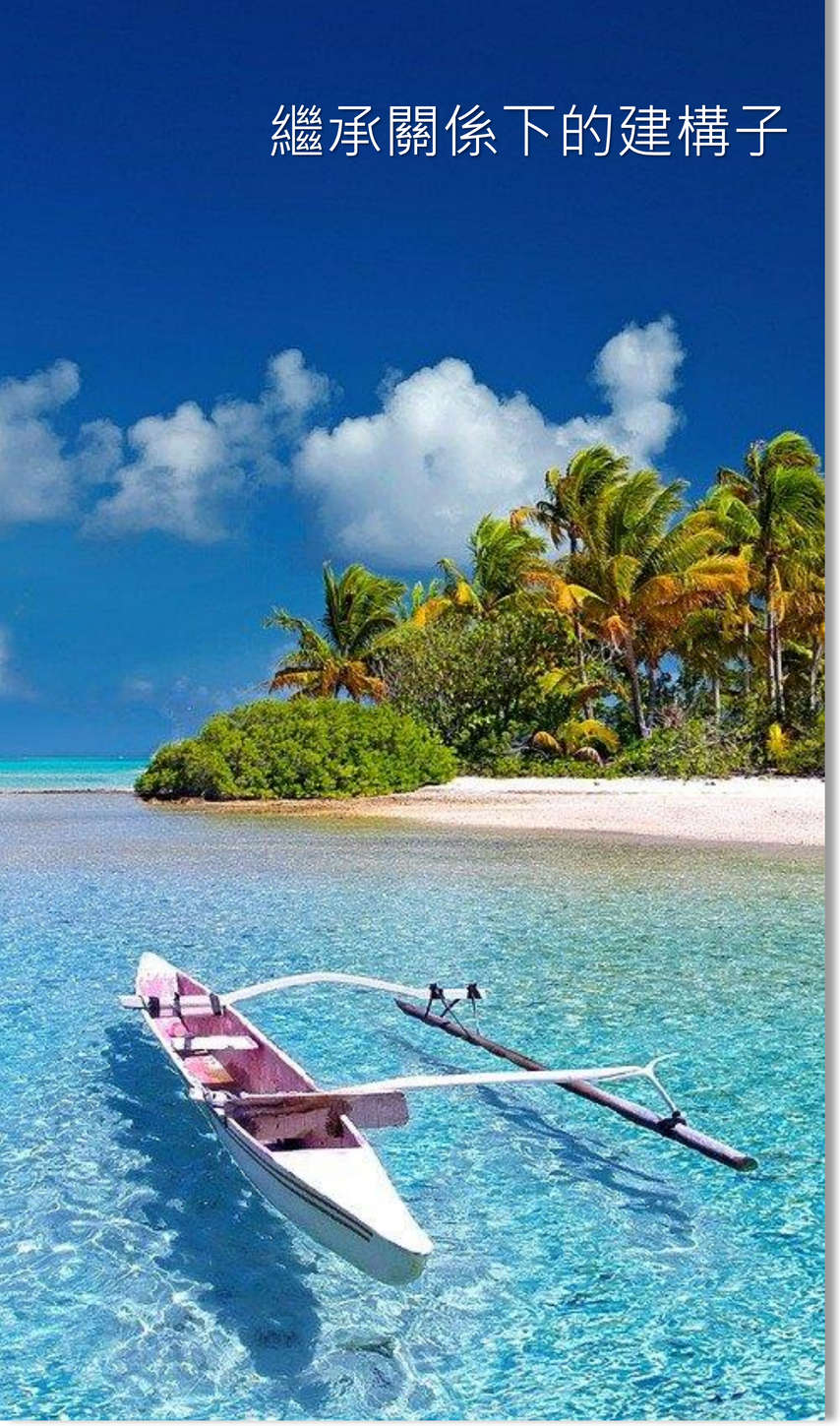
繼承關係下的建構子



一

- 指定建構子
  - super 與 this 關鍵字
    - super -> 父類別
    - this -> 本類別
  - super(參數列)
    - 子類別建構子中指定呼叫父類別的特定建構子
  - this(參數列)
    - 類別建構子中呼叫本類別其他建構子

繼承關係下的建構子

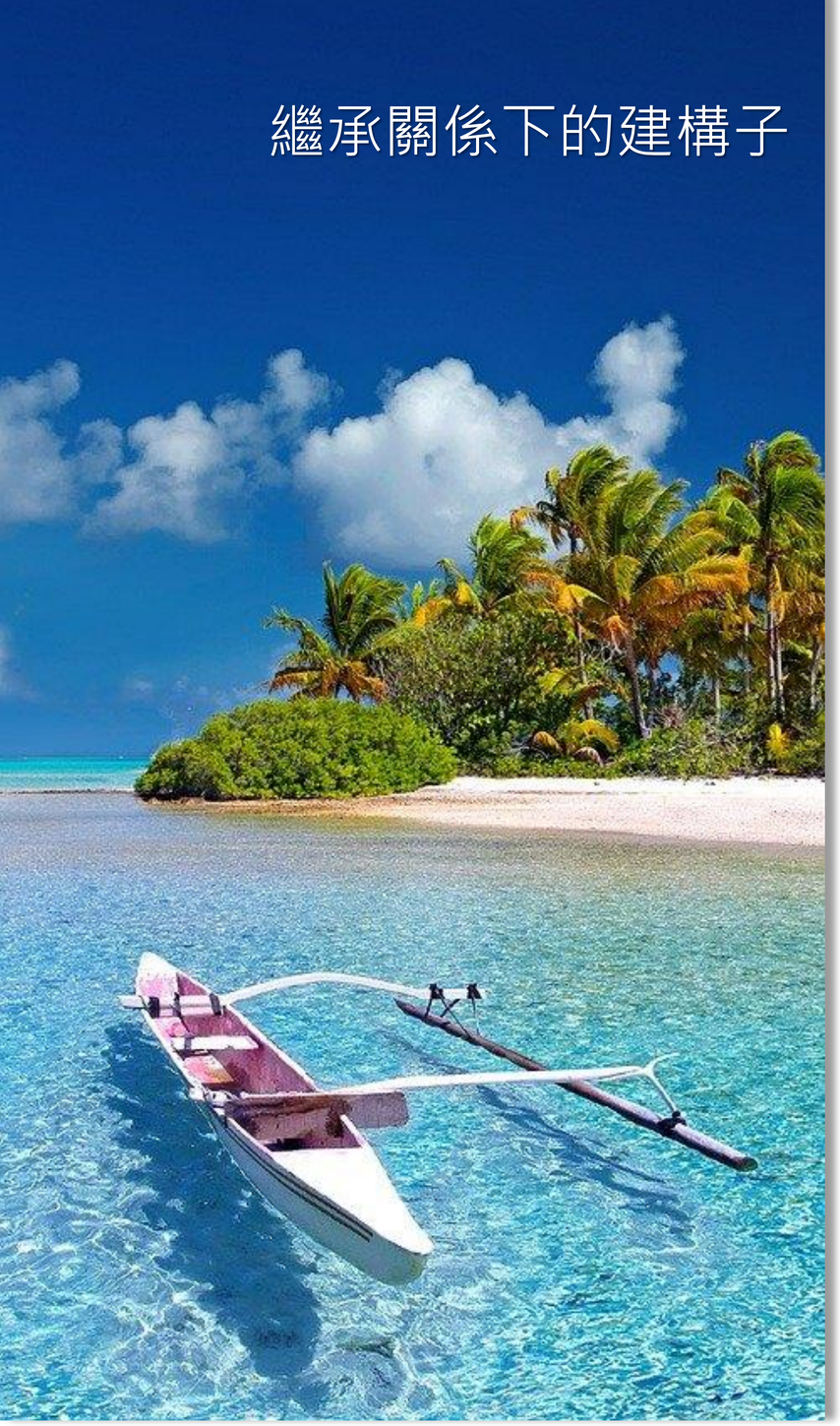


一

◆建構子使用 super( ) / this( ) 注意事項

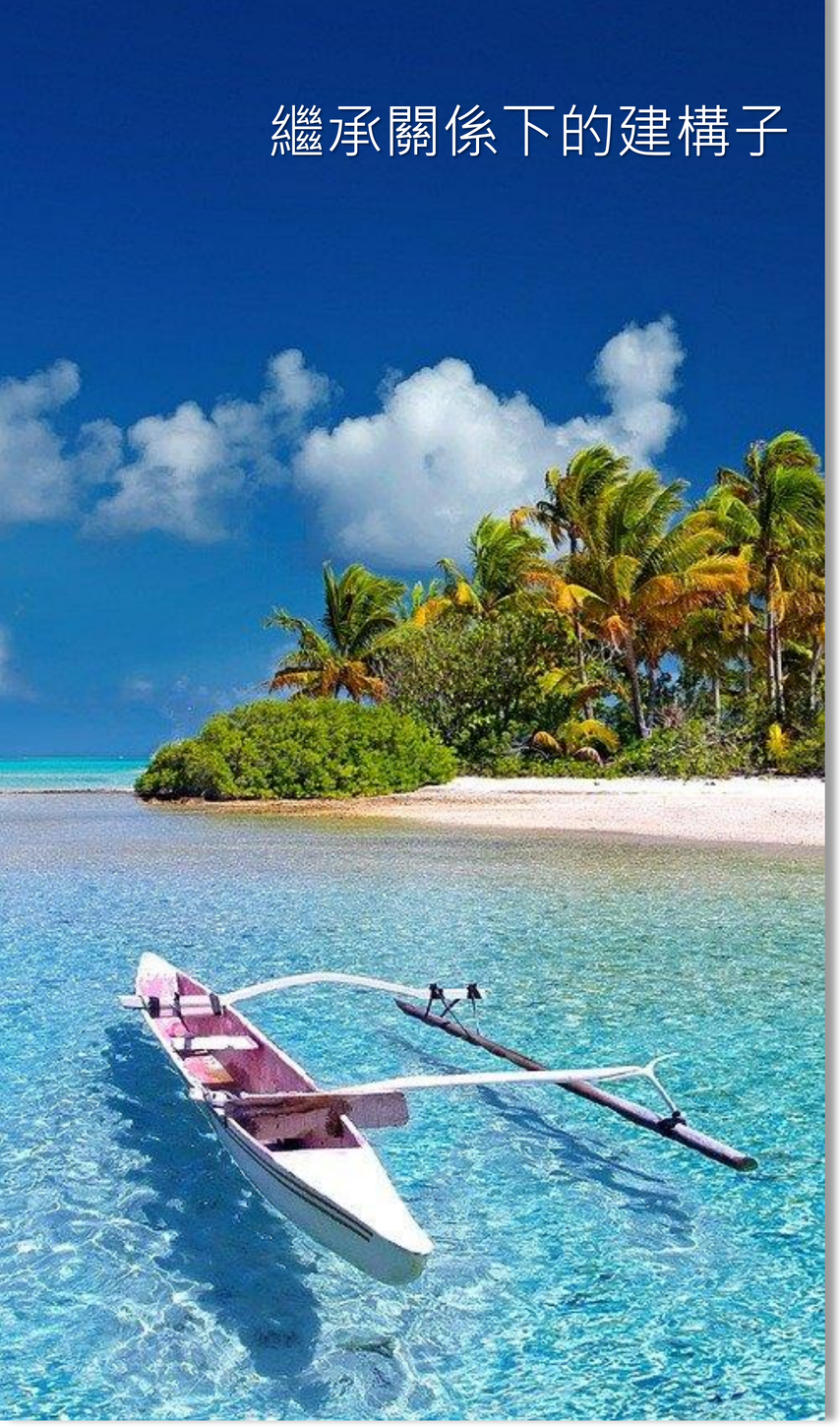
- 只能用在建構子之中，不能在程式其他位置出現
- 只能用在建構子程式碼第一行，一次只能使用一種
- 建構子之中，若沒有使用 super( )或this( )，編譯器自動在第一行加上 super( )

繼承關係下的建構子



- 父類別在編譯時會產生預設建構子
- 子類別在編譯時也會產生預設建構子
- super( ) 用來呼叫父類別預設建構子，this( )呼叫自己的建構子，在撰寫上必須撰寫在建構子實作區塊的第一行
  - 1.起初
    - class Father{ }
    - class Son extends Father{ }
  - 2.因為上述二個重點，所以要看成
    - class Father{ Father( ){ } }
    - class Son extends Father{ Son( ){ } }
  - 3.又由於Son類別繼承Father類別，所以在編譯時會在Son( )建構子的第一行預設加入 super( )，也就是呼叫父類別 Father( )的無參數建構子
    - class Father{ Father( ){ } }
    - class Son extends Father{
    - Son( )
  - {
  - super();//呼叫Father類別中無參數建構子
  - }
  - }

繼承關係下的建構子

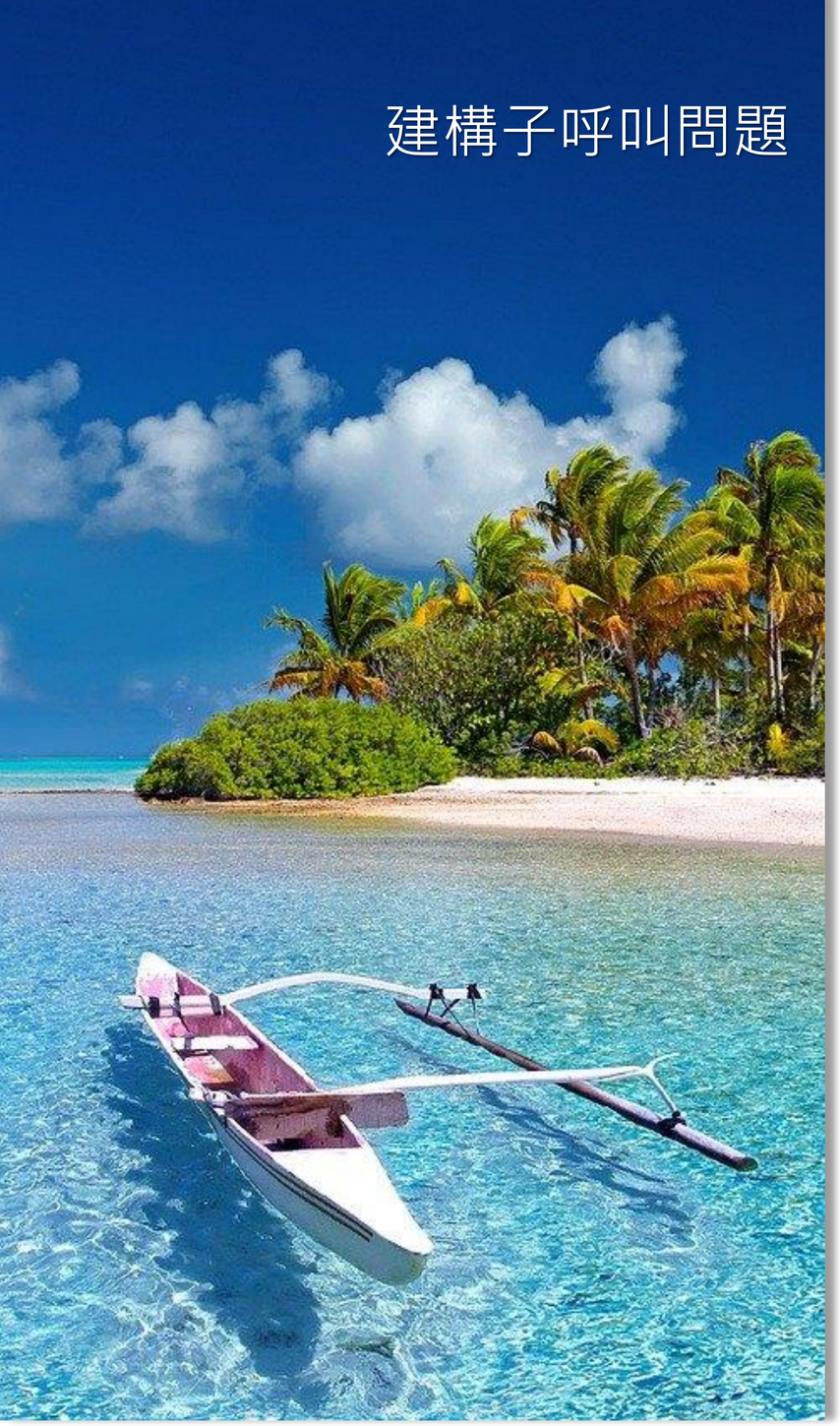


Manager m = new Manager ("Sean", 50000.0, "EDU");

```
public class Employee {
    private String name = "Sean";
    private double salary = 10000;
    public Employee( ){
    }
    public Employee(String n, double s) {
        name  = n;
        salary= s;
    }
}
```

```
public class Manager extends Employee {
    private String dept = "EDU";
    public Manager(String n, double s, String d) {
        super(n, s);
        dept = d;
    }
}
```

建構子呼叫問題



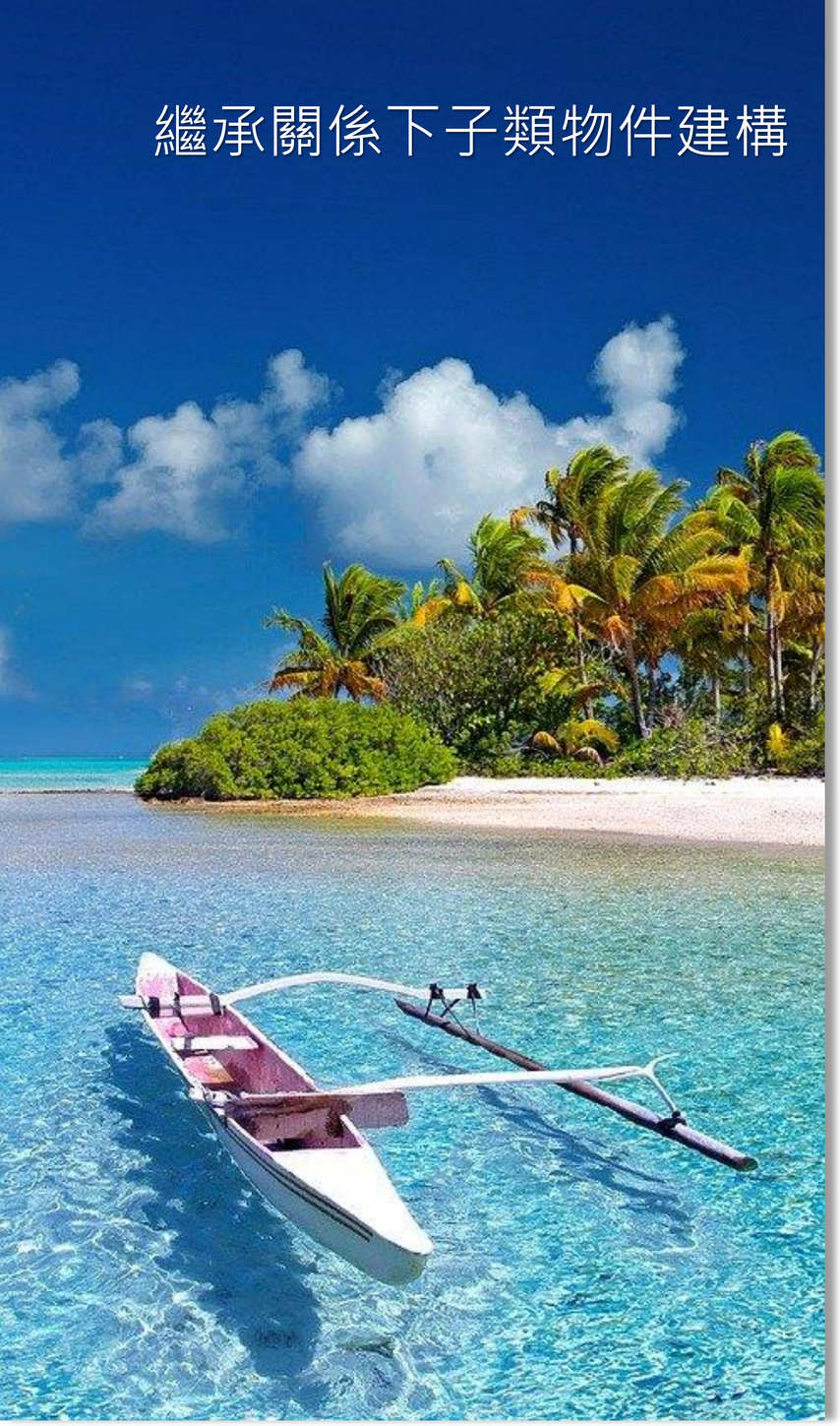
```
public class Employee {
    private String name = "Sean";
    private double salary = 10000;
    public Employee(String n, double s) {
        name  = n;
        salary= s;
    }
    public void getDetail( ) {
        System.out.println("Name:" + name);
        System.out.println("Salary:" + salary);
    }
}
```

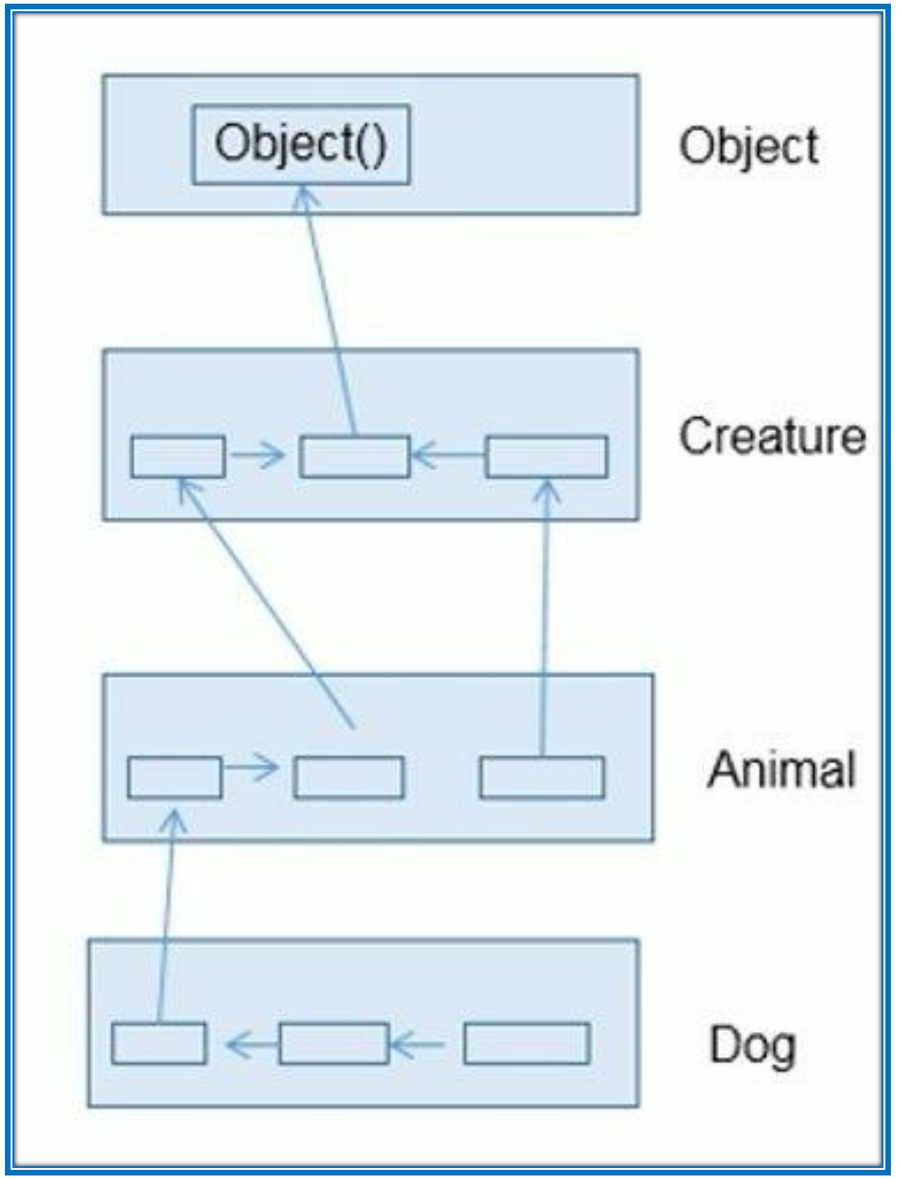
沒有無參數建構子

```
public class Manager extends Employee {
    private String dept = "EDU";
    public Manager( ) {
        super( );
    }
}
```

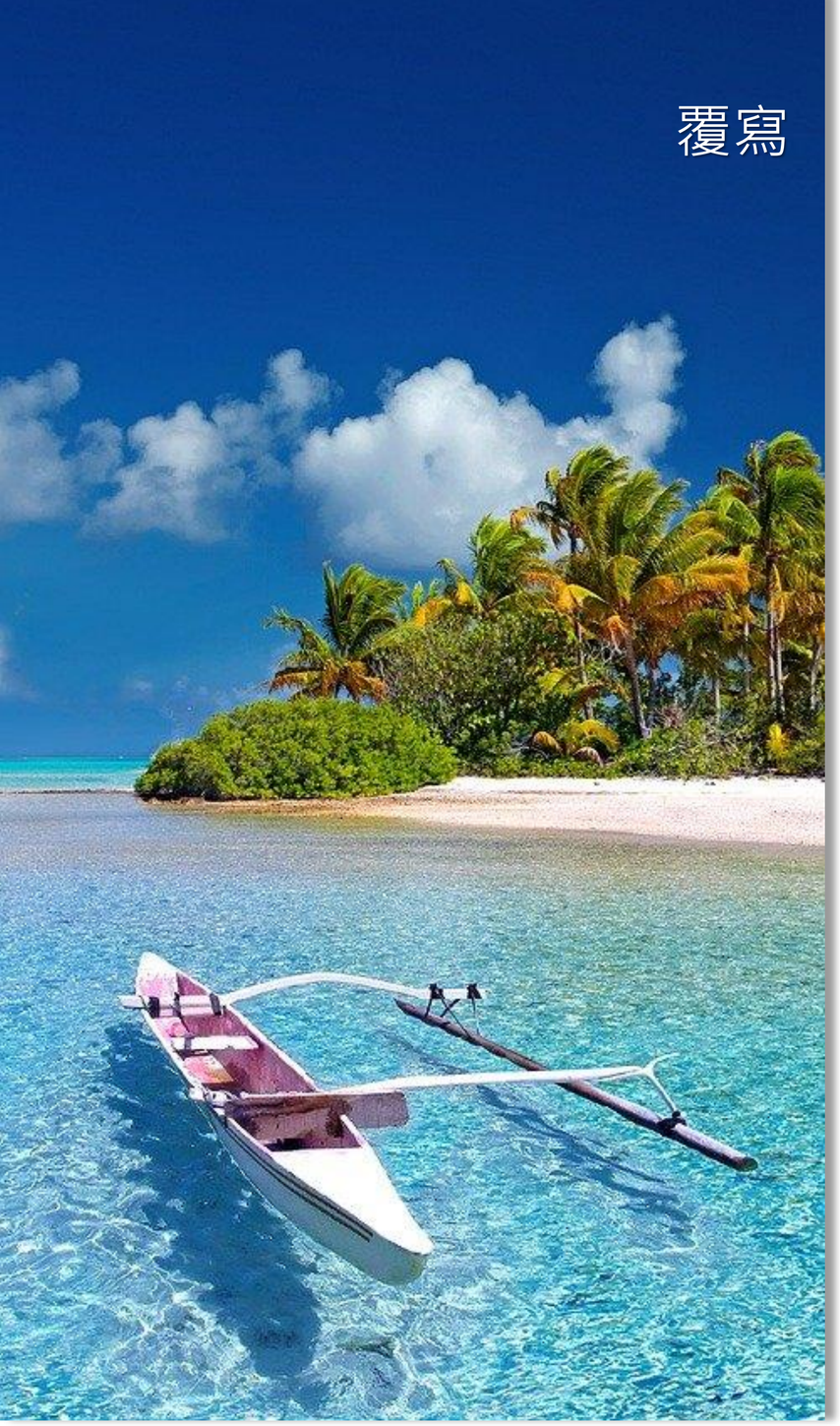
Manager m = new Manager( );

繼承關係下子類物件建構



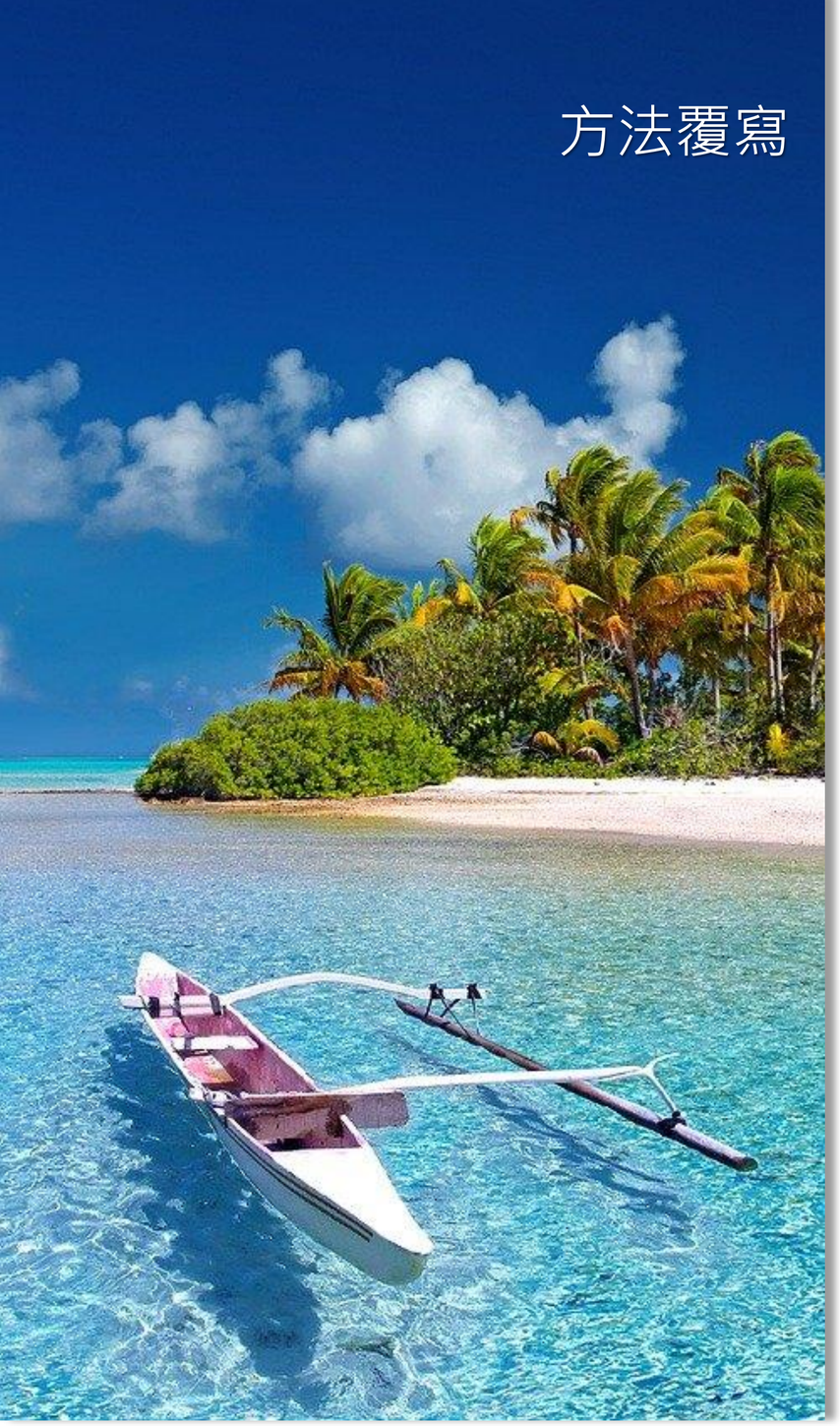


覆寫 OVERRIDE

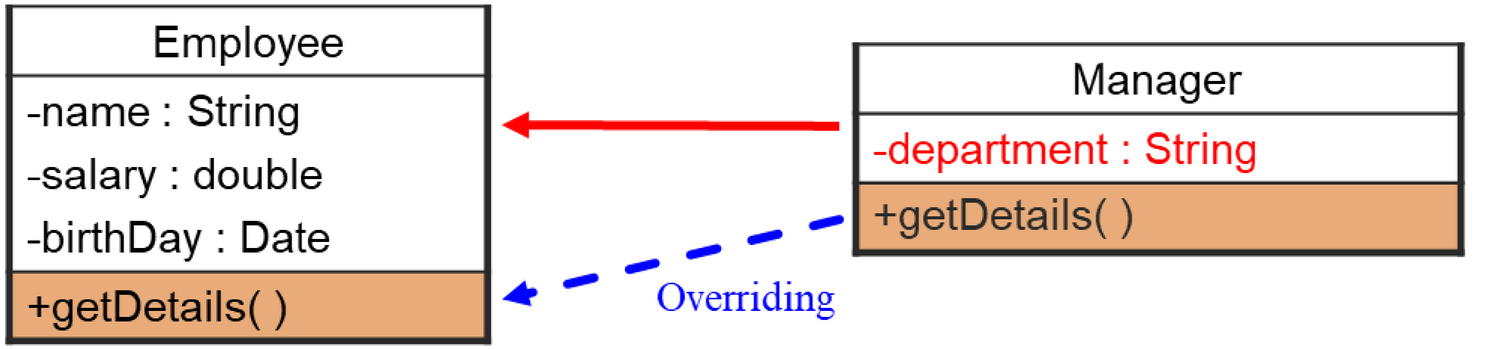


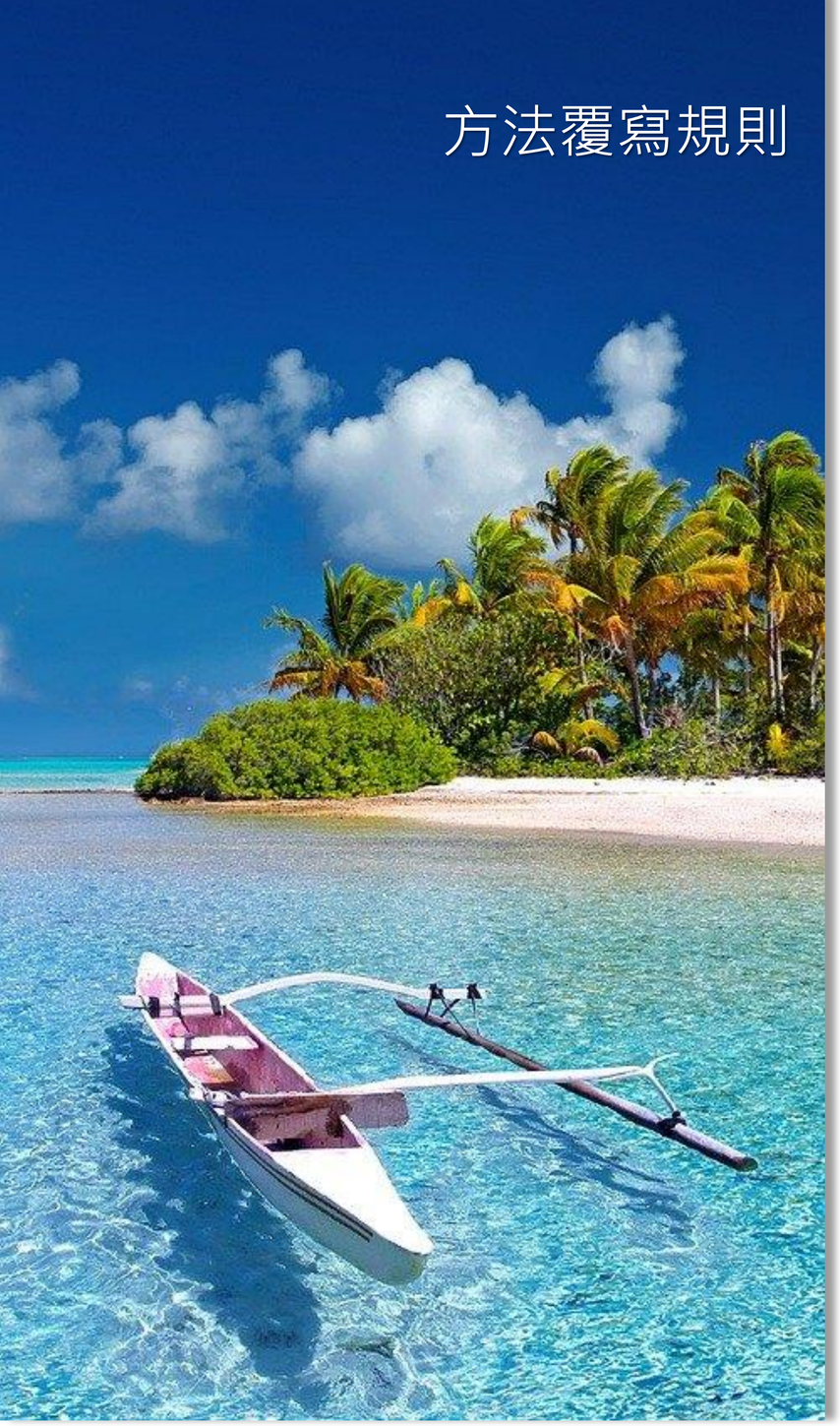
覆寫

- 所有覆寫(override)是發生在繼承關係中，子類別自行實作一個方法來取代父類別所提供的方法，程式在執行中會執行子類別的方法，而不執行父類別的方法，若欲執行父類別的方法，則可以在子類別覆寫的方法中加上 super.（點）父類別被覆寫的方法名稱
- 子類別繼承父類別的方法，若父類別方法內容不符合需求，可以宣告和父類別相同的名稱，相同的回傳值，相同的參數的方法來取代父類別繼承的方法
- 隱藏 - 如果父類別中有宣告static的類別方法，子類別的物件方法不能覆寫父類別的類別方法，但可以撰寫相同類別方法取代
- 子類別物件建立之後，若呼叫到此同名方法，會以子類別的方法優先執行

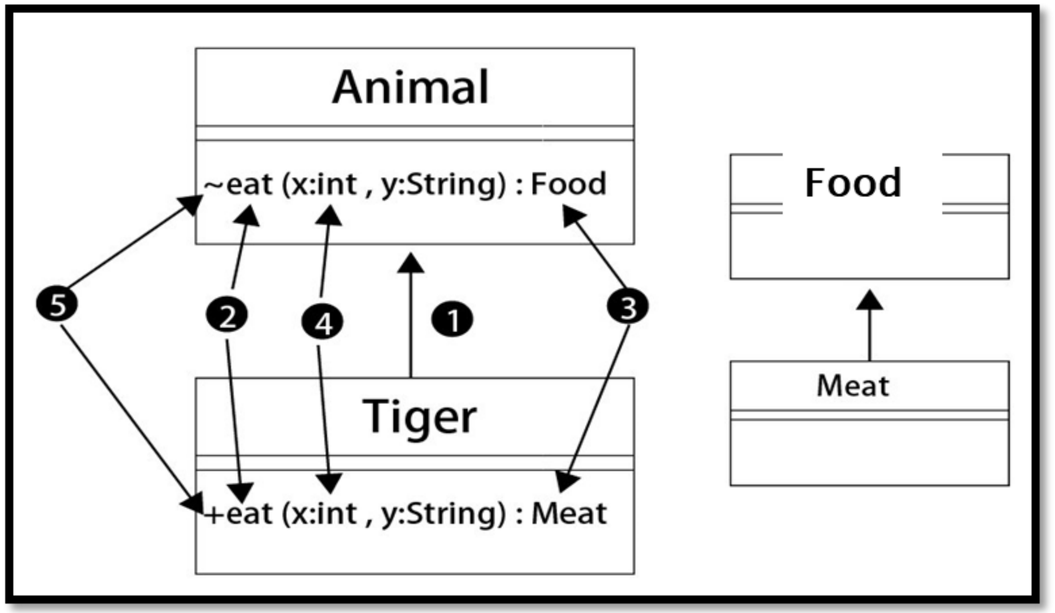


方法覆寫


Employee
-name : String
-salary : double
-birthDay : Date
+getDetails( )
Manager
-department : String
+getDetails( )
Overriding

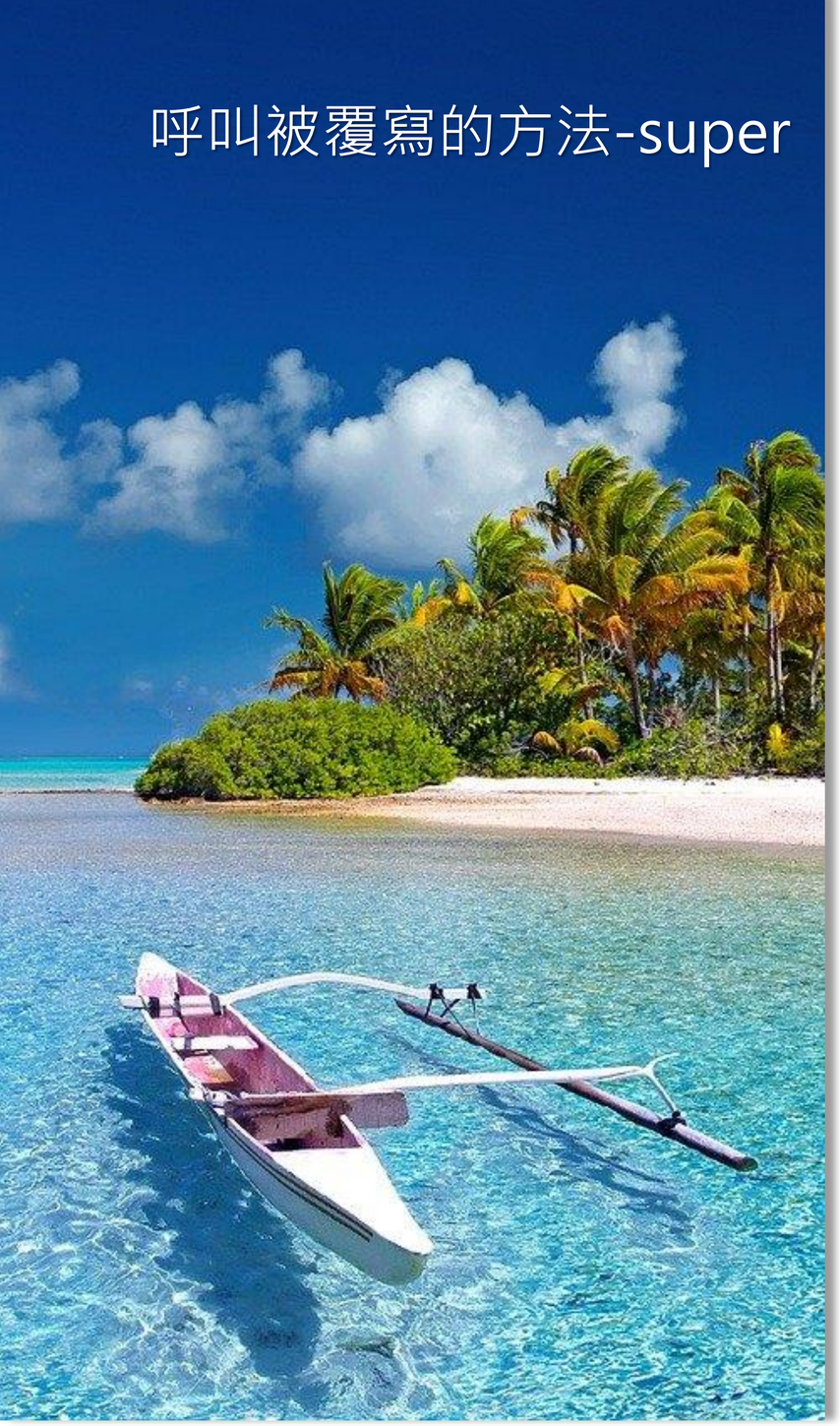


方法覆寫規則



- 有關覆寫的限製與注意事項以UML圖來說明
  1. 覆寫是發生在有繼承關的類別體系中
  2. 覆寫的方法名稱宣告必須相同
  3. 若方法有回傳值，其回傳型態基本資料型態必須相同若是物件則需相同或是子類別
  4. 方法中的參數列（不論數量，資料型別及擺放順序）都必須相同
  5. 存取權限不可小於原方法
  6. 當父類別的方法簽章有拋出例外事件,子類別覆寫方法的方法簽章可以
     - 選擇不拋出
     - 與父類別的例外相同或是子類別

呼叫被覆寫的方法-super

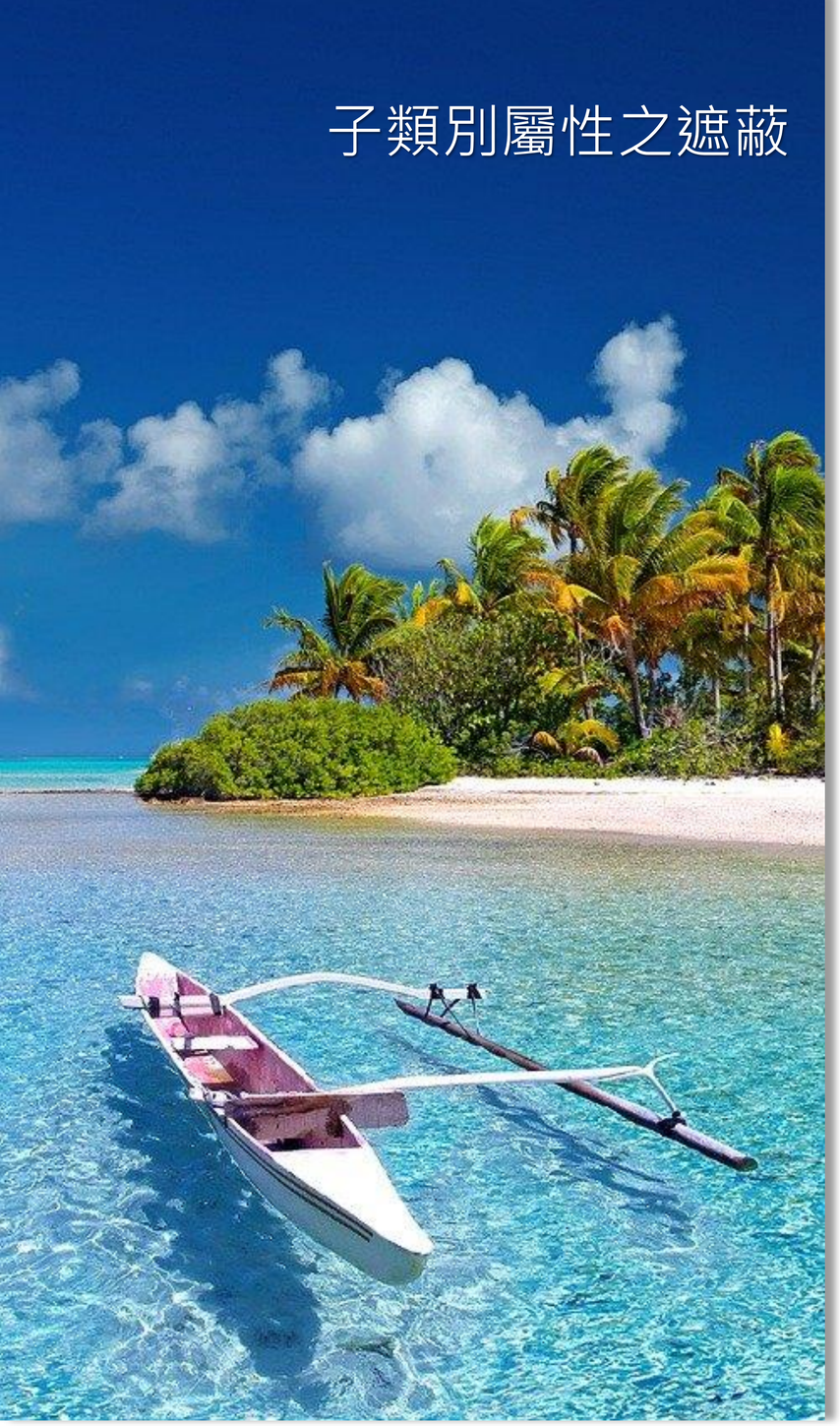


super 關鍵字

- 子類別物件中欲參考父類別物件的屬性、方法及建構子
  - super.屬性
  - super.方法(參數列)
  - Super 關鍵字必須在繼承關係的運作下才有意義

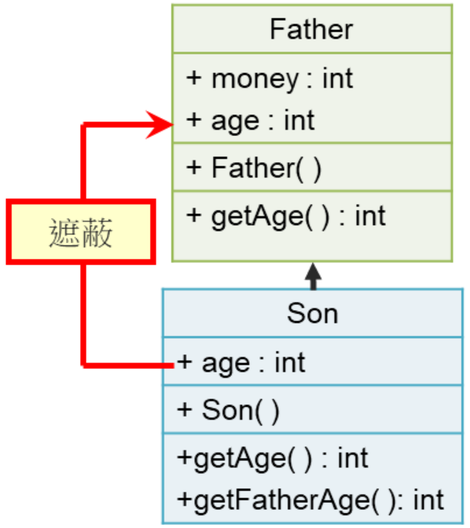
```
public class Employee {
    private String name = “Sean”;
    private double salary = 10000;
    public void getDetails( ) {
        System.out.println(“Name:” + name);
        System.out.println(“Salary:” + salary);
    }
}
```

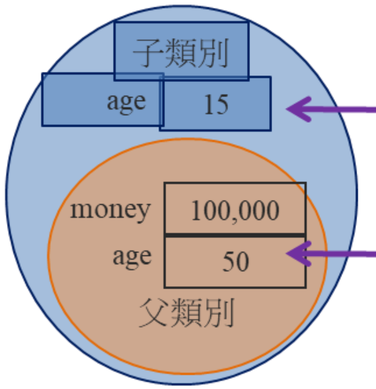
```
public class Manager extends Employee {
    private String dept = “EDU”;
    public void getDetails( ) {
        super.getDetails( );
        System.out.println(“Department:”+dept);
    }
}
```



子類別屬性之遮蔽

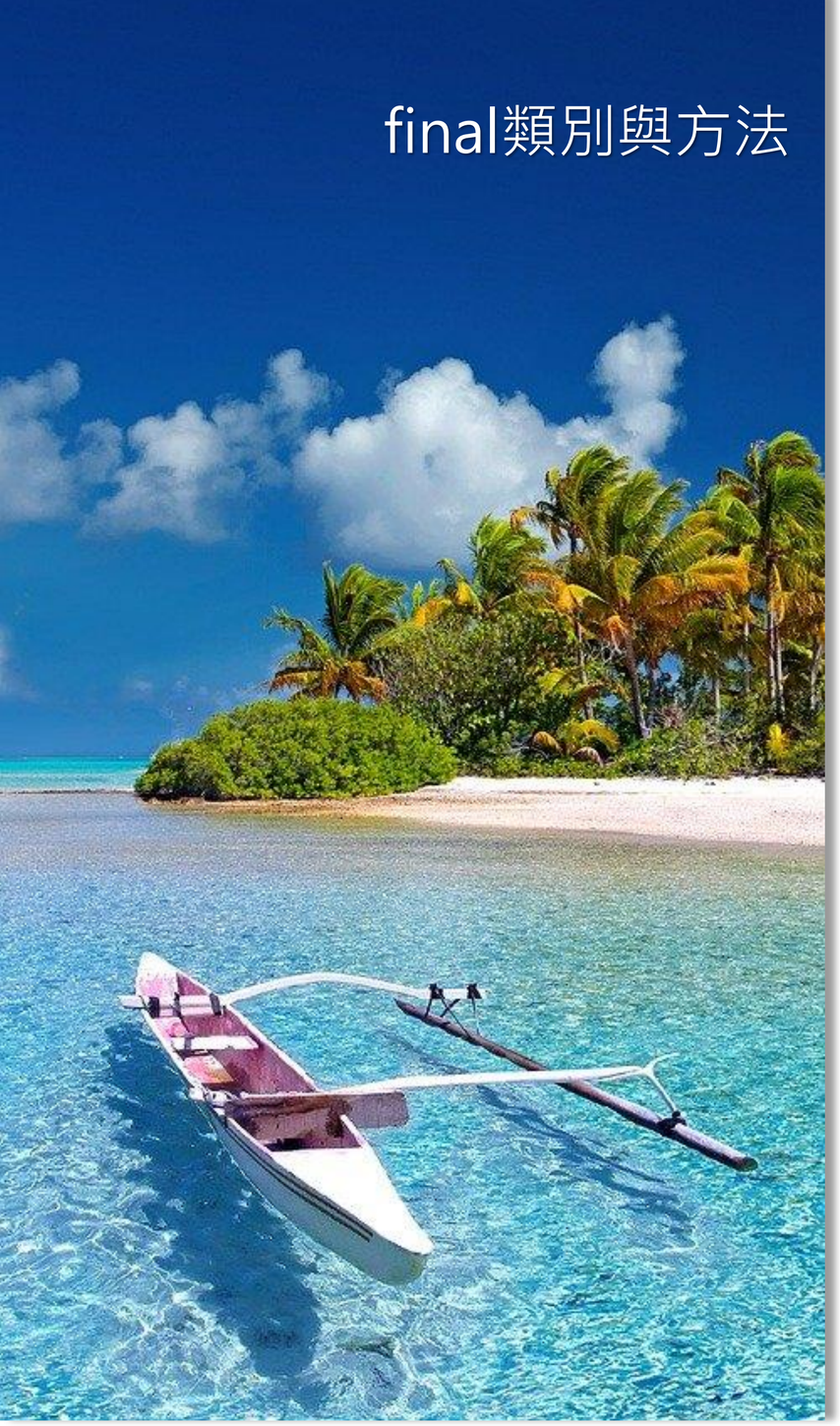
- 子類別屬性遮蔽
  - 子類別定義與父類別同名的屬性，產生遮蔽現象
  - 可利用 super 關鍵字來取用父類別的屬性






```
Son s = new Son( );
s.getAge( );
s.getFatherAge( );
public int getAge( ){
    return age;
}
public int getFatherAge( ){
    return super.age;
}
```

final類別與方法



- final 關鍵字
  - 表示被修飾元素是不可變更 (immutable) 的
  - 可用於修飾類別、變數及方法
- final classes
  - 類別不可被繼承

```
public final class FinalParentClass {
   ......
}
```

compile time error

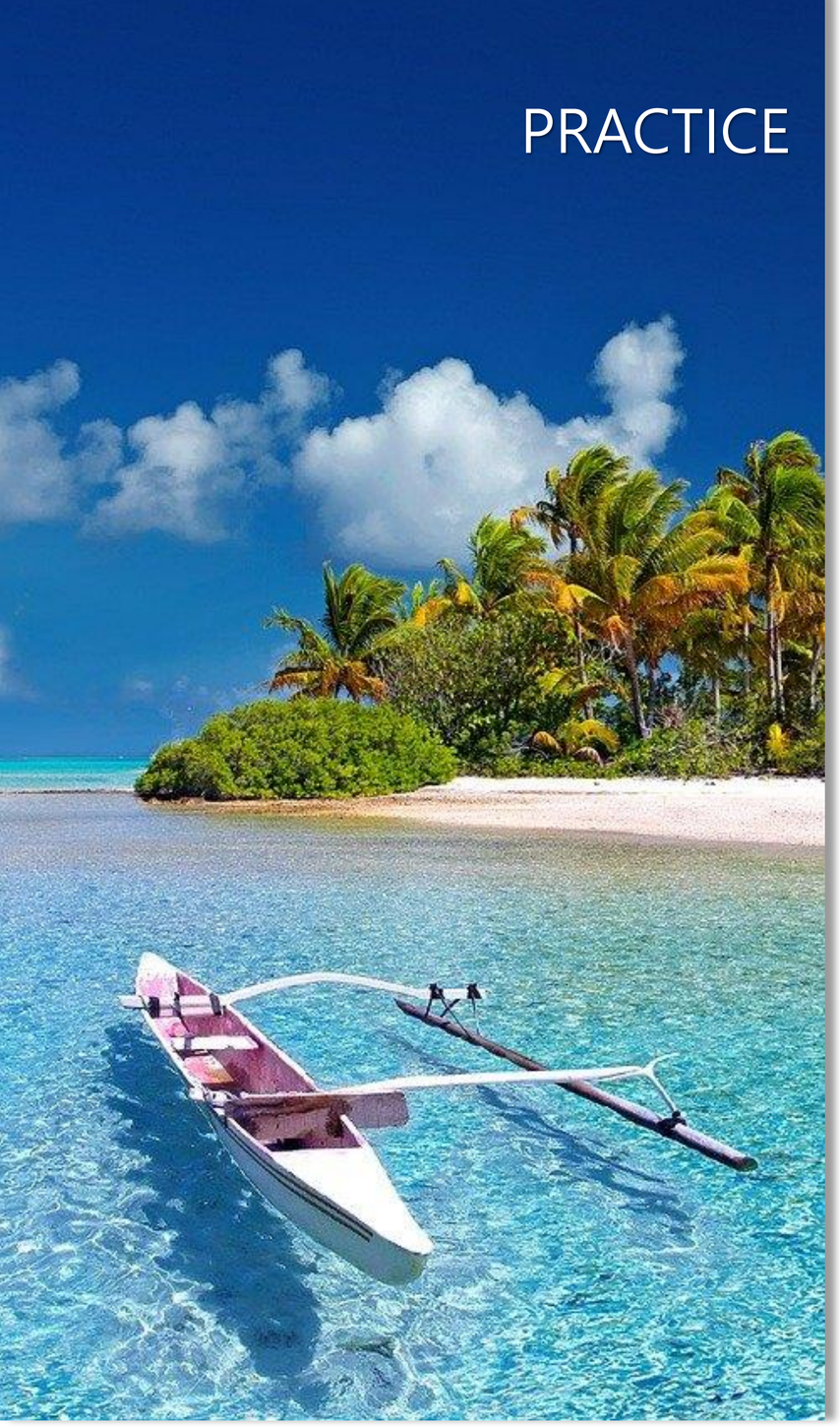
```
public class ChildClass extends FinalParentClass {
   ......
}
```

- final method
  - 方法不可被覆寫 (override)

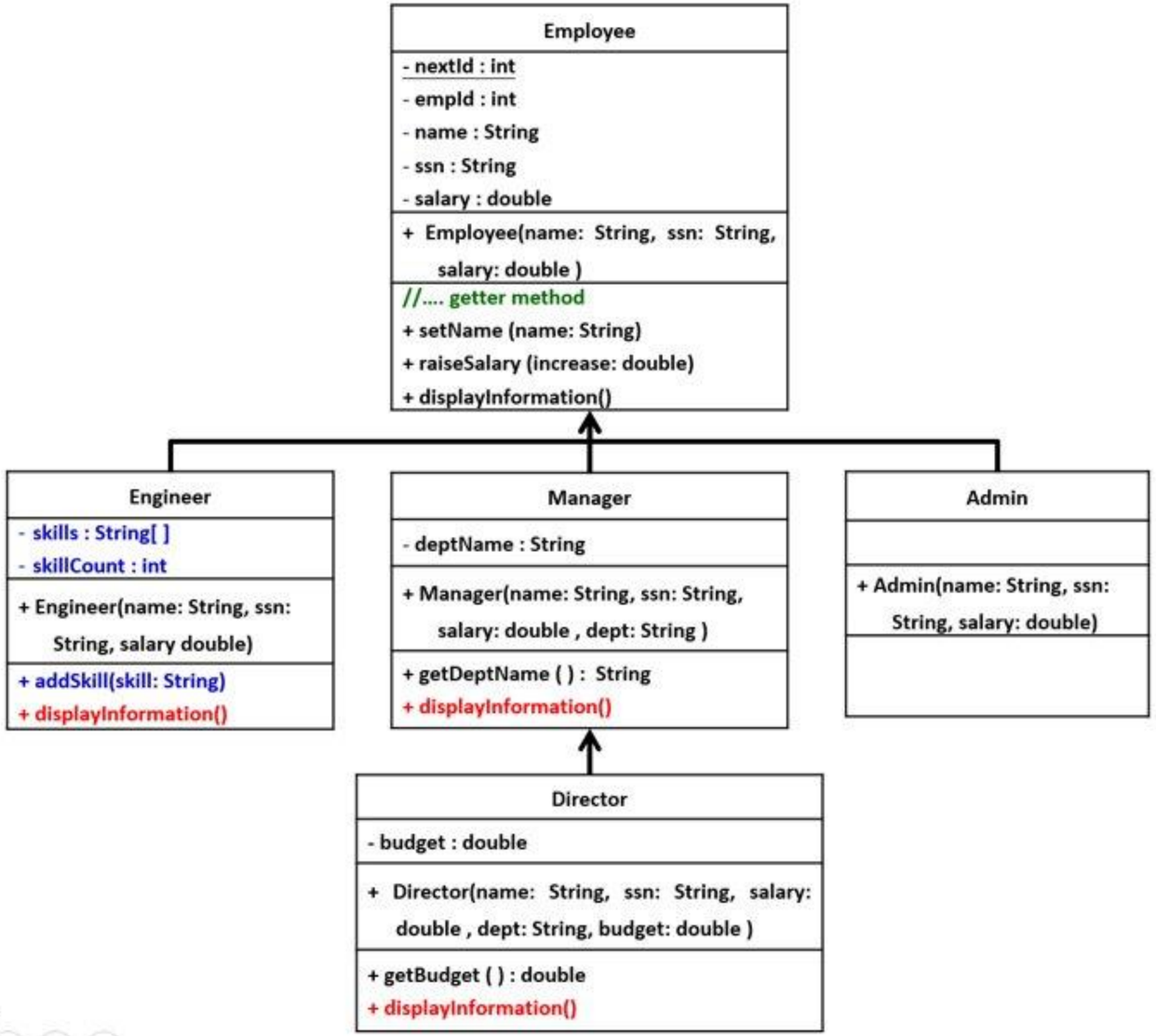
```
public class MethodParentClass {
   public final void printMessage() {
      System.out.println("This is a final method");
   }
}
```

compile time error

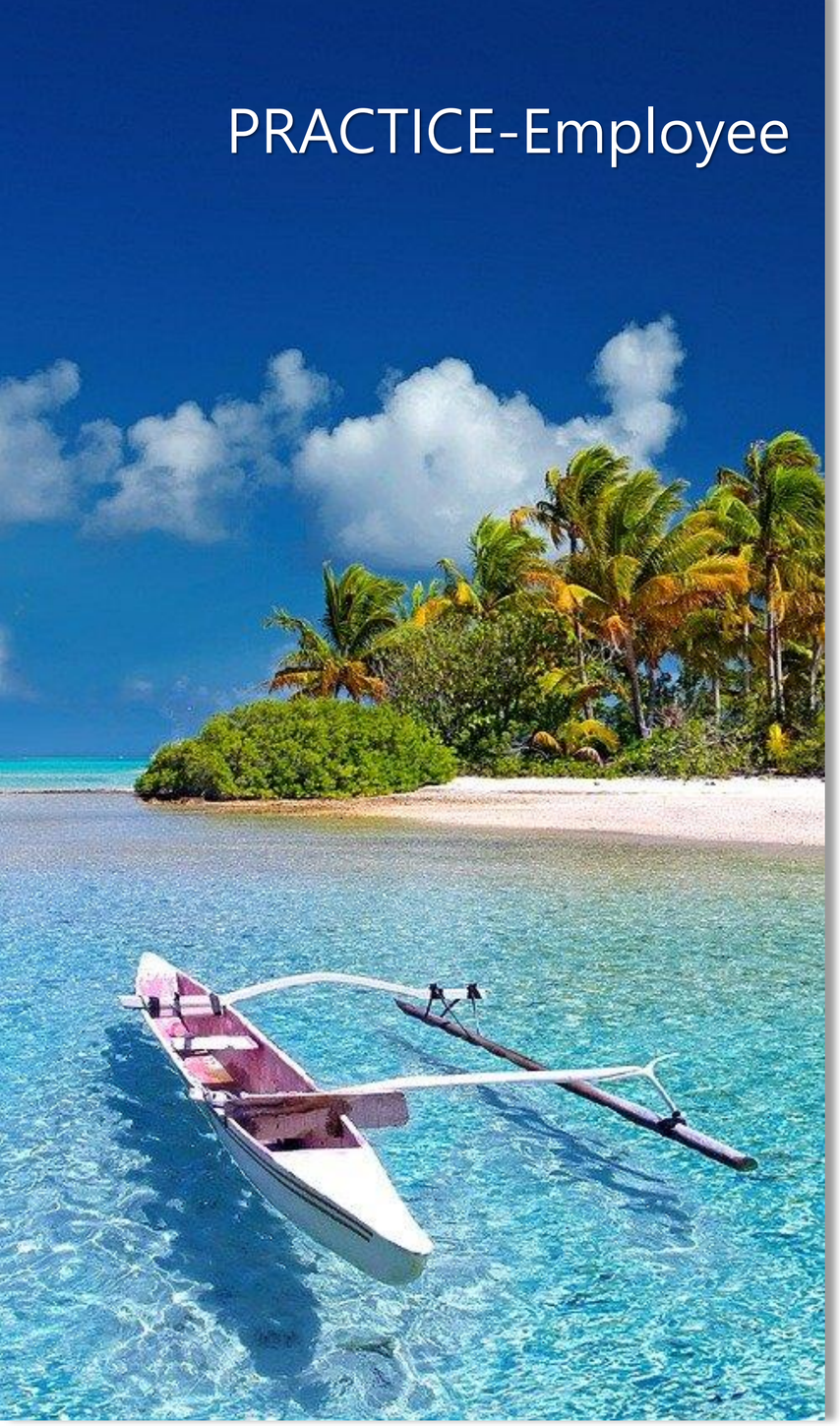
```
public class MethodChildClass extends MethodParentClass {
   public void printMessage() {
      System.out.println("Cannot override method");
   }
}
```



PRACTICE

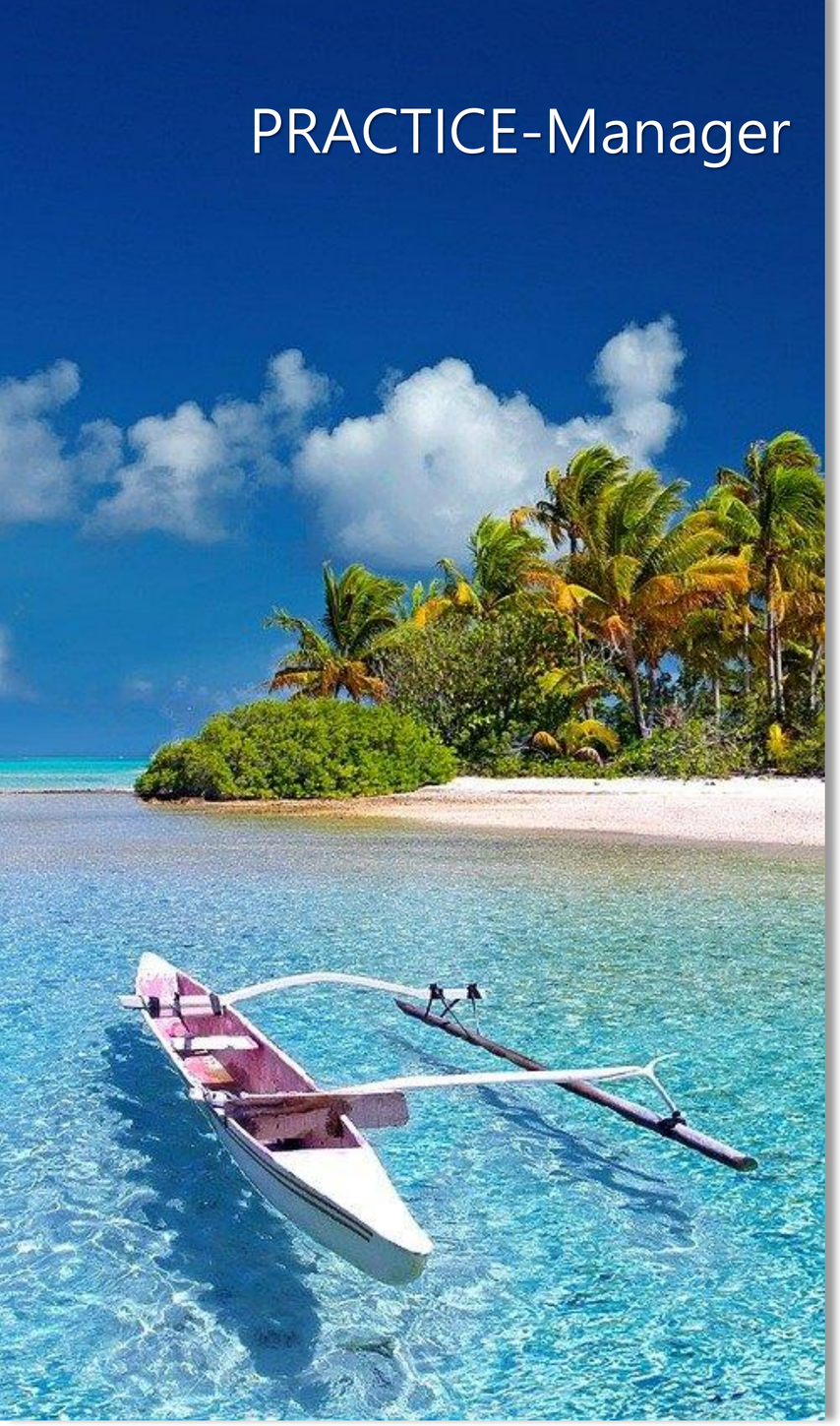

Employee
- nextId : int
- empId : int
- name : String
- ssn : String
- salary : double
+ Employee(name: String, ssn: String, salary: double)
//.... getter method
+ setName (name: String)
+ raiseSalary (increase: double)
+ displayInformation()
Engineer
- skills : String[ ]
- skillCount : int
+ Engineer(name: String, ssn: String, salary double)
+ addSkill(skill: String)
+ displayInformation()
Manager
- deptName : String
+ Manager(name: String, ssn: String, salary: double , dept: String)
+ getDeptName ( ) : String
+ displayInformation()
Admin
+ Admin(name: String, ssn: String, salary: double)
Director
- budget : double
+ Director(name: String, ssn: String, salary: double , dept: String, budget: double)
+ getBudget ( ) : double
+ displayInformation()

PRACTICE-Employee



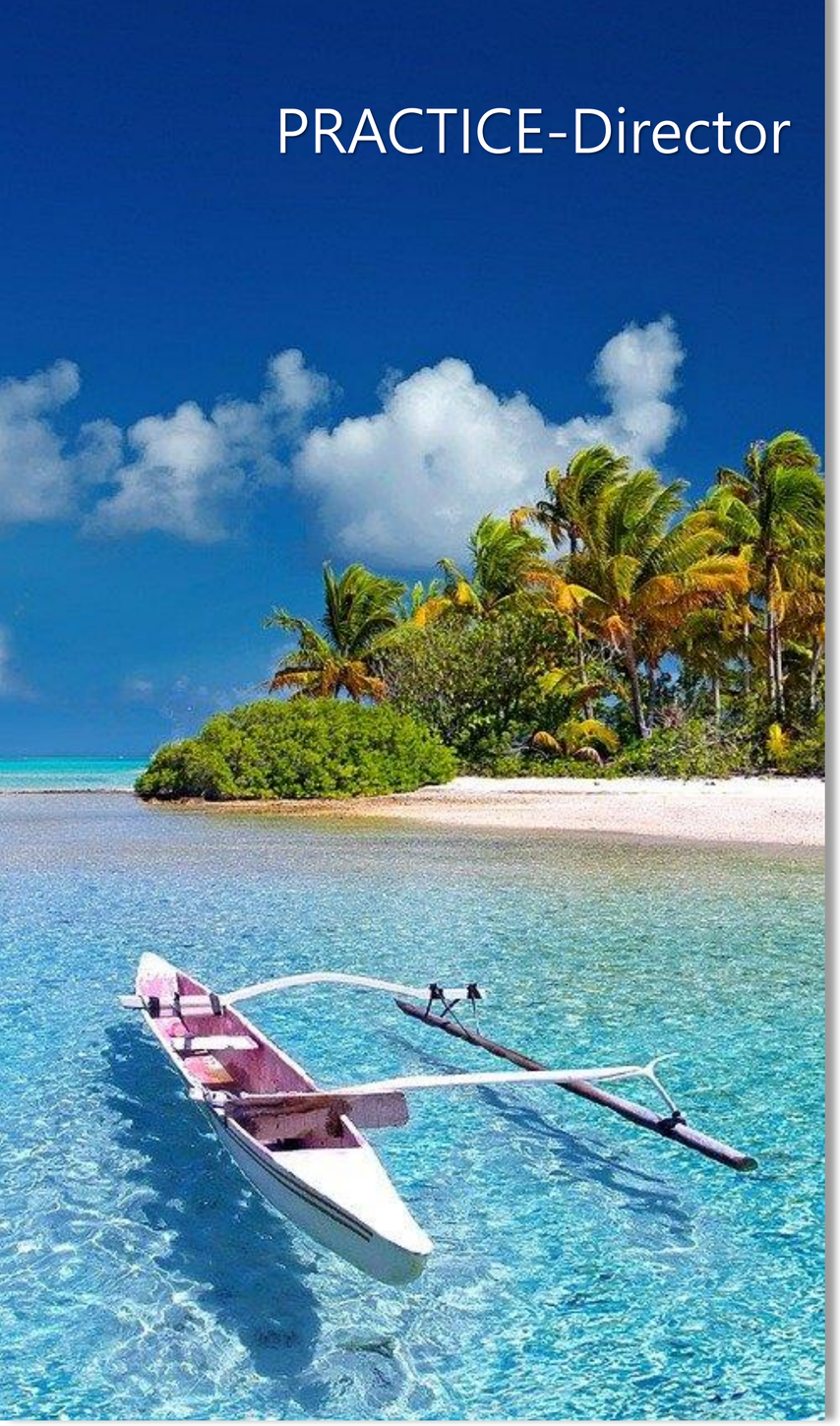
```
import java.text.NumberFormat;
import javax.swing.JOptionPane;


public class Employee {
private static int nextId=101;
private int empId;
private String name;
private String ssn;
private double salary;
NumberFormat formatter=NumberFormat.getCurrencyInstance();
public Employee(String name,String ssn,double salary)
{
this.empId=nextId++;
this.name=name;
this.ssn=ssn;
this.salary=salary;
}
public void setName(String name)
{
if(name==null || name.trim().length()>0)
this.name=name;
else
JOptionPane.showMessageDialog(null,"姓名輸入錯誤");
}
public void raiseSalary(double increase)
{
if(increase>0)
salary+=increase;
else
JOptionPane.showMessageDialog(null,"加薪必需大於0");
}
public void displayInformation(){
System.out.println("====員工資料====");
System.out.println("員工編號:"+empId);
System.out.println("員工姓名:"+name);
System.out.println("身份證號:"+ssn);
System.out.println("員工薪資:"+formatter.format(salary));


}
}
```

PRACTICE-Manager



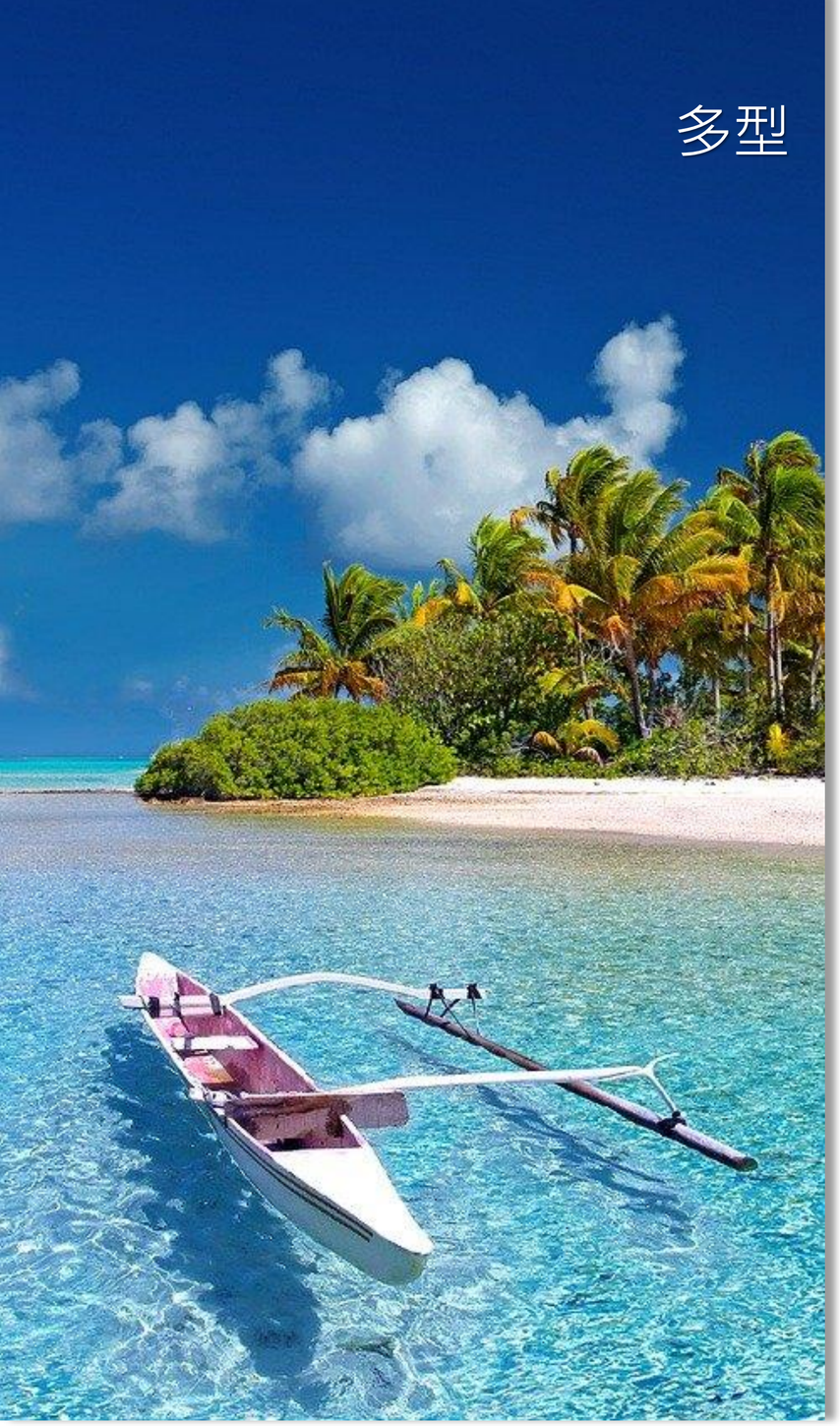
```
import javax.swing.JOptionPane;


public class Engineer extends Employee{
private String[] skills=new String[5];
private int skillCount=0;
public Engineer(String name,String ssn,double salary){
super(name,ssn,salary);
}
public void addSkill(String skill)
{
if(skillCount<=4)
skills[skillCount++]=skill;
else
JOptionPane.showMessageDialog(null,"技能最多只能有五個","錯誤
",JOptionPane.ERROR_MESSAGE);
}
@Override
public void displayInformation(){
super.displayInformation();
System.out.println("擁有的特殊技能:");
for(String s:skills)
{
if(s !=null)
System.out.print(s+"\t");
}
System.out.println();
}
}
```

PRACTICE-Director



```
public class Director extends Manager {
private double budget;
public Director(String name,String ssn,double salary,String dept,double budget)
{
super(name,ssn,salary,dept);
this.budget=budget;
}
public double getBudge(){
return this.budget;
}
@Override
public void displayInformation(){
super.displayInformation();
System.out.println("管理預算:"+formatter.format(budget));
}
}
```

多型 POLYMORPHISM

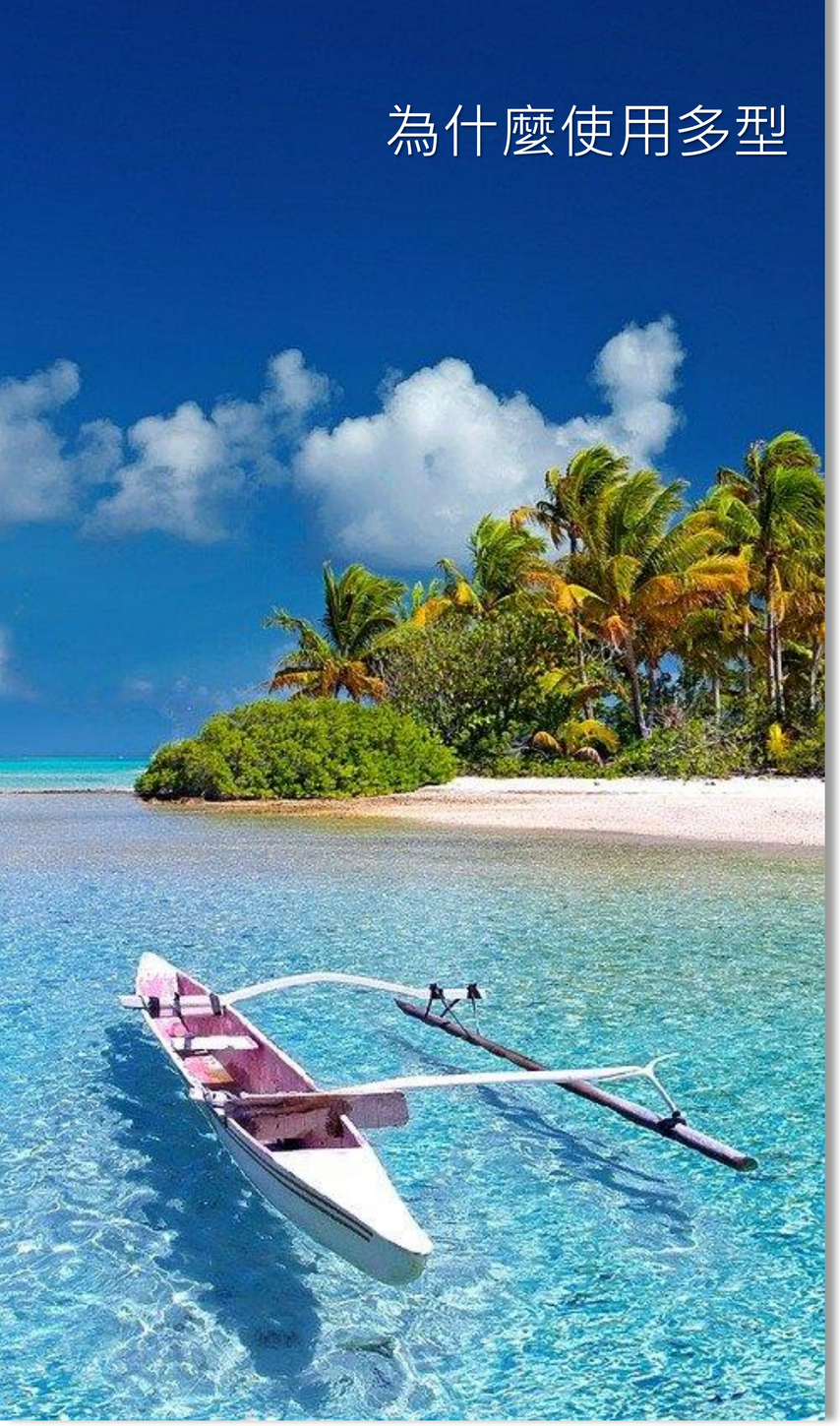


多型

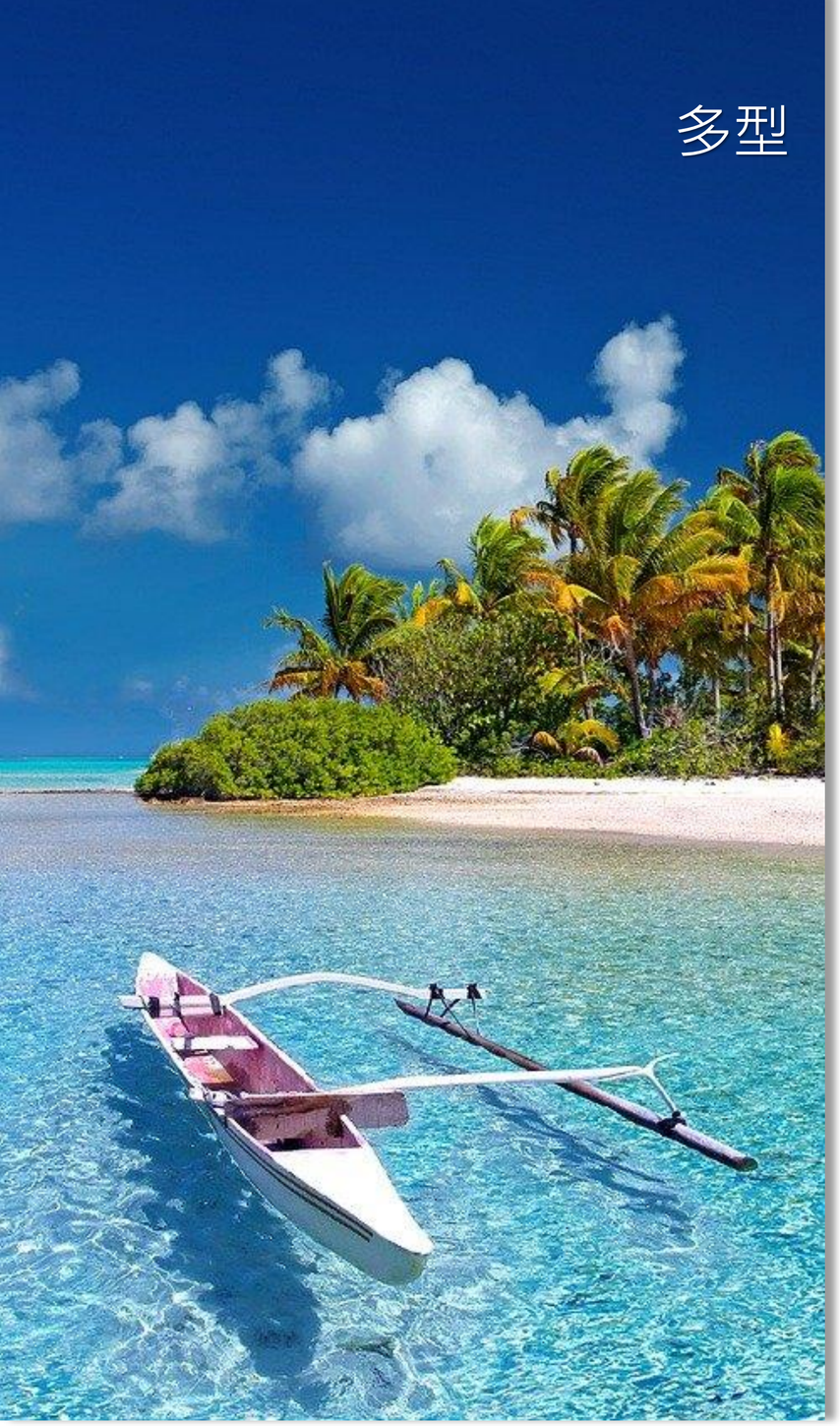
- 多型的意義
  - 一個物件可以用多種形態來看待。
  - 型態間需有繼承關係：子類別可以被看待為父類別
- Java 技術實作多型
  - 具有繼承關係的架構下，物件實體可以被視為多種型別。
  - 將子類別物件參考指定給父類別變數。

父類別　變數名稱 = new 子類別建構子( );

為什麼使用多型

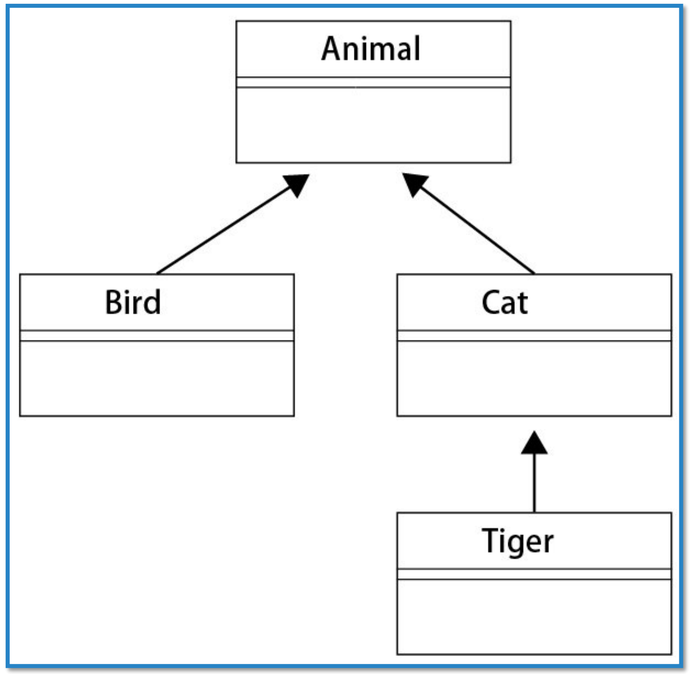


```
public class PersonTest {
    public static void main(String[] args) {
        new PersonTest().speakLanguage(new Person());
        new PersonTest().speakLanguage(new Taiwan());
        new PersonTest().speakLanguage(new
American());
        new PersonTest().speakLanguage(new Japan());
    }
    void speakLanguage(Person person){
        person.speak();
    }
//後續下方皆可省略
    void speakLanguage(Taiwan taiwan){
        taiwan.speak();
    }
    void speakLanguage(American american){
        american.speak();
    }
    void speakLanguage(Japan japan){
        japan.speak();
    }
//省略到此
}
class Person{
    void speak(){
        System.out.println("我是人類,我會講話");
    }
}
class Taiwan extends Person{
    @Override
    void speak(){
        System.out.println("我是台灣人,我會講台語");
    }
}
class American extends Person{
    @Override
    void speak(){
        System.out.println("我是美國人,我會講美語");
    }
}
class Japan extends Person{
    @Override
    void speak(){
        System.out.println("我是日本人,我會講日語");
    }
}
```

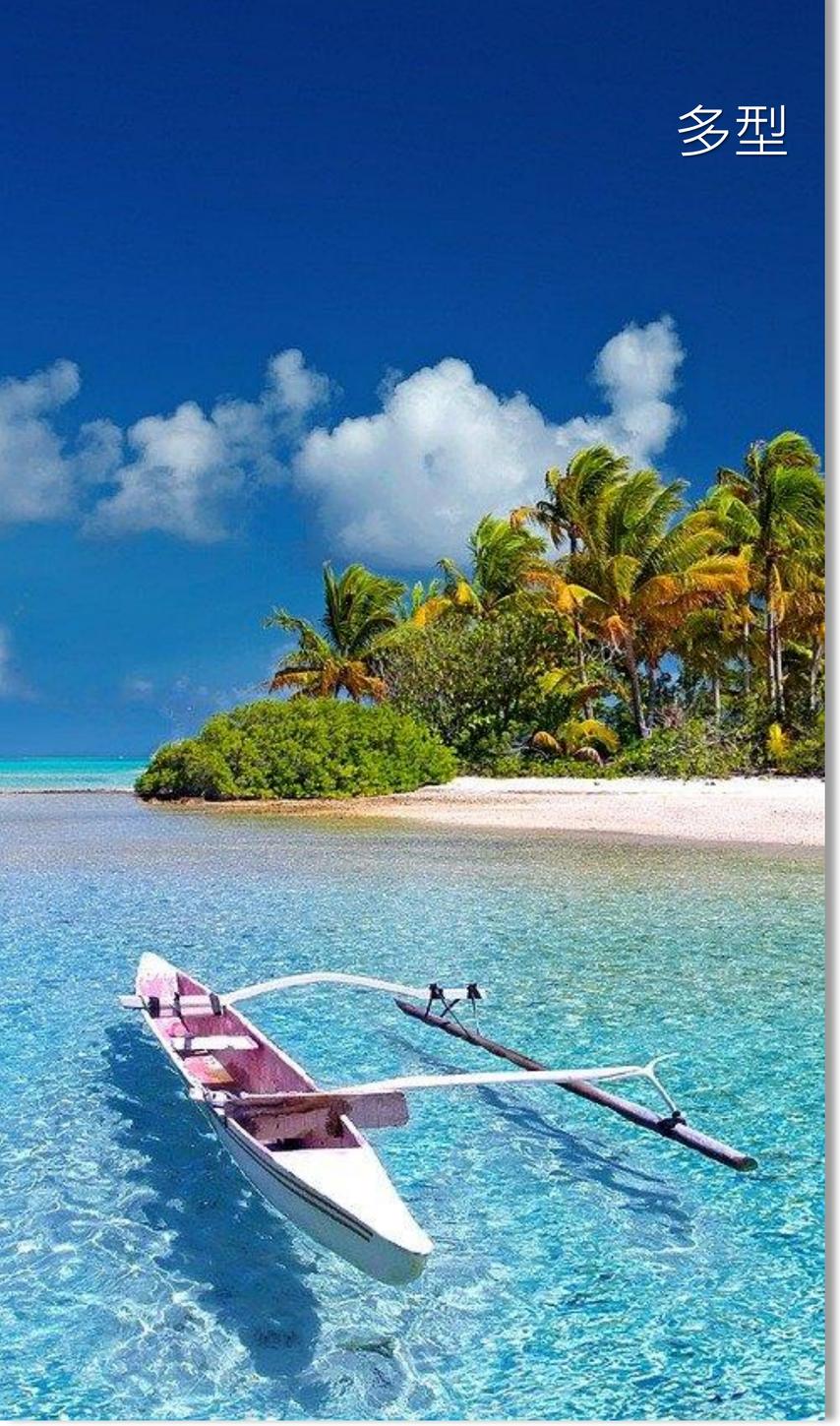


多型

```
class Animal{}
class Cat extends Animal{}
class Bird extends Animal{}
class Tiger extends Cat{}
```

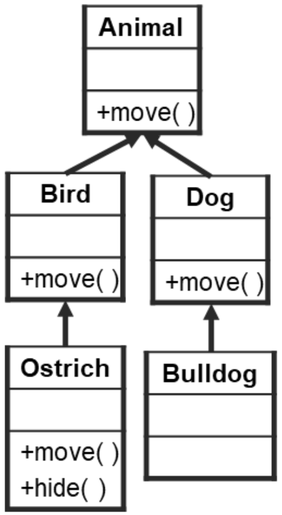


- 在new Tiger()時，資料型別可以有下列三種
- 第一種
  - Tiger t=new Tiger();//用Tiger的眼光來看Tiger
- 第二種
  - Cat c=new Tiger();//用Cat貓科動物的眼光來看Tiger
  - 利用Cat型別的觀點來建立Tiger的實體--new Tiger()，因此在實作上c是一個具有Cat資料型別的物件變數，可以利用它來合法存取new Tiger()實體中的資源
- 第三種
  - Animal a=new Tiger();//用Animal動物的眼光來看Tiger
  - 同樣地，利用Animal的觀點來建立Tiger的實體，實作上a就是一個具有Animal資料型別的物件變數，利用它來合法存取new Tiger()實體中的資源
- 由上可知，單一實體new Tiger()可以被宣告成多種型別如 Tiger，Cat與Animal此即為多型，單一實體Tiger可以被宣告成多種型別 Animal，Cat，Tiger此即為多型，不過這些都要在繼承關係下才能成立



多型

- 鬥牛犬是一種狗，所有的狗皆是一種動物。
- 鴕鳥是一種鳥類，所有鳥類皆是一種動物。




```
class Animal {
    void move( ){…}
}
class Bird extends Animal {
    void move( ){…}
}
class Dog extends Animal {
    void move( ){…}
}
class Ostrich extends Bird {
    void move( ){…}
    void hide( ){…}
}
class Bulldog extends Dog {
}
```

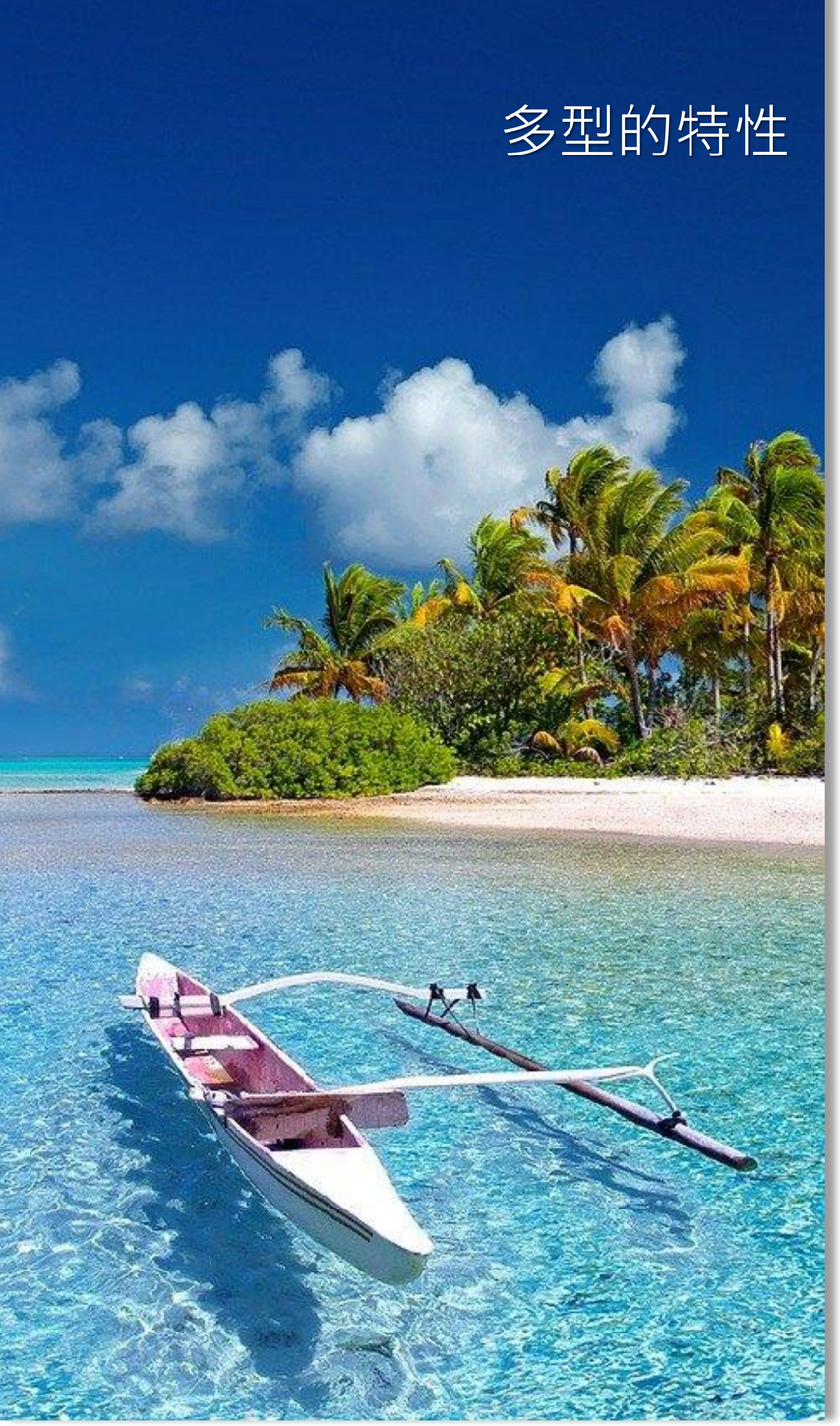
Bulldog b = new Bulldog( );
用 **Bulldog** 鬥牛犬的眼光來看 **Bulldog**

Dog d = new Bulldog( );
用 **Dog** 狗的眼光來看 **Bulldog**

Animal a = new Bulldog ( );
用 **Animal** 動物的眼光來看 **Bulldog**

Bulldog b1 = new Dog( );
用 **Bulldog** 鬥牛犬的眼光來看所有的狗

編譯錯誤

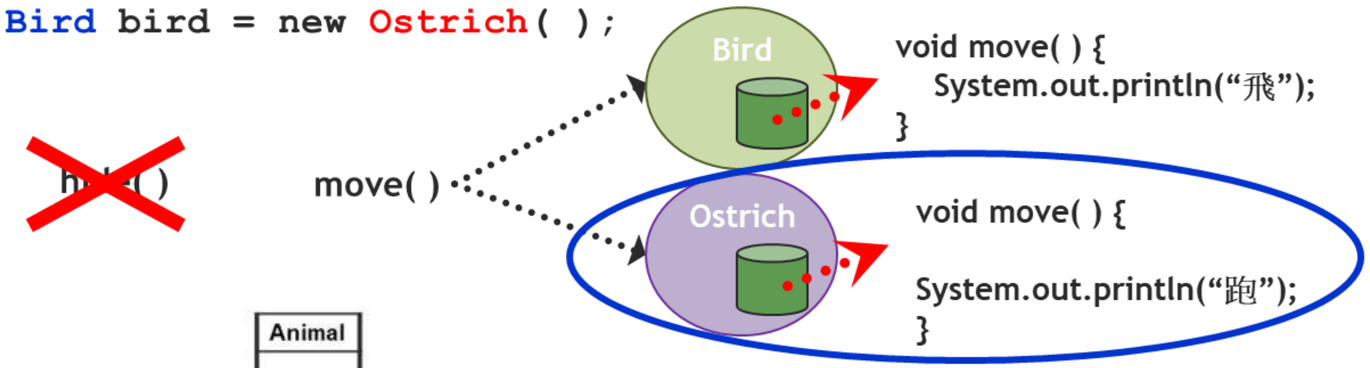


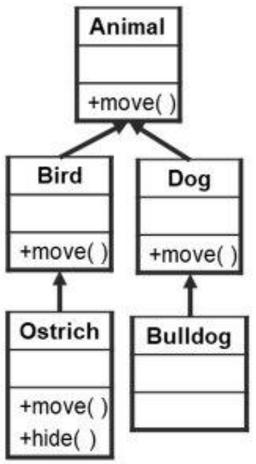
多型的特性

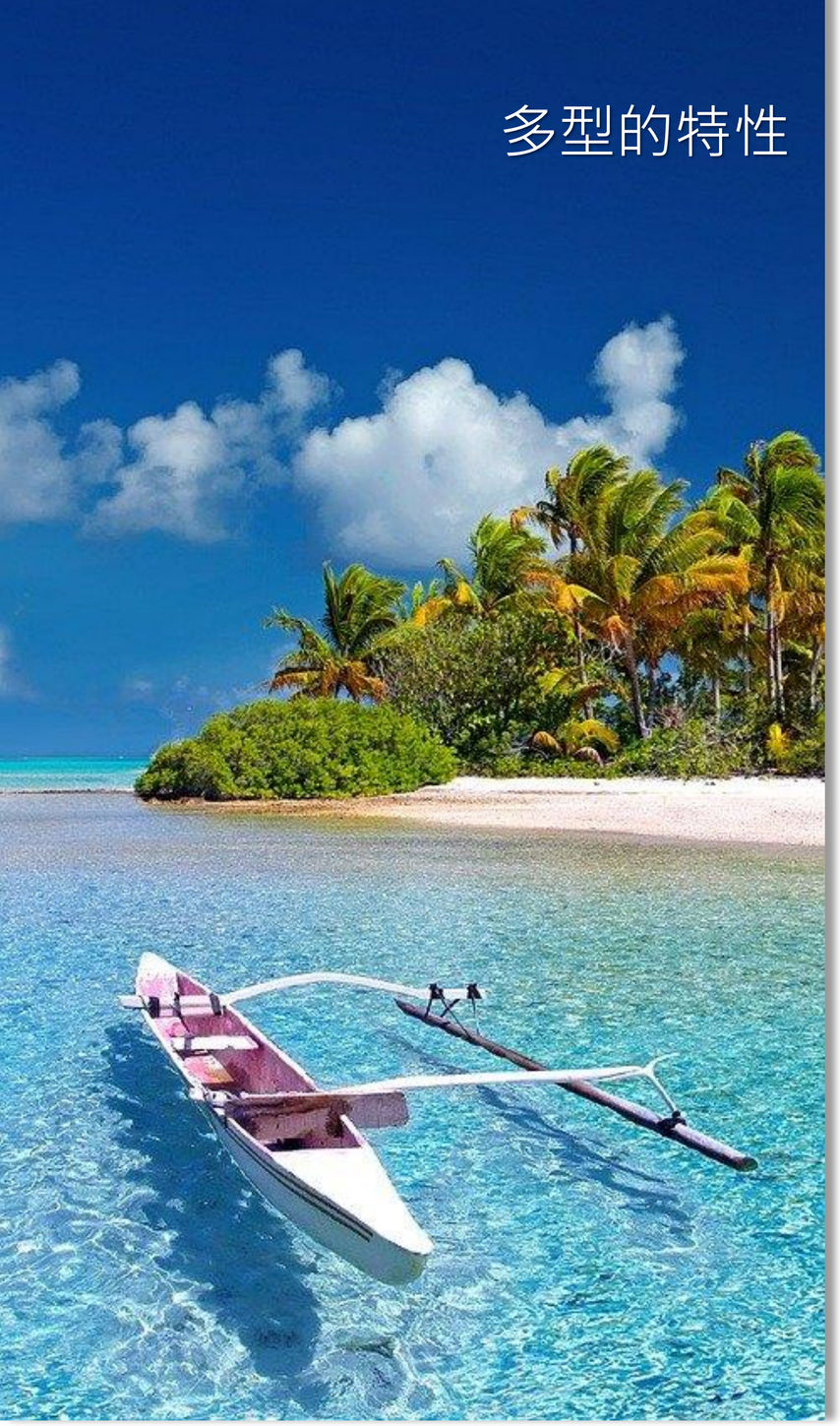
多型的特性

- 不同型態表示並不會改變原來的實體。
- 將物件視為父類別,只能用父類別有定義之屬性及方法。
- 父類別方法被子類別覆寫，多型用父類別的觀點呼叫，仍會執行子類別的方法。

```
Bird bird = new Ostrich( );
```







多型的特性

```
class Animal{
  void move( ) {
    System.out.println("動");
  }
}
```

```
class Bird extends Animal{
  void move( ) {
    System.out.println("飛");
  }
}
```

```
class Ostrich extends Bird {
  void move( ) {
    System.out.println("跑");
  }
  void hide( ) {
    System.out.println("頭埋在土裡");
  }
}
```

```
Bird bird = new Bird( );
bird.move( );
```

⇨ 飛

```
Ostrich ostrich = new Ostrich( );
ostrich.move( );
ostrich.hide( );
```

⇨

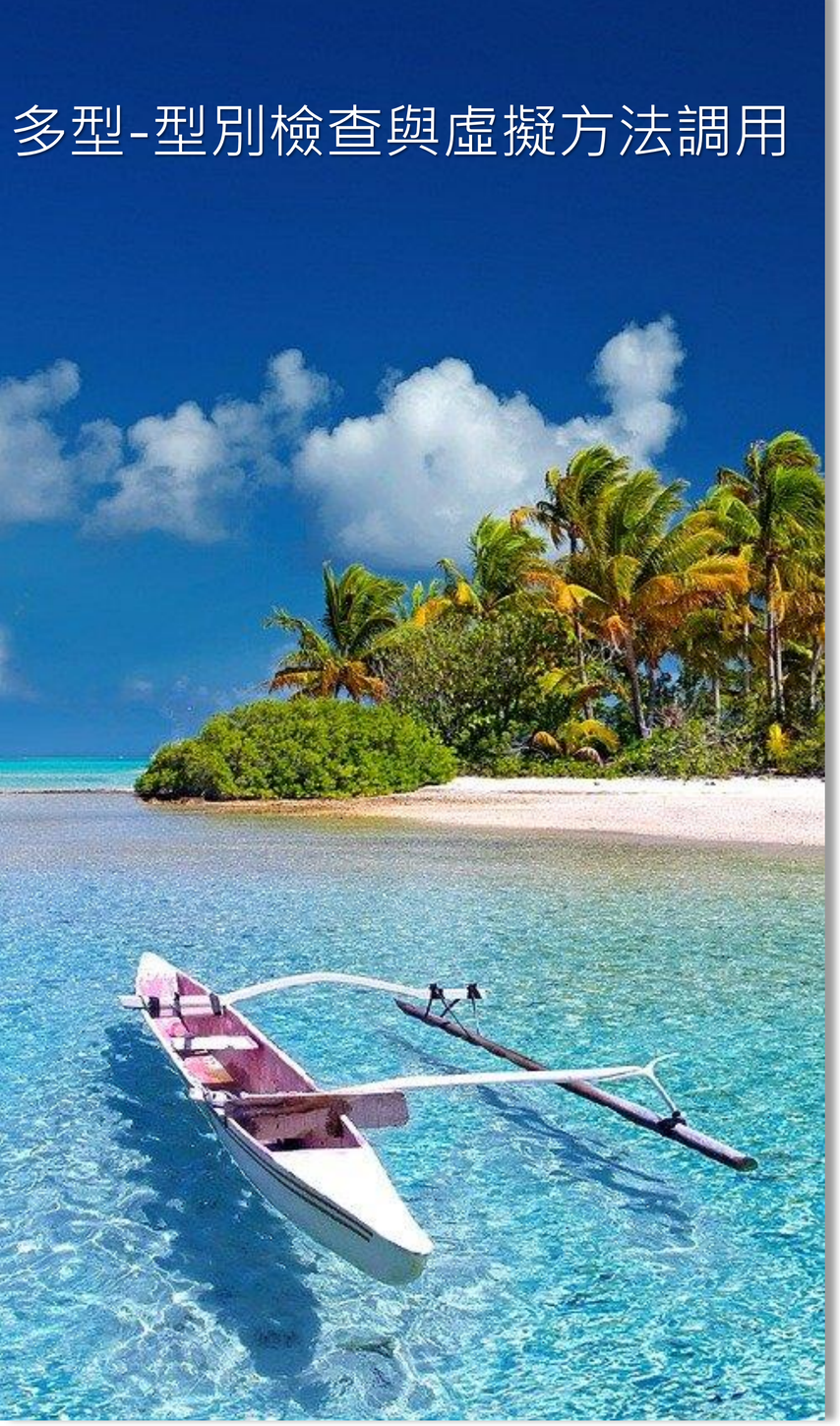
跑
頭埋在土裡

```
Bird bird2 = new Ostrich( );
bird2.hide( );

bird2.move( );
```

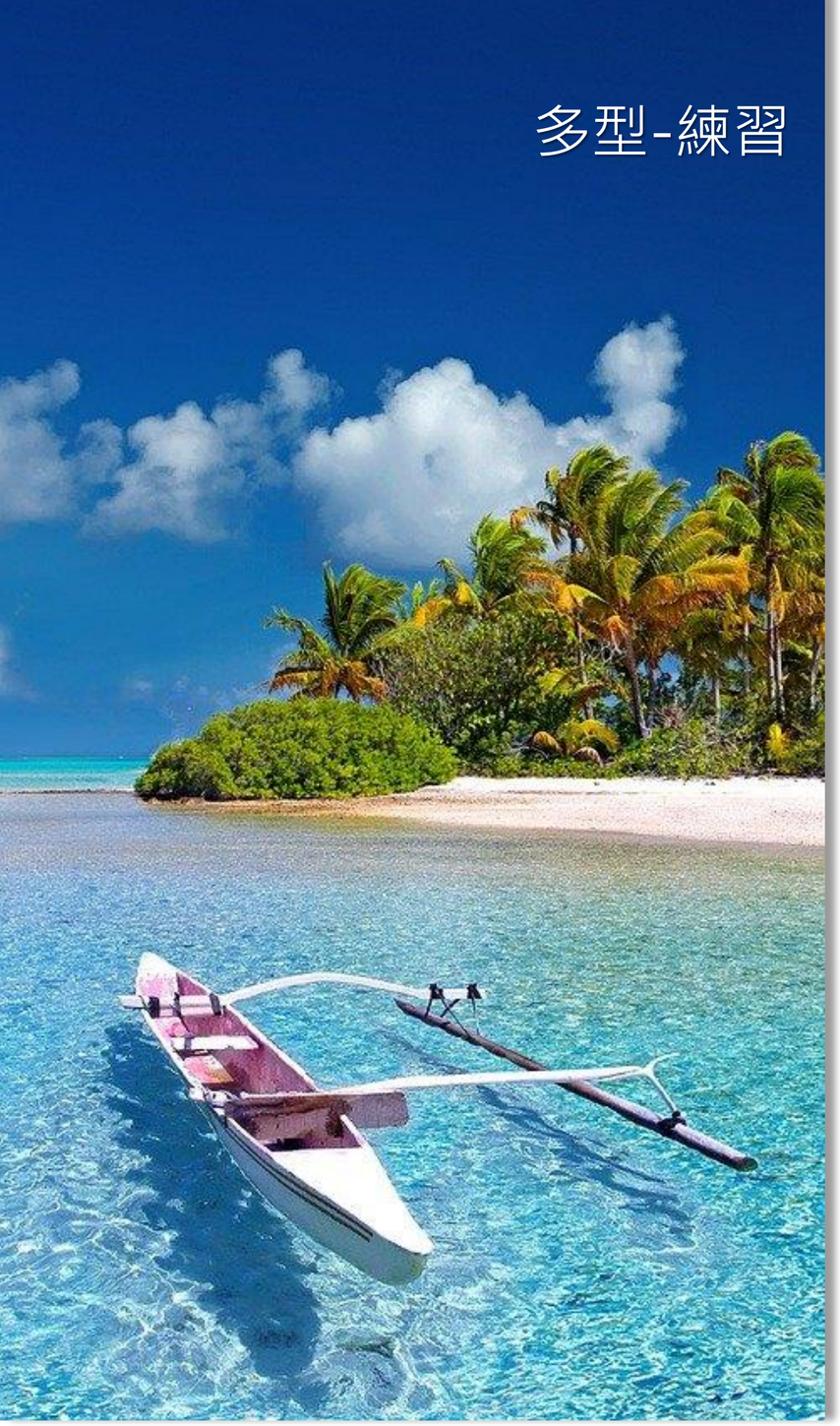
⇨ 跑

在 **Bird** 型別中並不知道有 **hide( )** 方法

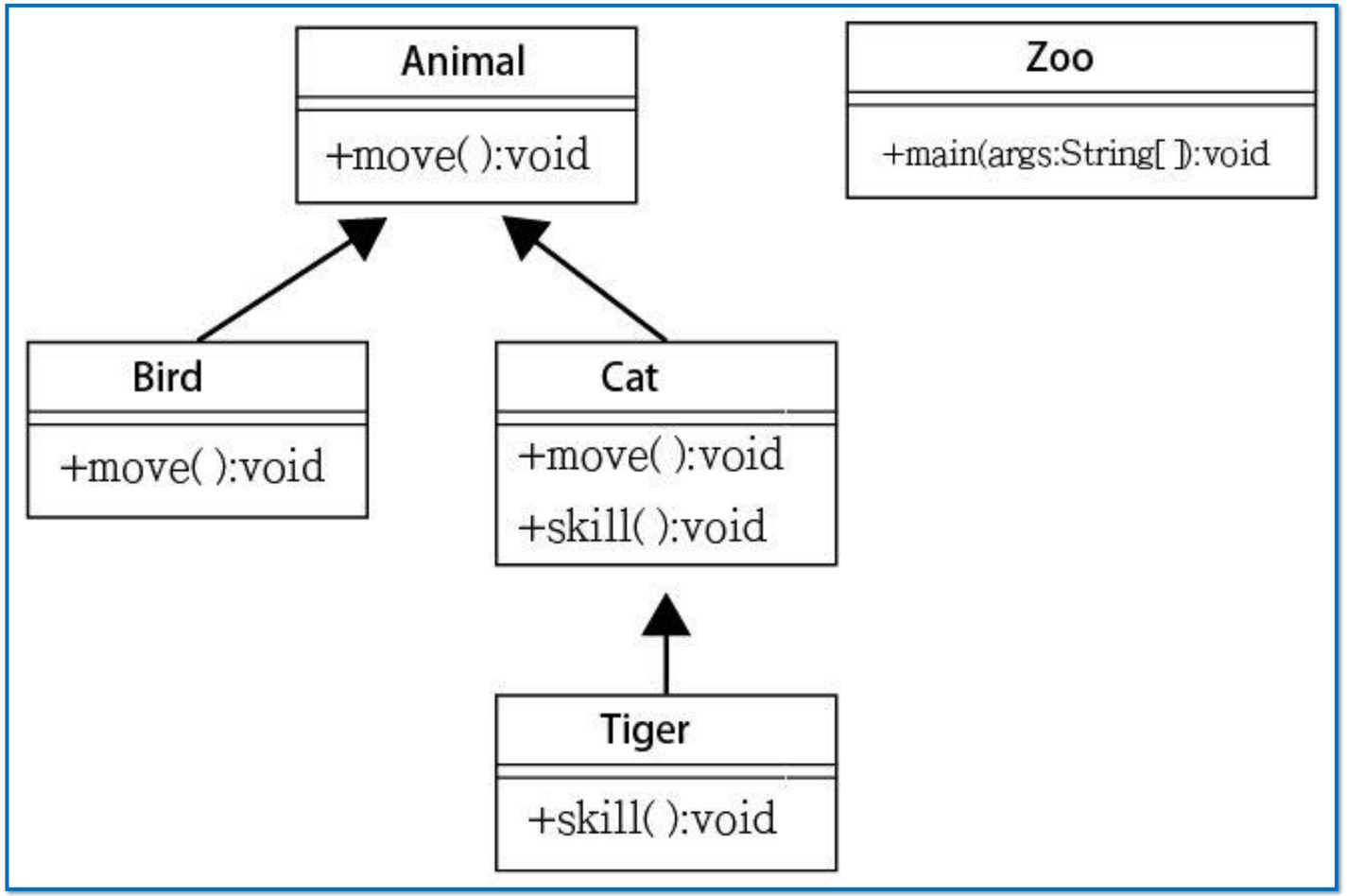
多型-型別檢查與虛擬方法調用



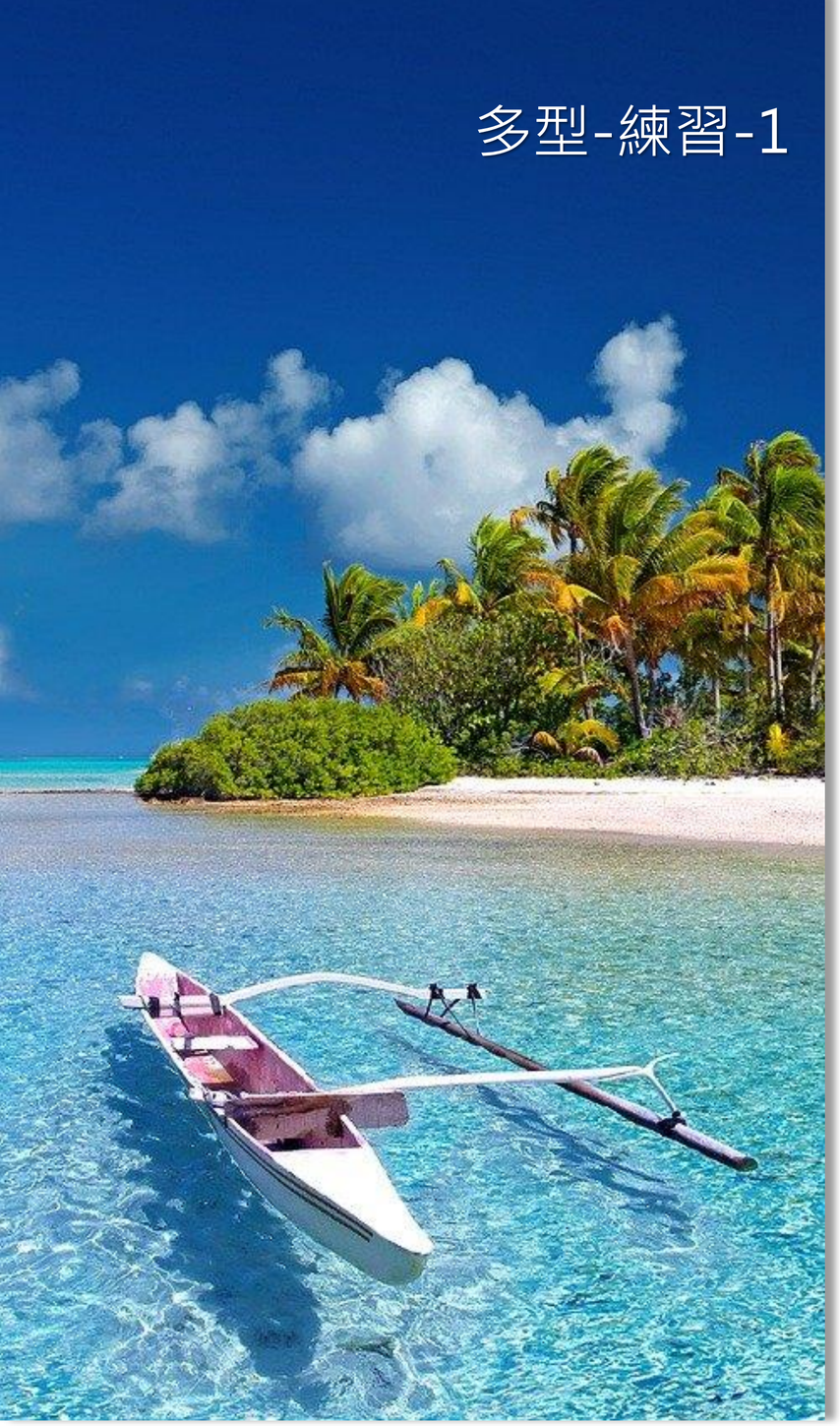
- Java 的型別檢查
  - 編譯時期，compiler 會以宣告的型別作型別檢查
    - 確保物件被視為父類別，只能用父類別定義之屬性及方法
  - 執行時期，JVM 會以實際的型別作型別檢查
    - 確保多型時,用父類別的觀點呼叫，仍會執行子類別的方法
- 虛擬方法調用 (呼叫) Virtual Method Invocation
  - Java 程式會調用 (呼叫) 變數在執行時期所參考之物件的行為，而不是在編譯時期宣告類別的行為



多型-練習

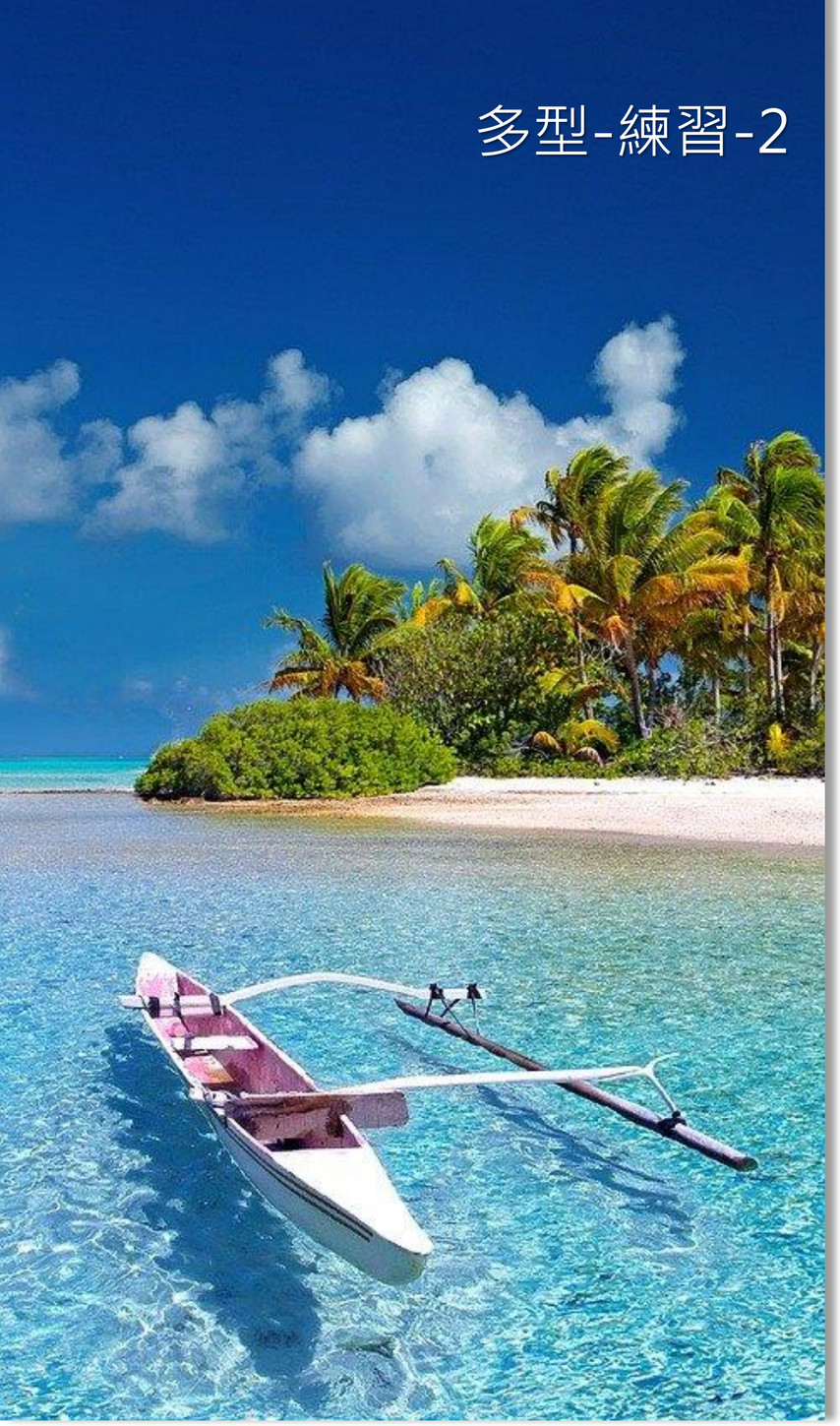

Animal
+move( ):void
Zoo
+main(args:String[ ]):void
Bird
+move( ):void
Cat
+move( ):void
+skill( ):void
Tiger
+skill( ):void

多型-練習-1



```
class Animal{
    public void move(){
        System.out.println("我是動物,我會移動");
    }
}
class Birds extends Animal{
    @Override
    public void move(){
        System.out.println("我是鳥類,我會飛來飛去");
    }
}

class Cat extends Animal{
    @Override
    public void move(){
        System.out.println("我是貓科,我會跑跑跳跳");
    }
    public void skill(){
        System.out.println("我是貓科,我的技能是抓魚");
    }
}
class Tiger extends Cat{
    @Override
    public void skill(){
        System.out.println("我是老虎,我的技能是狩獵");
    }
}
```

多型-練習-2

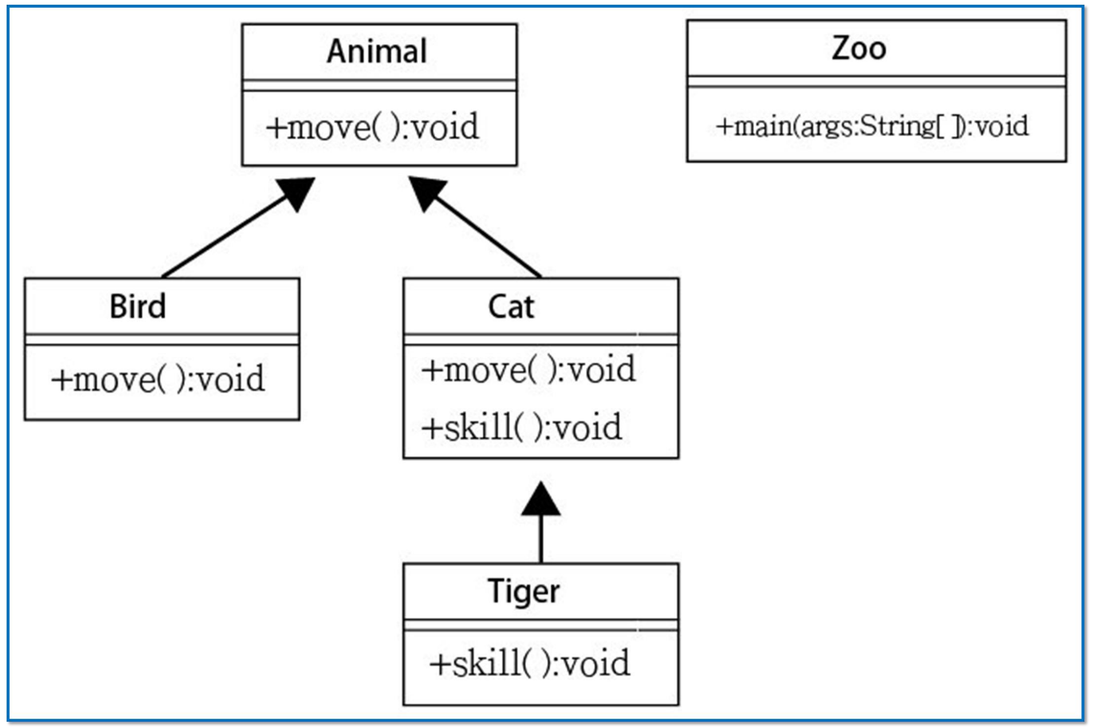


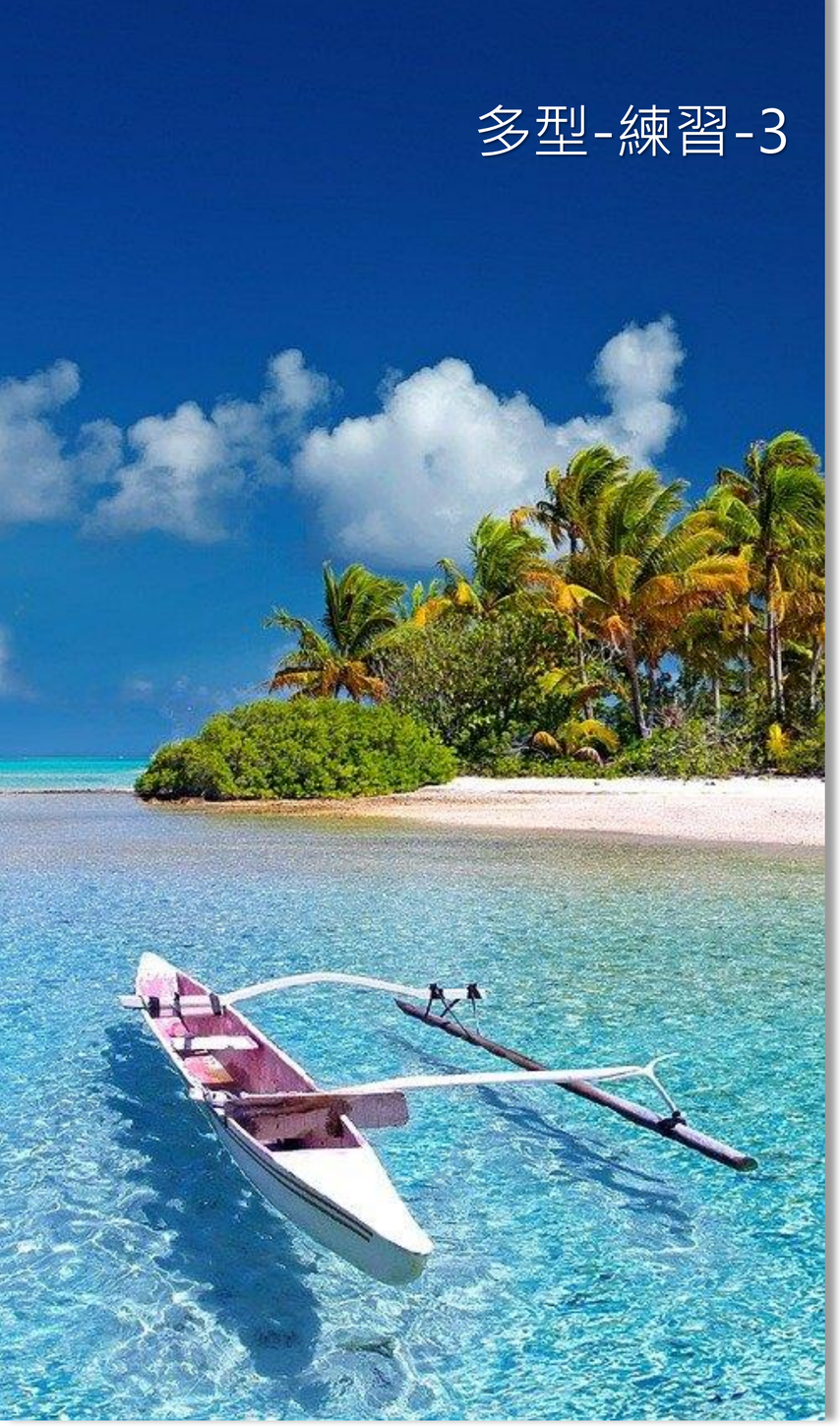
在Zoo類別主方法中new Tiger物件

Tiger t=new Tiger();

t.skill();//執行自己的skill方法

t.move();//自己沒有此方法,但因繼承關係,呼叫父層的move方法





多型-練習-3

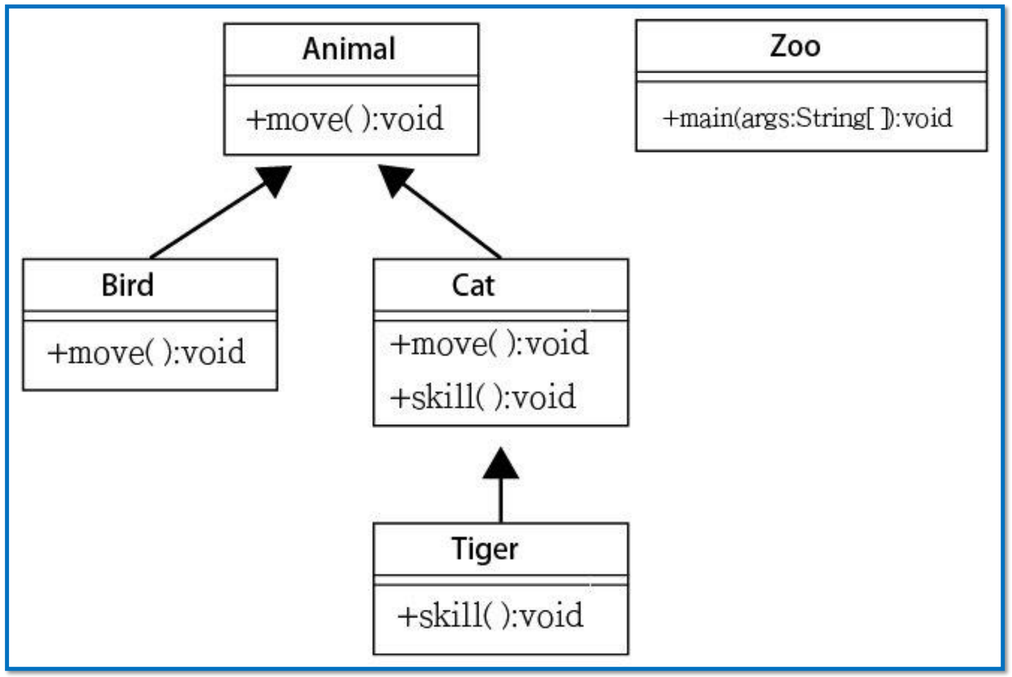
Cat c=new Tiger();

c.skill();

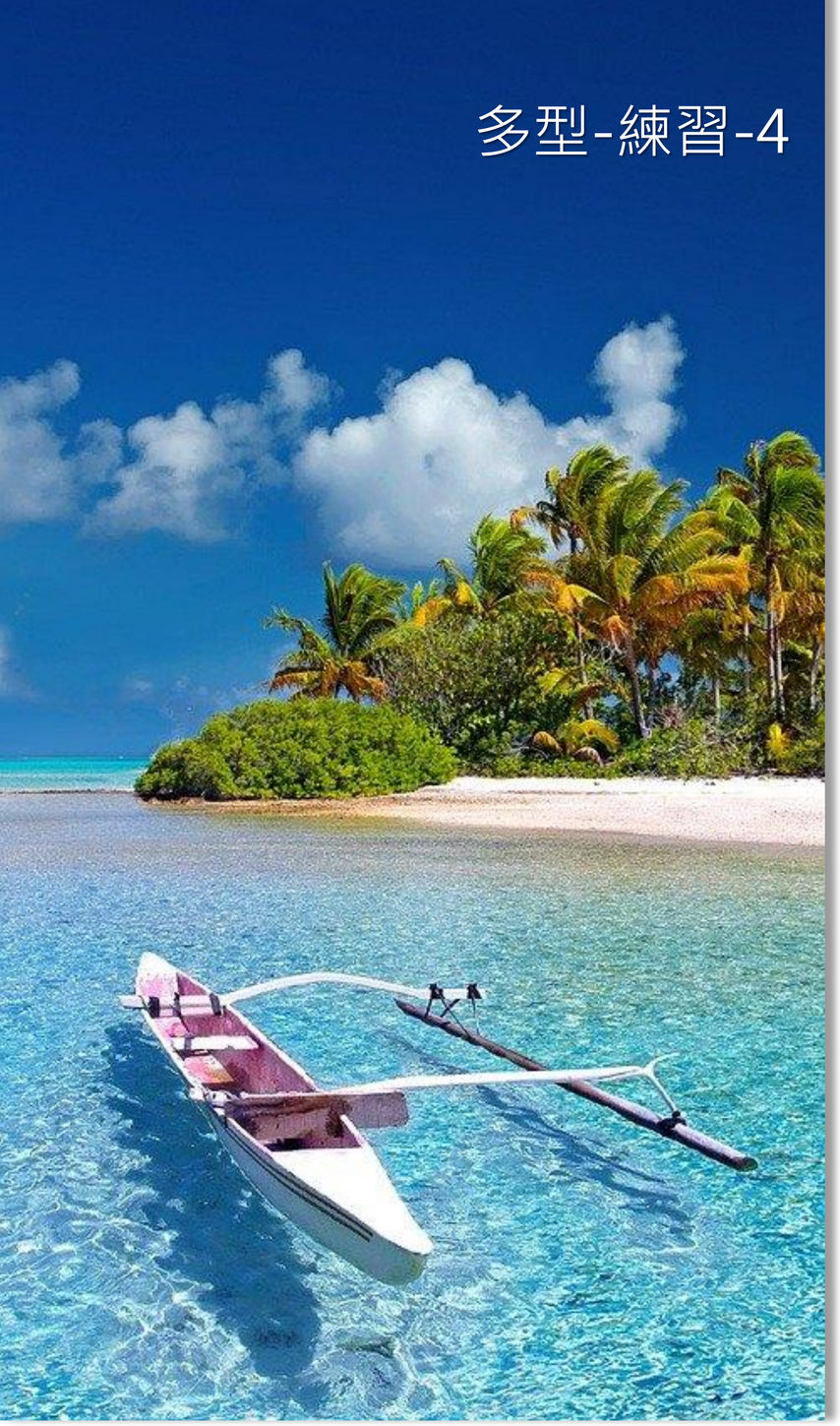
//呼叫是呼叫自己的skill方法,但因被子類Tiger覆寫了,所以執行的是Tiger的skill()方法

c.move();

//因為Tiger沒有move的方法,所以呼叫自己的move方法,沒有被覆寫



多型-練習-4



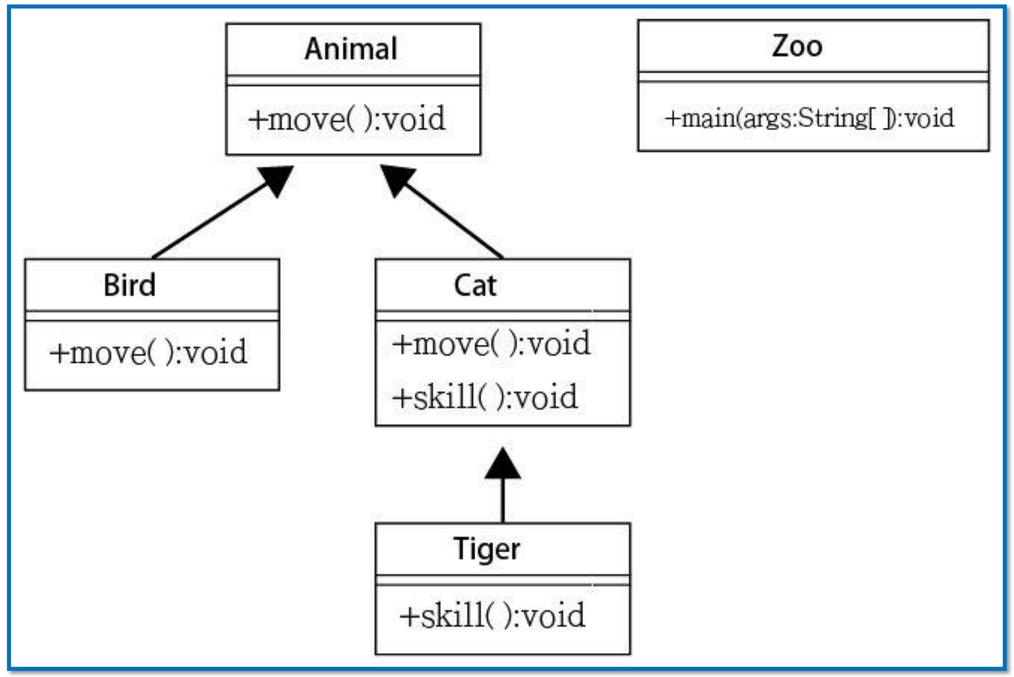
Animal a=new Tiger();

//a.skill();

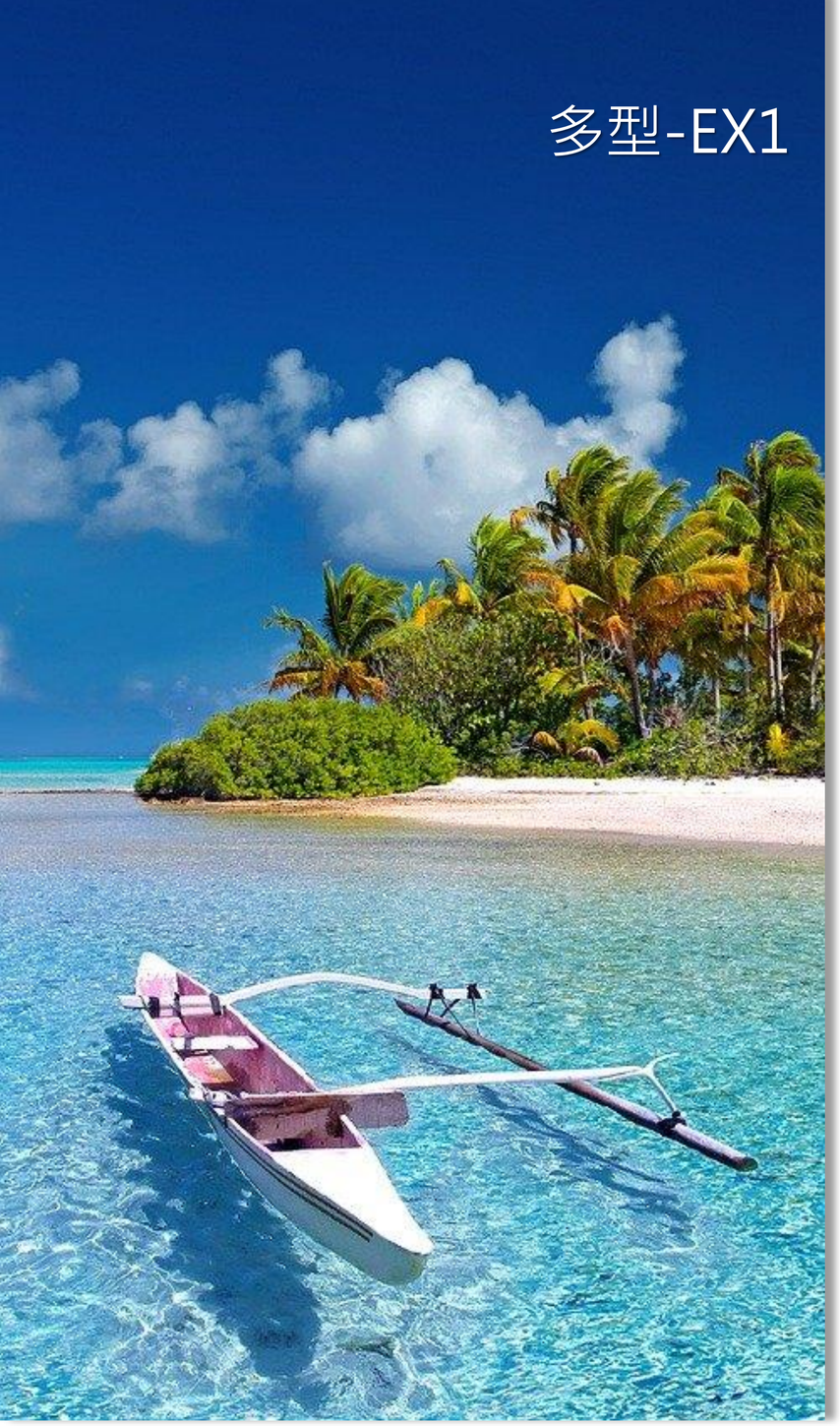
//此行會編譯錯誤,因為Animal並沒有skill的方法,父類別看不到子類別的成員,所以cannot find sysbol

a.move();

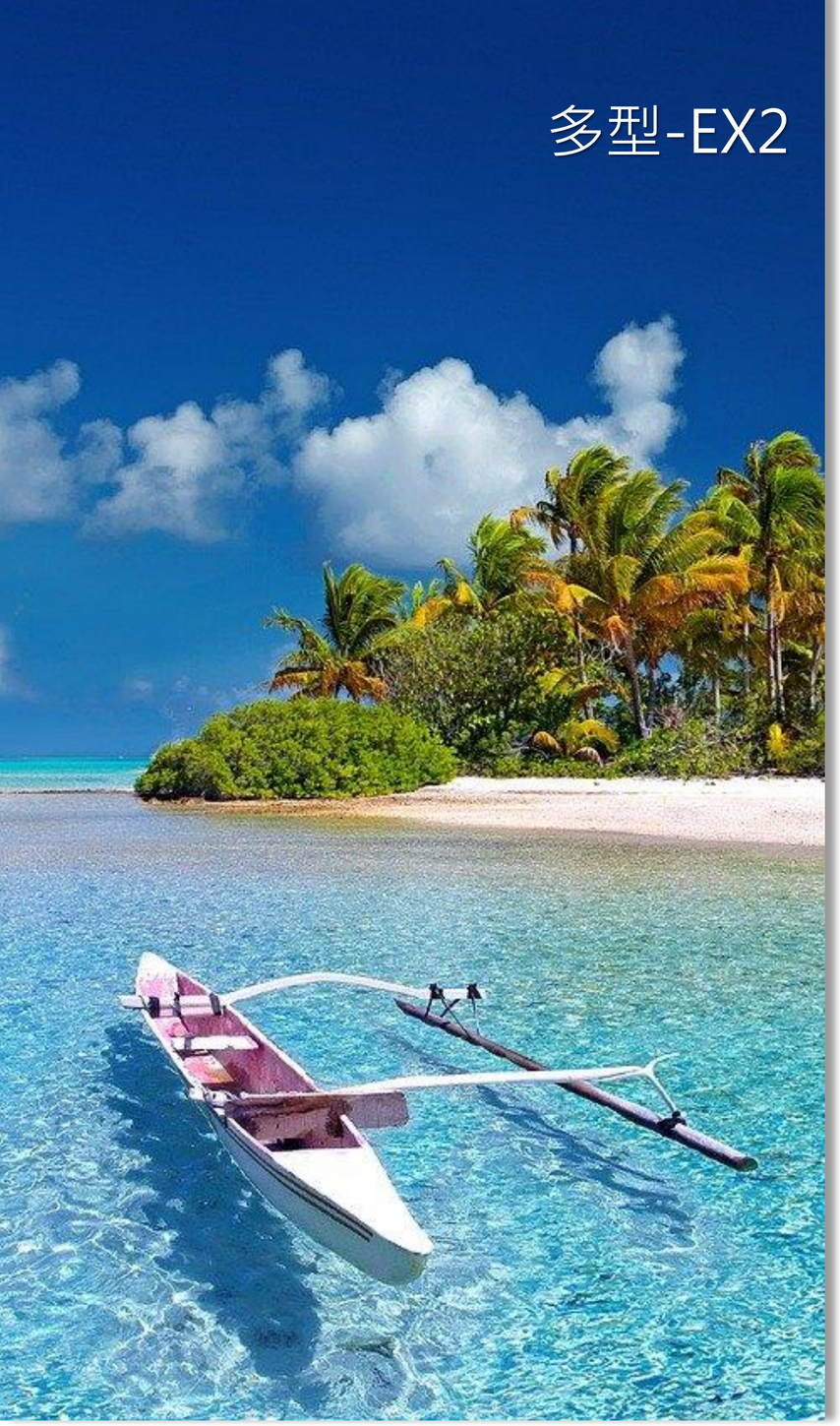
//因為Tiger沒有move的方法,所以呼叫Cat的move方法,animal的move方法被Cat覆寫



多型-EX1



```
public class Test2 {
    public static void main(String[] args) {
        Sub s=new Sub();
        System.out.println(s.count);
        s.display();
        System.out.println("==========");
        Base b=s;
        System.out.println(b==s);
        System.out.println(b.count);
        b.display();
    }
}

class Base{
    int count=10;
    public void display(){
        System.out.println(this.count);
    }
}
class Sub extends Base{
    int count=20;
    @Override
    public void display(){
        System.out.println(this.count);
        System.out.println(super.count);
    }
}
```



多型-EX2

```
class Person{
    protected String name="Person";
    protected int age=25;
    public void getInfo(){
        System.out.println("姓名:"+name+"\n年齡:"+age);
    }}
class Student extends Person{
    protected String school="台灣大學";
    @Override
    public void getInfo(){
        super.getInfo();
        System.out.println("\n學校名稱:"+school);
    }}
class Master extends Student{
    public String major="程式設計";
    @Override
    public void getInfo(){
        super.getInfo();
        System.out.println("\n主修課程:"+major);
    }
}
```

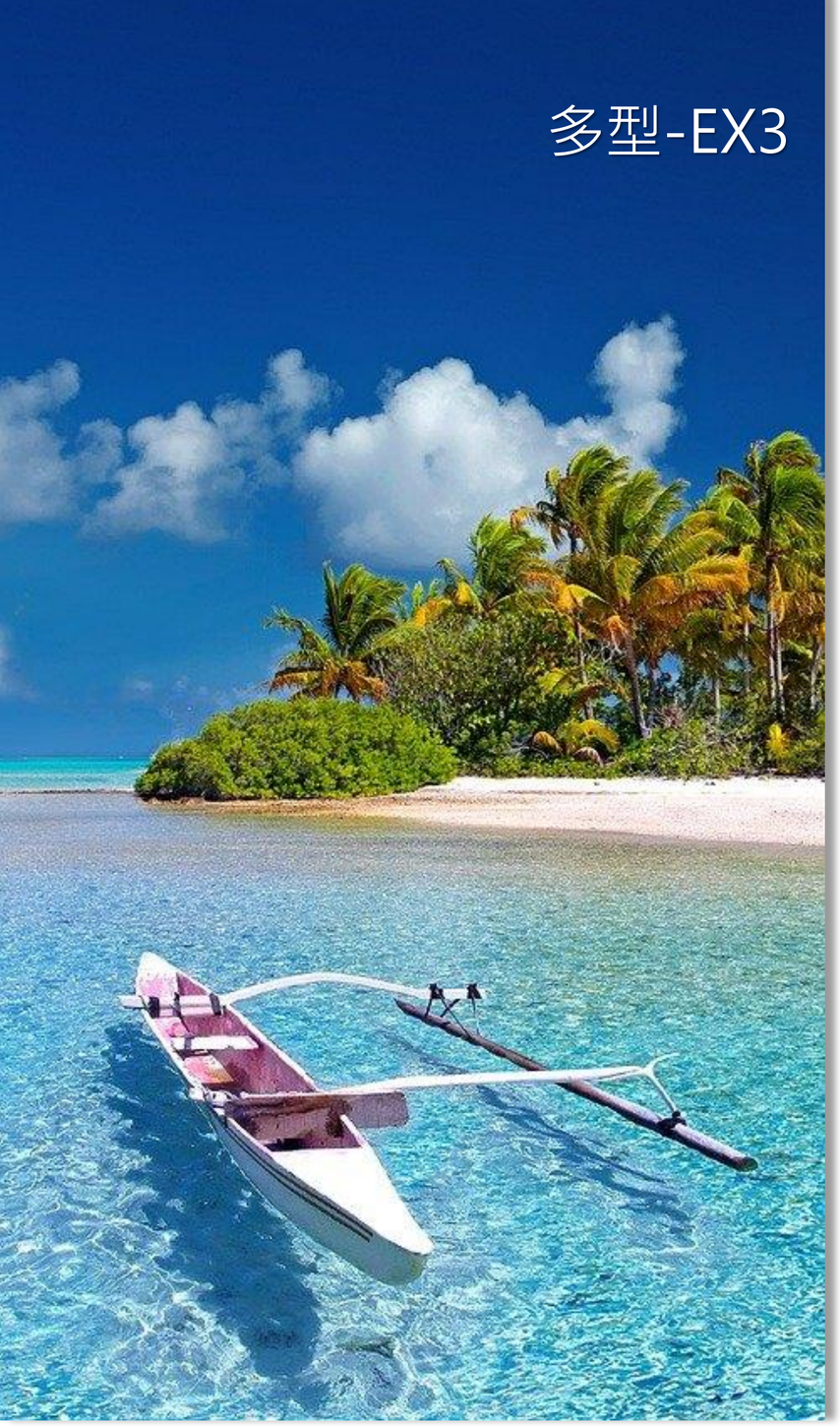
●撰寫程式解決如下問題

●建立PolyTest類別,在類別中定義方法 method(Person e)

●在method中:

- ■根據e的類型調用相對的類別的getInfo()方法.
- ■根據e的類型執行
- ◆如果為Person類的物件,輸出"是人類 "
- ◆如果為Student類的物件,輸出"是人類","是學生 "
- ◆如果為Master類的物件,輸出"是人類","是學生","是研究生"

多型-EX3



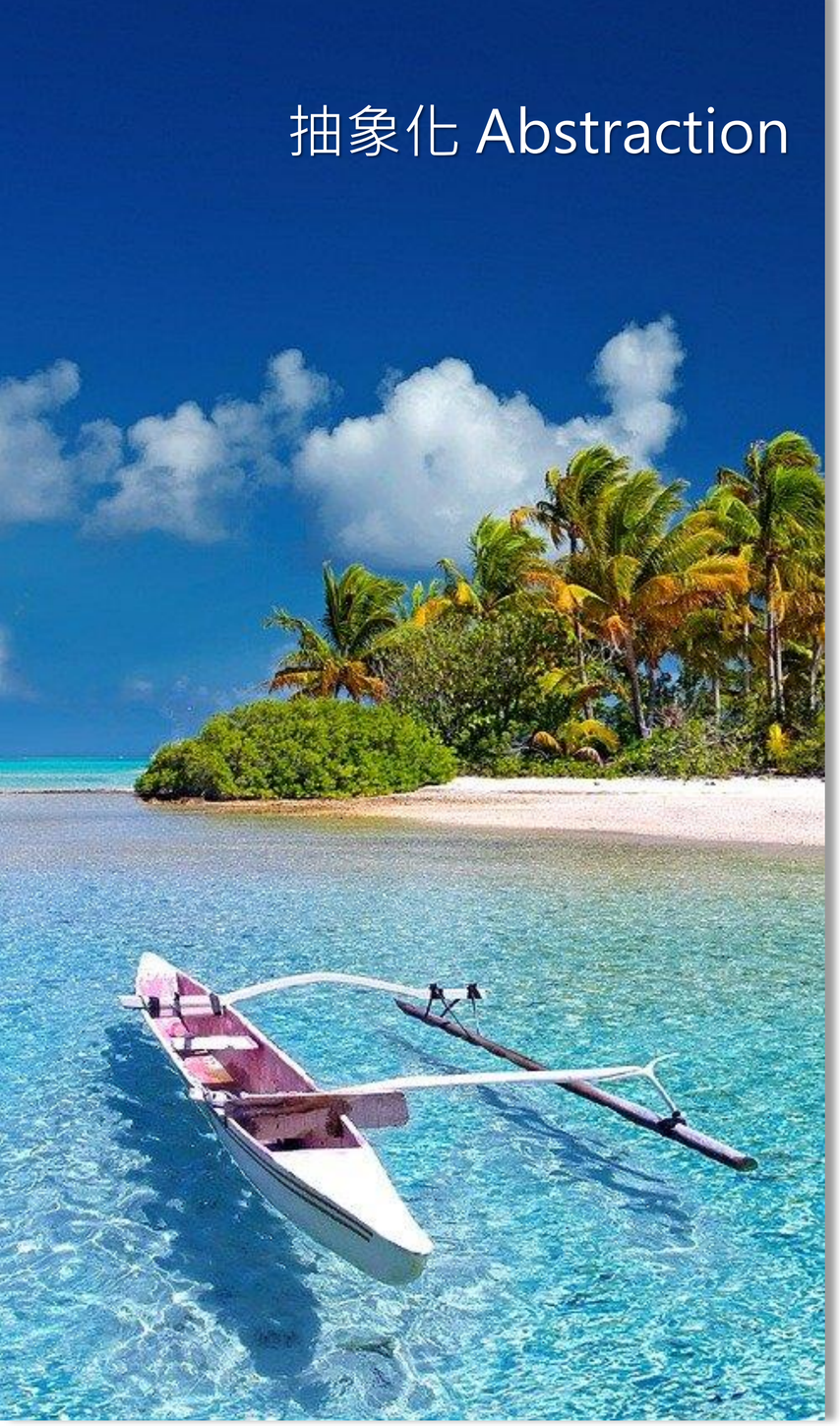
```
class Father {
    void sum(int a, int... b) {
        System.out.println("Father");
    }
}
class Son extends Father {
    void sum(int a, int[] b) {
        System.out.println("Son");
    }
//  void sum(int a, int b, int c) {
//      System.out.println("Son-part2");
//
//}
}
```

主方法程式如下

```
public static void main(String[] args) {
        Father f = new Son();
        f.sum(1,2,3);
//      ((Son) f).sum(1, 2, 3);
    }
```

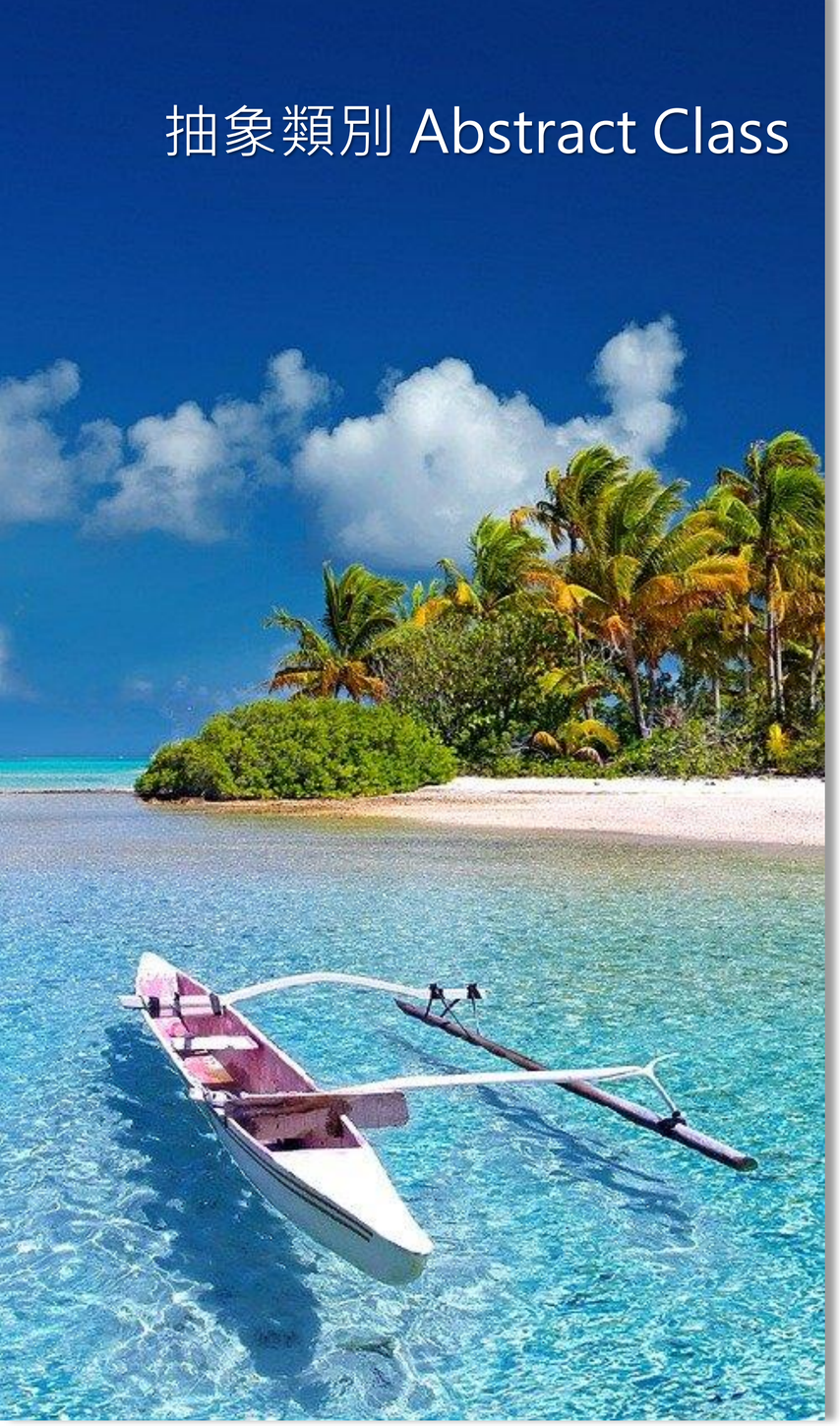
抽象 ABSTRACT

抽象化 Abstraction



- 物件導向語言提供抽象化的特性
  - 將實作類別與使用服務的類別解構/分開
  - 將實作類別宣告成抽象的父類別或父介面
- 抽象化的優點
  - 可抽換的實作
    - 只要定義出溝通的介面，可由不同廠商提供實作
  - 易於維護
    - 實作邏輯變更時，只須修正/抽換實作類別，不影響使用該實作的程式
  - 專業分工
    - 將商業邏輯與使用者介面分開，讓不同專長工程師開發

抽象類別 Abstract Class



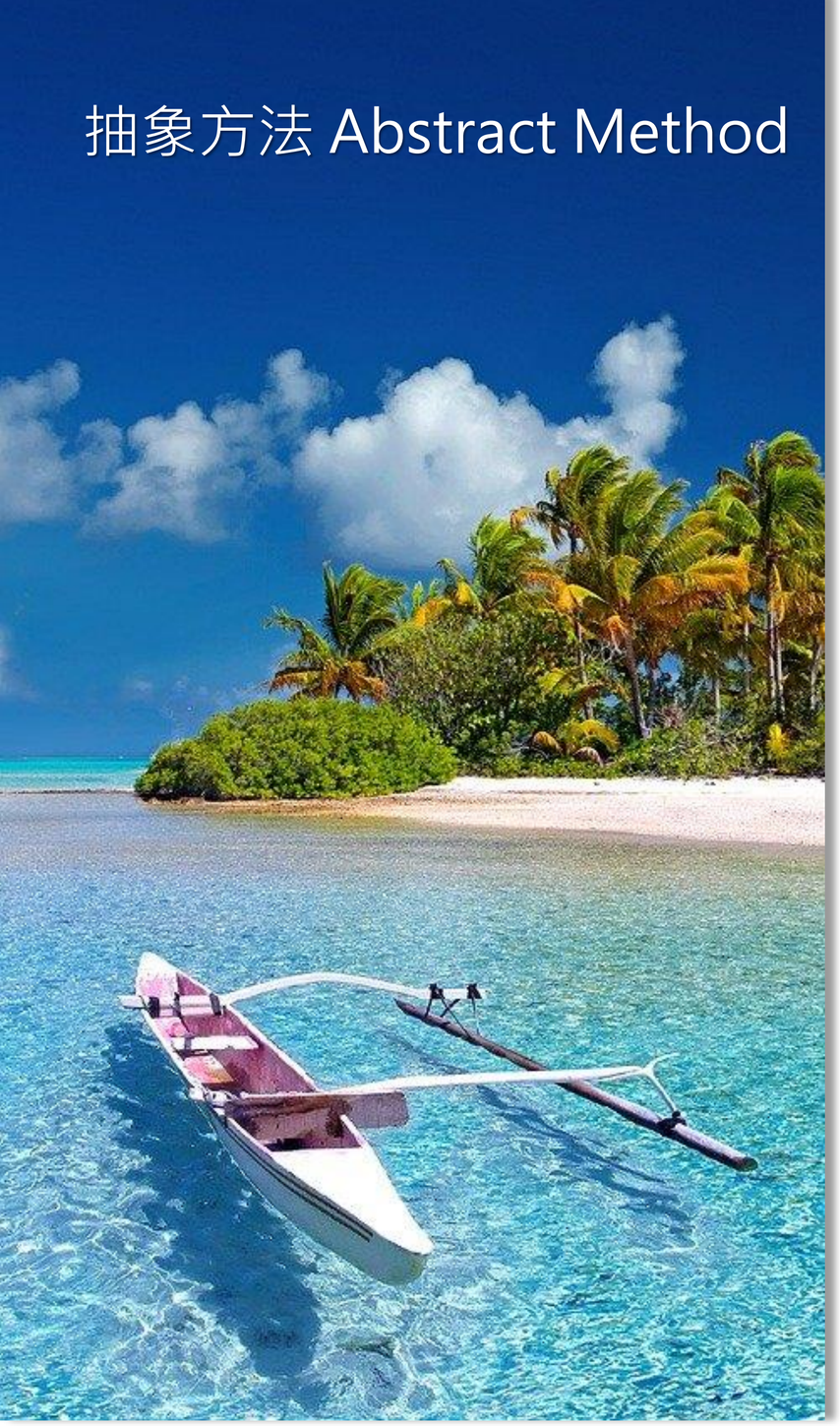
一

- 抽象類別
  - 類別只要有一個以上的抽象方法，就必須為抽象類別
  - 語法:

```
abstract class <Name> {
    ……
}
```

  - 抽象類別不能建構物件實體
  - 抽象類別仍可以有建構子，屬性與具體方法
    - 建構子用來初始化實體成員，只能由子類別用 super( ) 呼叫
  - 繼承自抽象類別的子類別一定要實作抽象方法，除非繼續宣告為抽象類別

抽象方法 Abstract Method

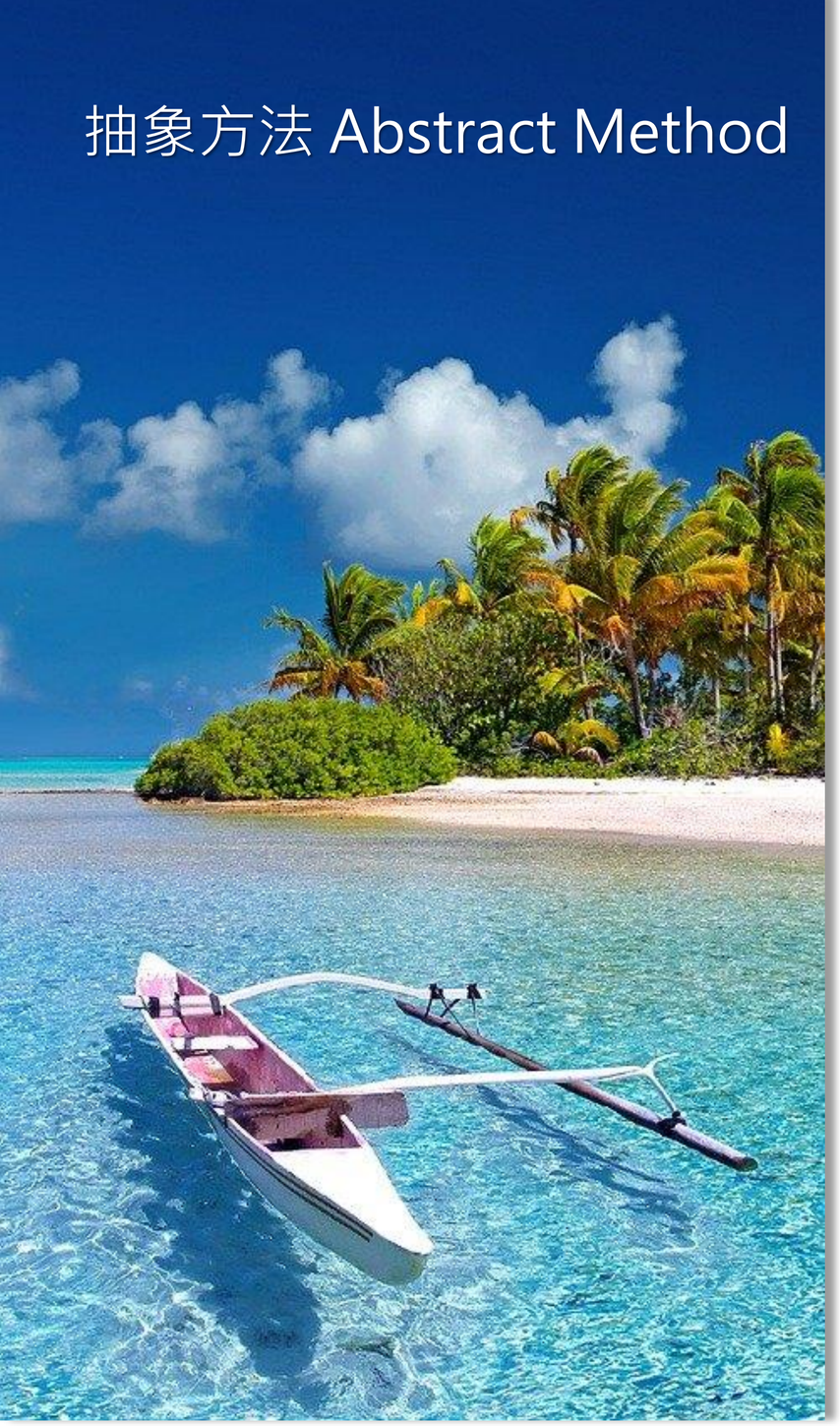


- 抽象方法：存取權限不能為private，且不能為static
  - 只宣告方法定義,而沒有撰寫方法的內容(本體)
  - 語法

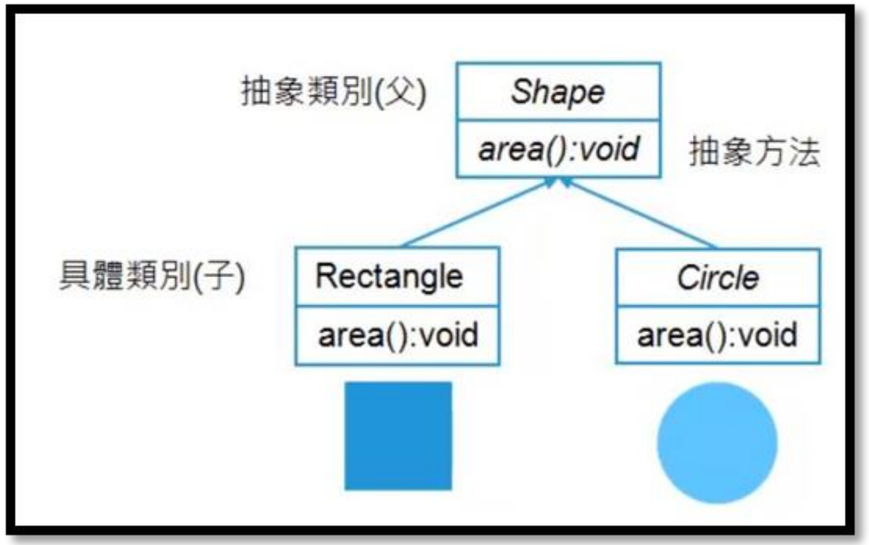
```
abstract <return_type> <name> ([arguments]);
```

- 用途
  - 同樣要有某個方法，但實作方法卻大不相同
  - 使多型可以使用
  - 抽象方法強迫有繼承關係的子類別去覆寫此方法

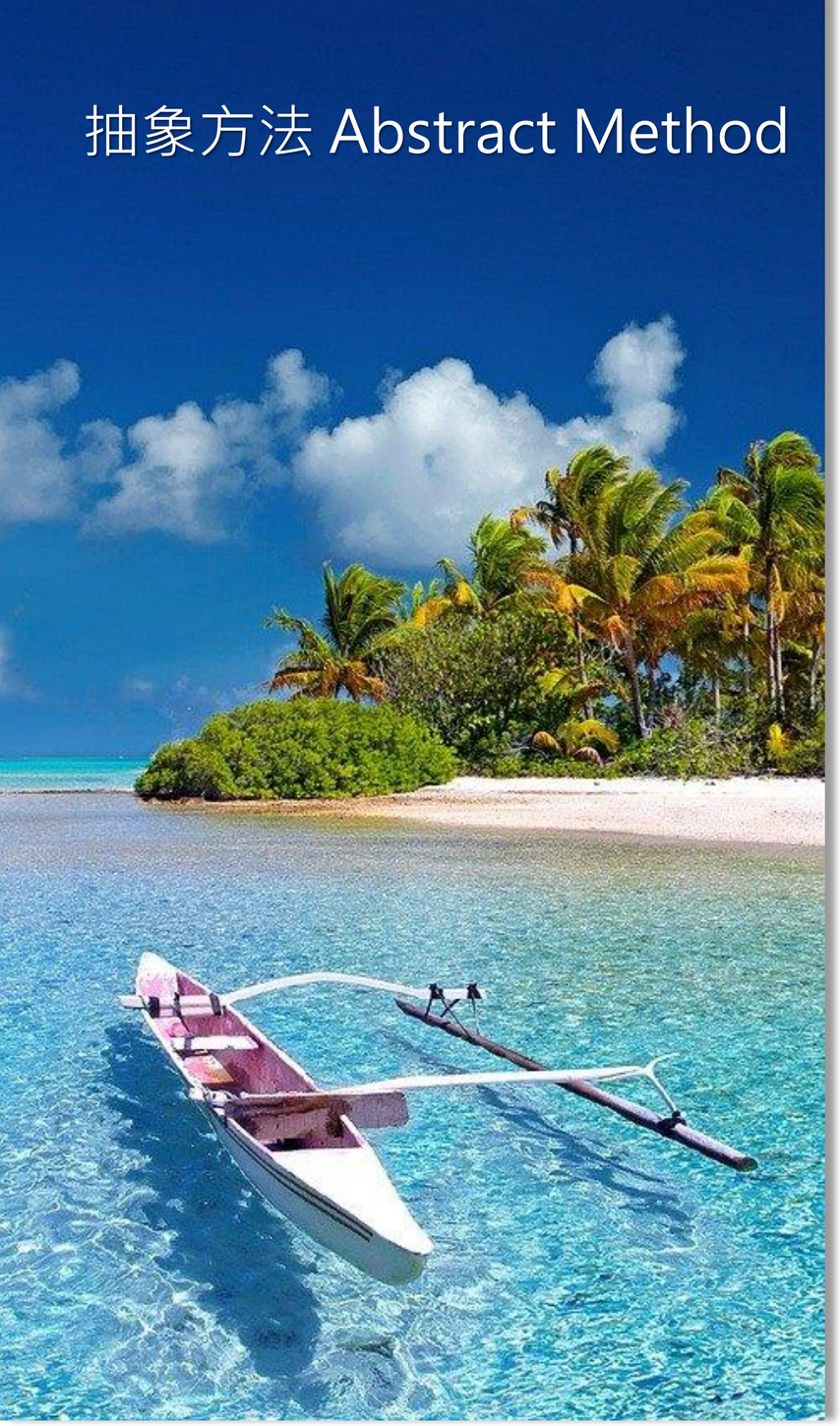
抽象方法 Abstract Method

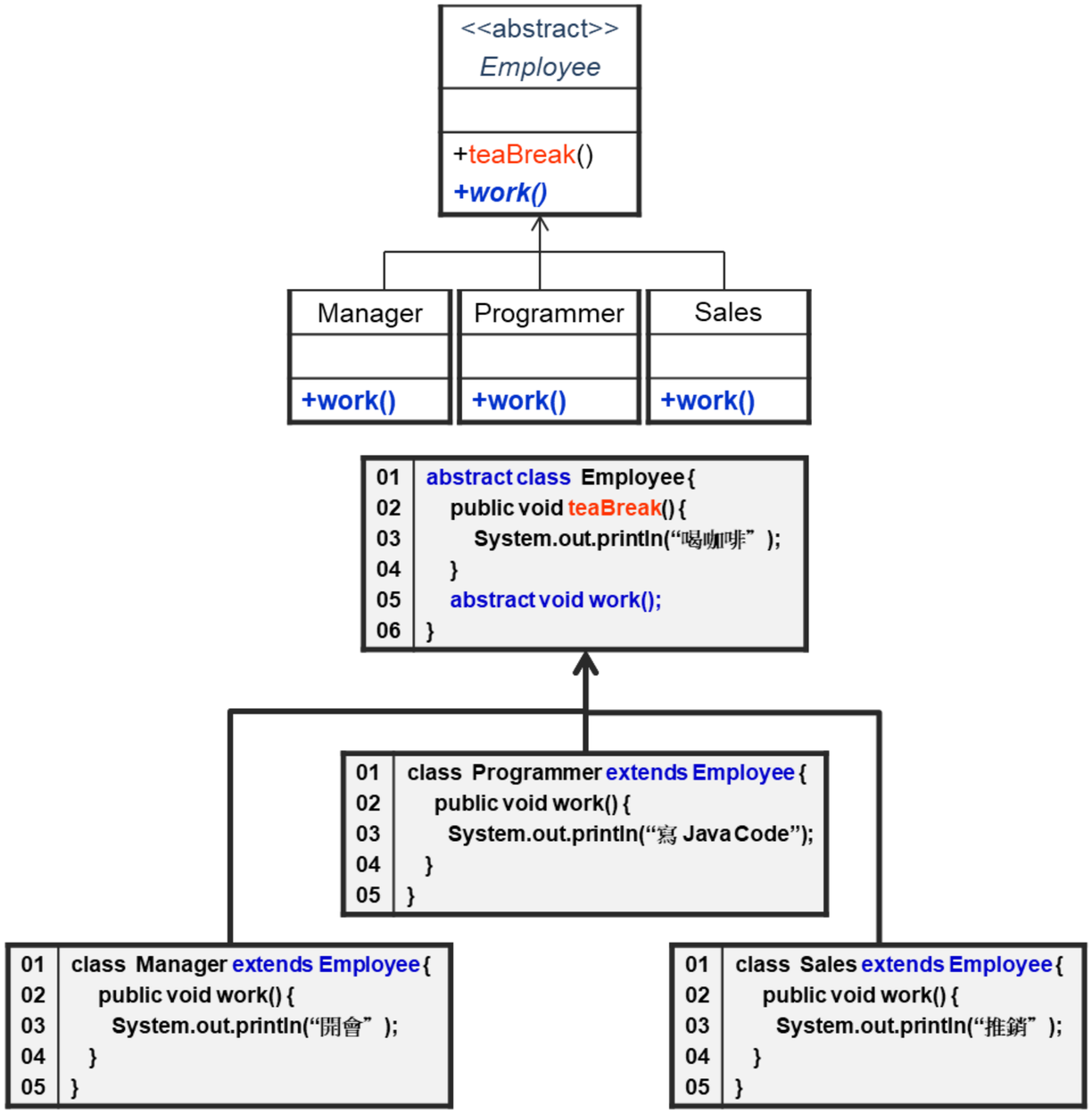


```
abstract class Shape {
    public abstract void area();
}
class Rectangle extends Shape {
    private int length;
    private int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }
    @Override
    public void area() {
        System.out.println("長為:" + length + "寬為:" + width + "的矩形面積為:" + length * width);
    }
}
class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public void area() {
        System.out.println("半徑為:" + radius + "的圓面積為:" + Math.PI * Math.pow(radius, 2));
    }
}
```



抽象方法 Abstract Method



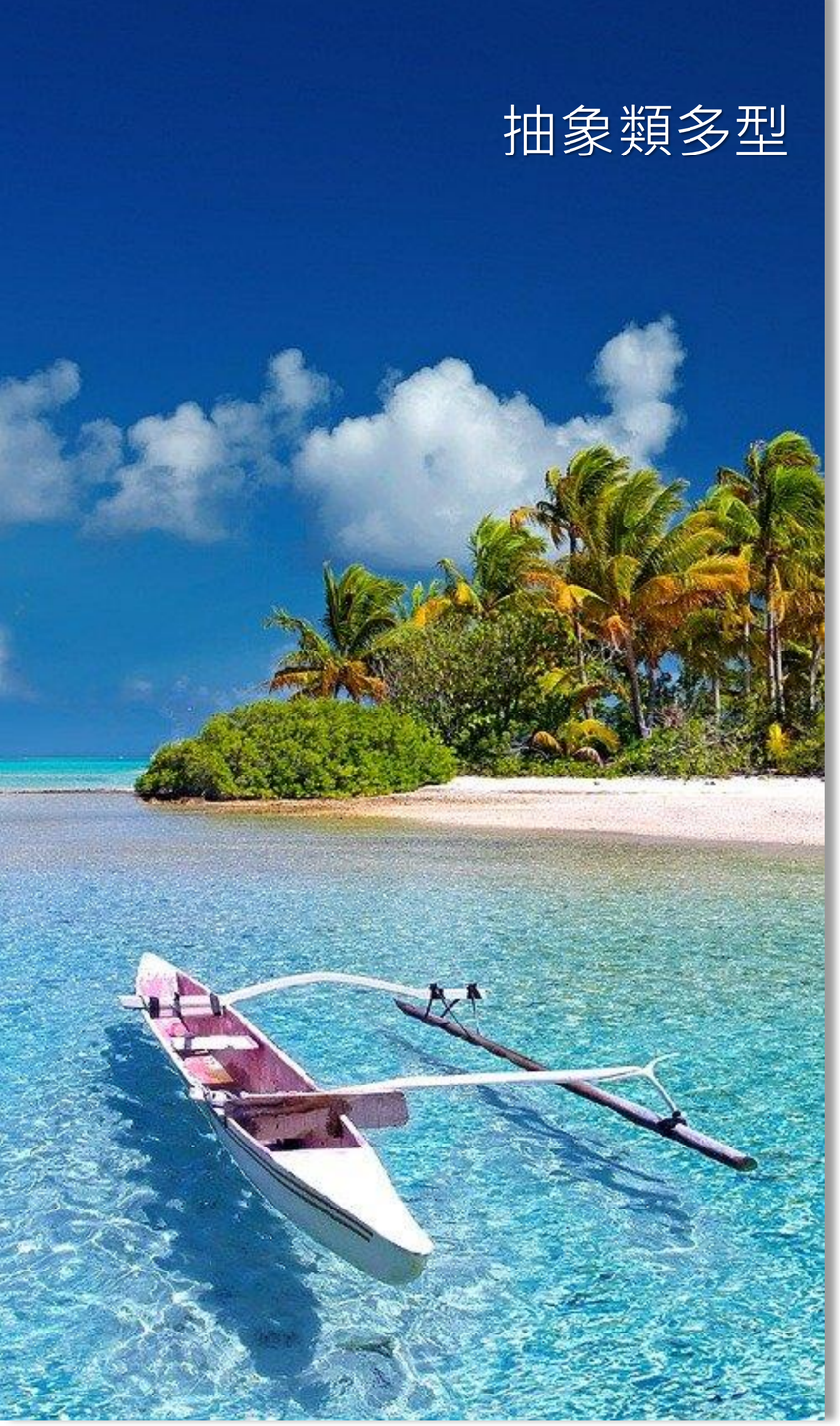



```
01  abstract class  Employee{
02      public void teaBreak(){
03          System.out.println("喝咖啡");
04      }
05      abstract void work();
06  }
```

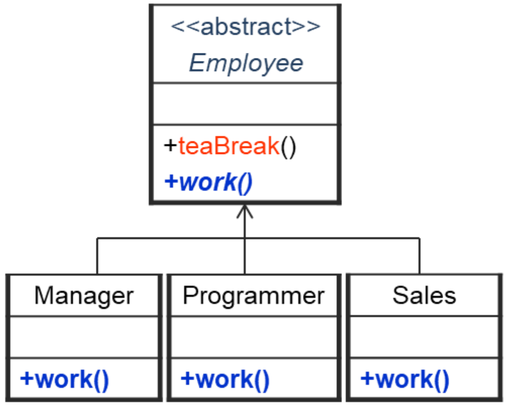
```
01  class  Programmer extends Employee{
02      public void work(){
03          System.out.println("寫 Java Code");
04      }
05  }
```

```
01  class  Manager extends Employee{
02      public void work(){
03          System.out.println("開會");
04      }
05  }
```

```
01  class  Sales extends Employee{
02      public void work(){
03          System.out.println("推銷");
04      }
05  }
```

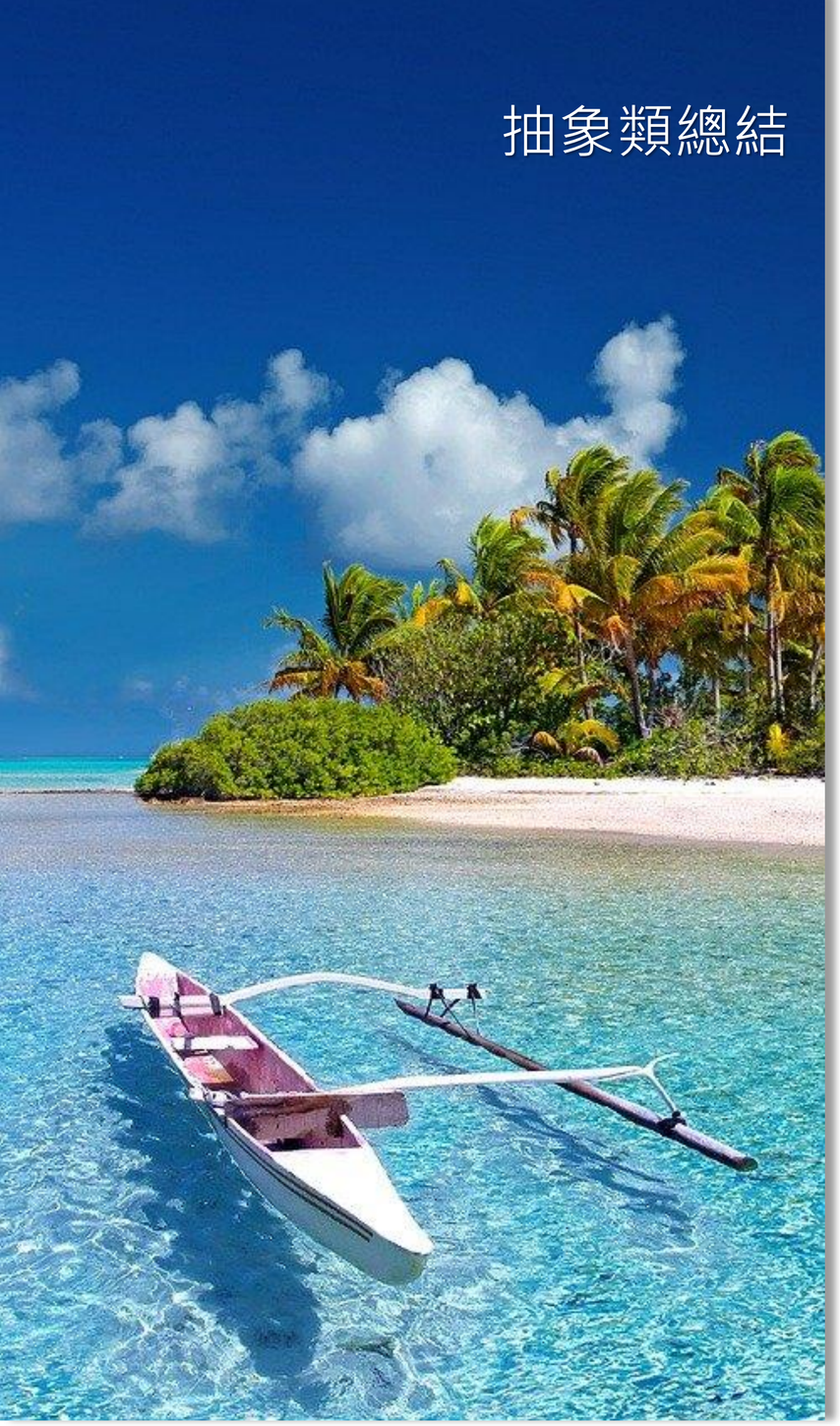


抽象類多型



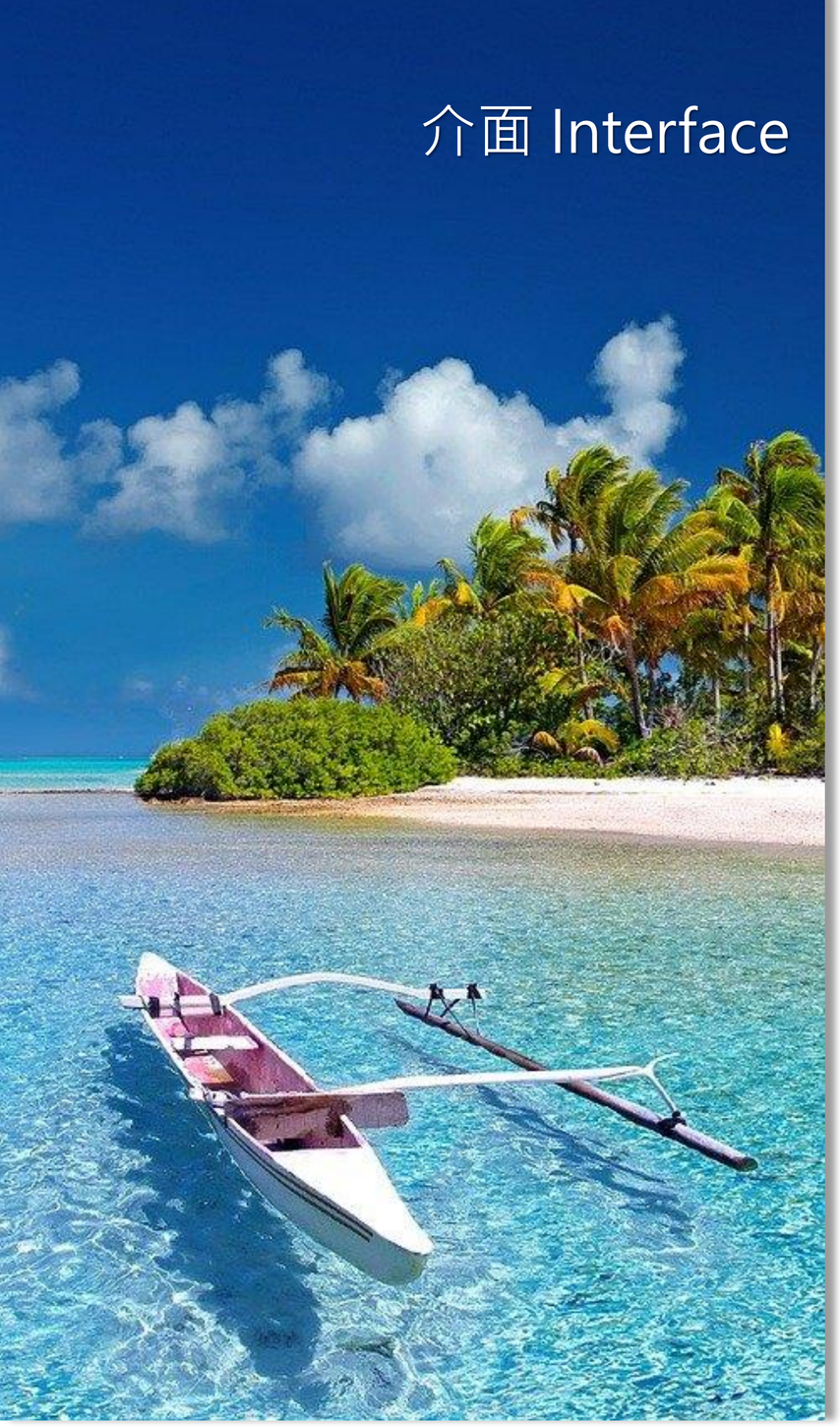

```
01  public class  EmployeeTest{
02      public static void main(String[] args) {
03
04          Employee manager = new Manager();
05          Employee programmer = new Programmer();
06          Employee sales= new Sales();
07
08          manager.work();
09          programmer.work();
10          sales.work();
11
12          manager.teaBreak();
13          programmer.teaBreak();
14          sales.teaBreak();
15
16      }
17  }
```

抽象類總結



- 抽象類別無法使用new關鍵字建構實例
- 抽象類別讓方法的實作由子類別來決定，使其使用更多樣化
- 抽象方法不可以是 static 類別方法
- 若父類別是一個抽象類別，其所產生的抽象方法都必須由子類別來重新實作，有點類似實作介面類別的方法，如果子類別沒有全部實作出父類別所提供的所有抽象方法，那麼子類別也必須宣告成抽象類別，且子類別在使用父類別的抽象方法時，一定要覆寫才能使用並實作
- 定義類別的時候，先在父類別定義出子類別必須共同遵守的行為，但父類別的目的只是先定義而無需實作此行
- 抽象類別的抽象方法在定義時，abstract這個關鍵字是一定要加的，且存取權限不可是private。且不能有方法實作區塊{.....}
- 抽象方法必須為public protected default，預設情況下預設為public。
- 當類別中有抽象方法，即有abstract修飾語之方法，則此類別需宣告為抽象類別，且抽象類別無法被實體化

介面 INTERFACE



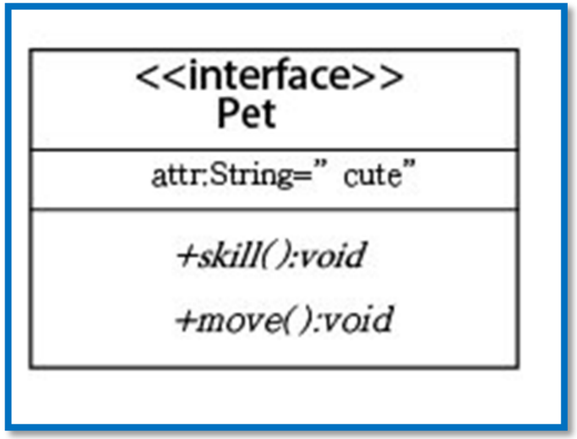
介面 Interface

- 介面也是類別的一種，但和類別是並列關係，它是用戶端程式與類別之間的溝通 利用相同的介面可讓程式撰寫時有一定的規範何循，有時必須從幾個類別中抽取出一些共同的行為特徵,但它們之間又沒有is-a的關係，僅僅只是具有相同的行為特徵而已，例如　滑鼠　鍵盤　手機　硬碟　攝影機　麥克風　都支持ＵＳＢ的連接。但沒有繼承關係，不能講手機是一種ＵＳＢ，介面類別在設定完成後就不允許更改，否則就失去了定義使用者介面類別的意義，介面還有一個很重要的特性是利用它來收集多個類別，而這些類別彼此可以不用建立任何關聯，這種特性讓 JAVA　單一繼承的限制有出口
- 類別需先繼承(extends)類別而後實作(implements)介面,此即為類優先(但只針對方法,屬性並無類優先規則)
- 類別可以實作(implements)多個介面,但無法繼承(extends)多個類別
- 介面可以繼承(extends)多個介面,但無法實作(implements)多個介面

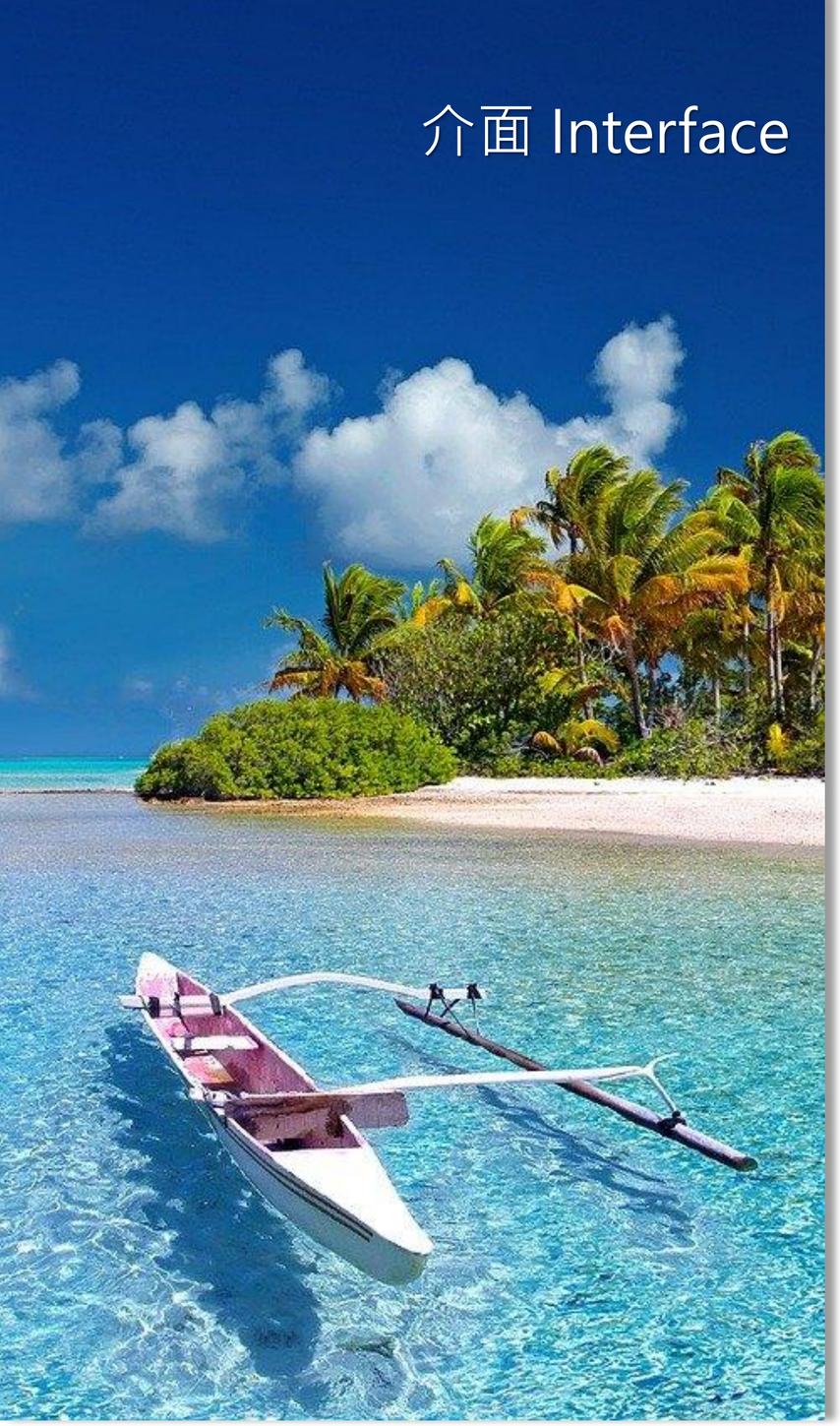
語法

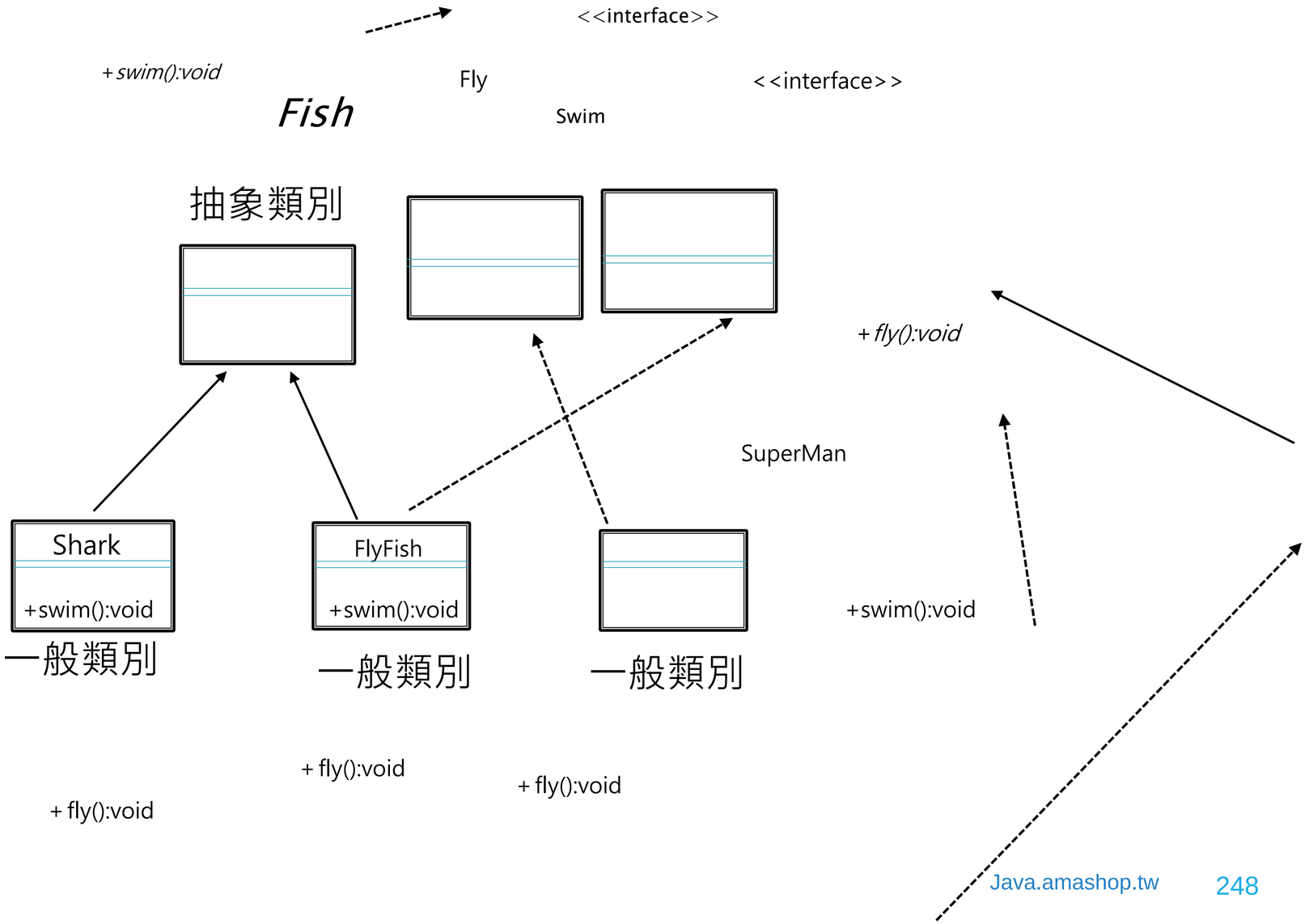
```
[存取修飾字] interface 介面名稱 [extends 父介面] {
      //宣告常數
      [存取修飾字] [修飾字] 資料型別 名稱=初始值;
            :
      //宣告方法
      [存取修飾字] [修飾字] 回傳值型別 方法名稱(參數列);
            :
}
```

- 存取修飾字：[public] 或 不撰寫
- 宣告常數修飾字：[static final] 或 不撰寫
- 宣告方法修飾字：[abstract] 或 不撰寫

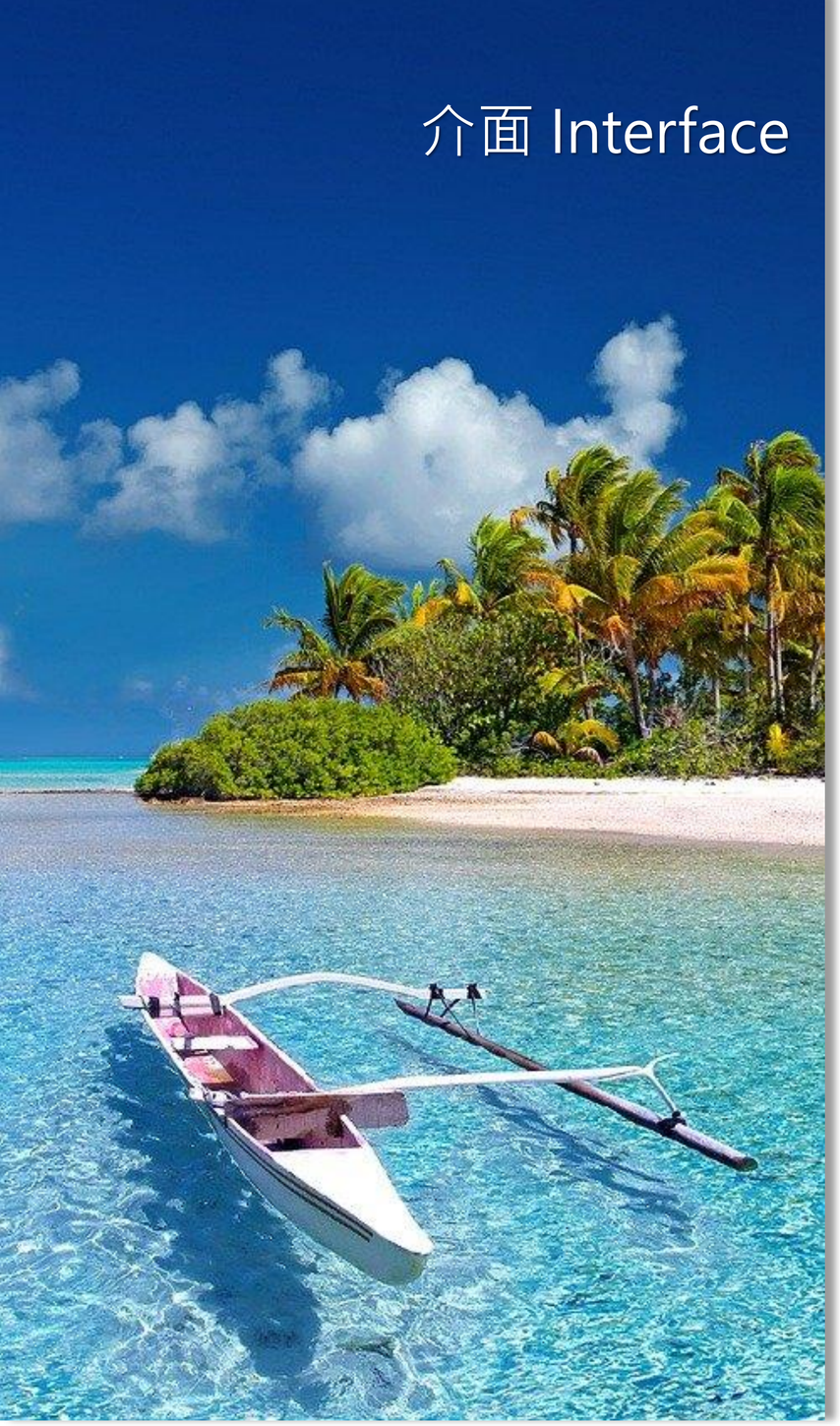


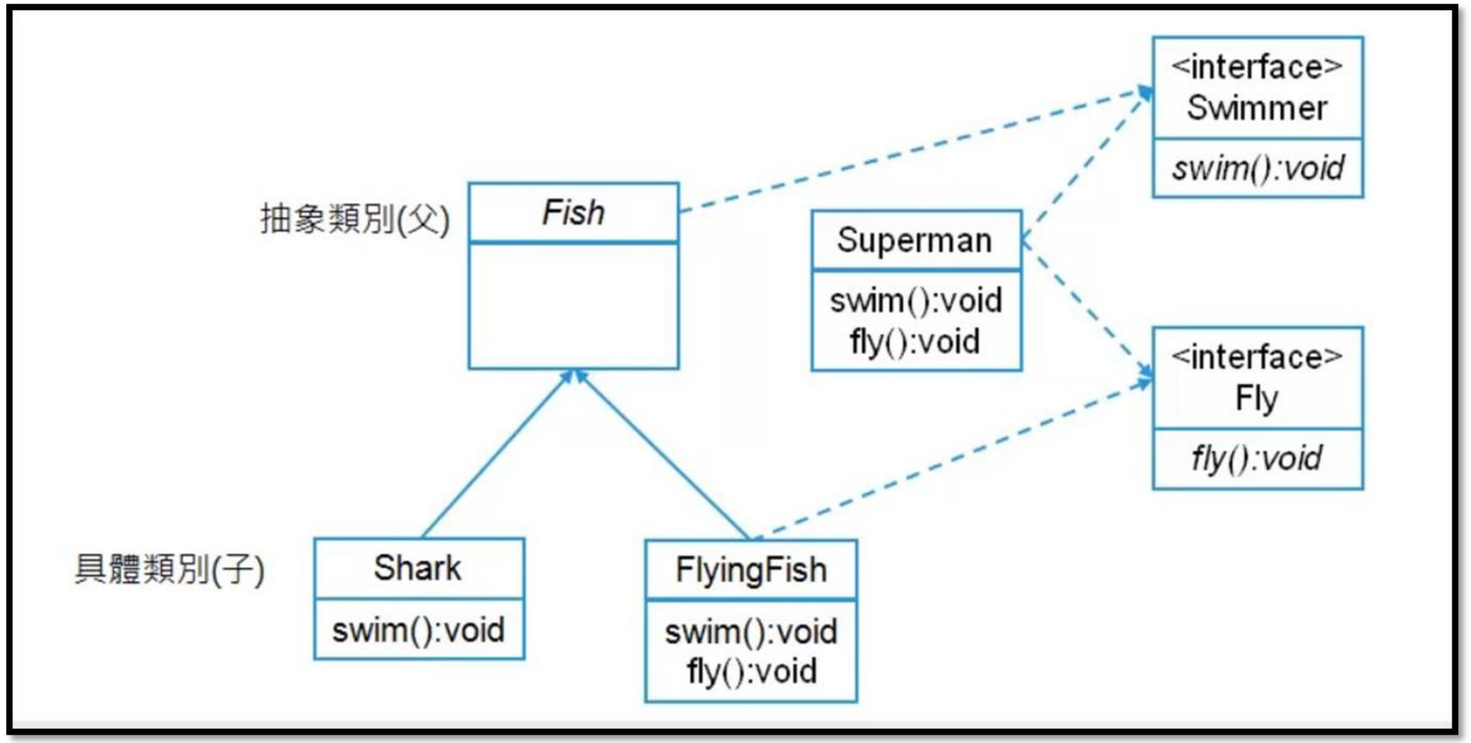
介面 Interface



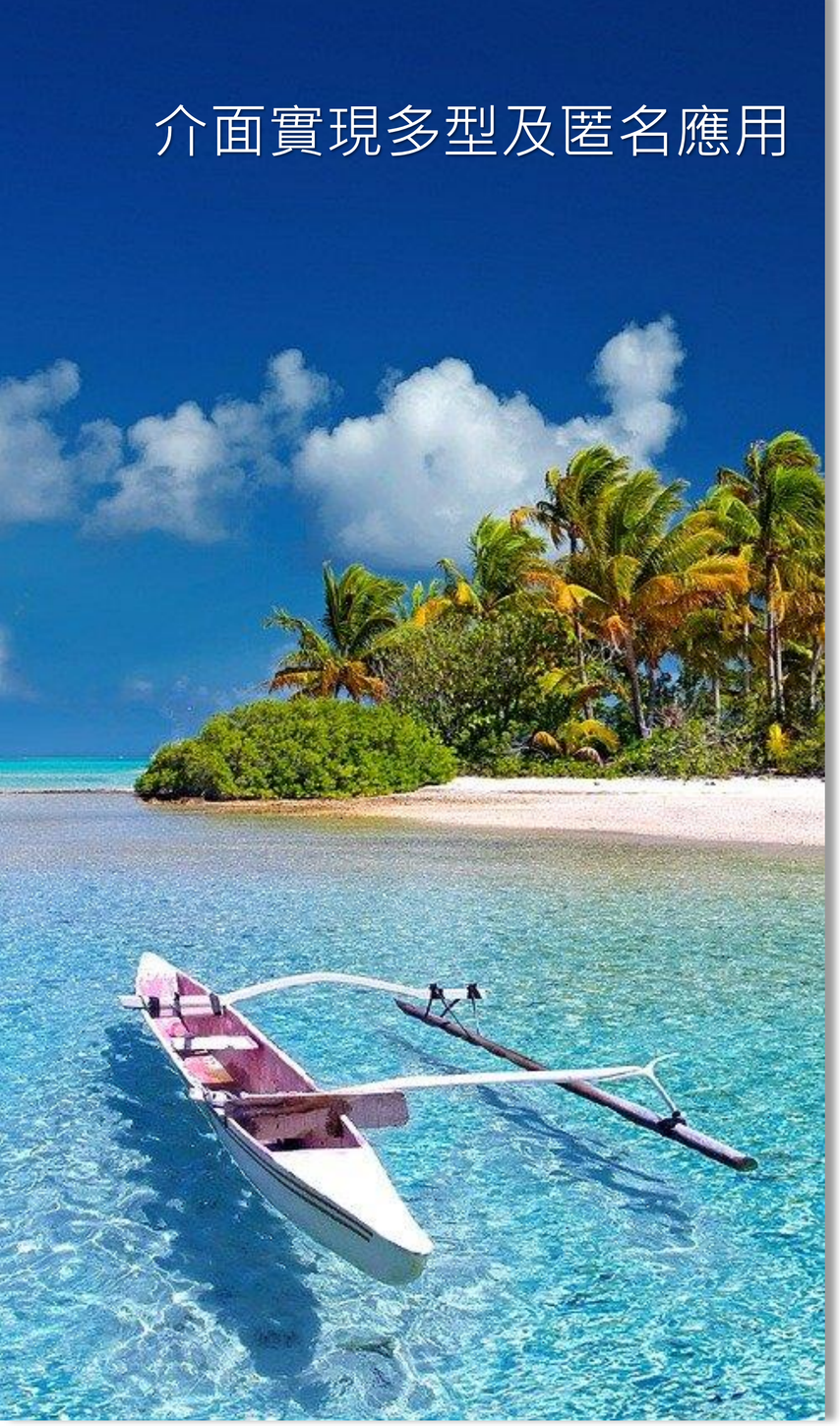


介面 Interface





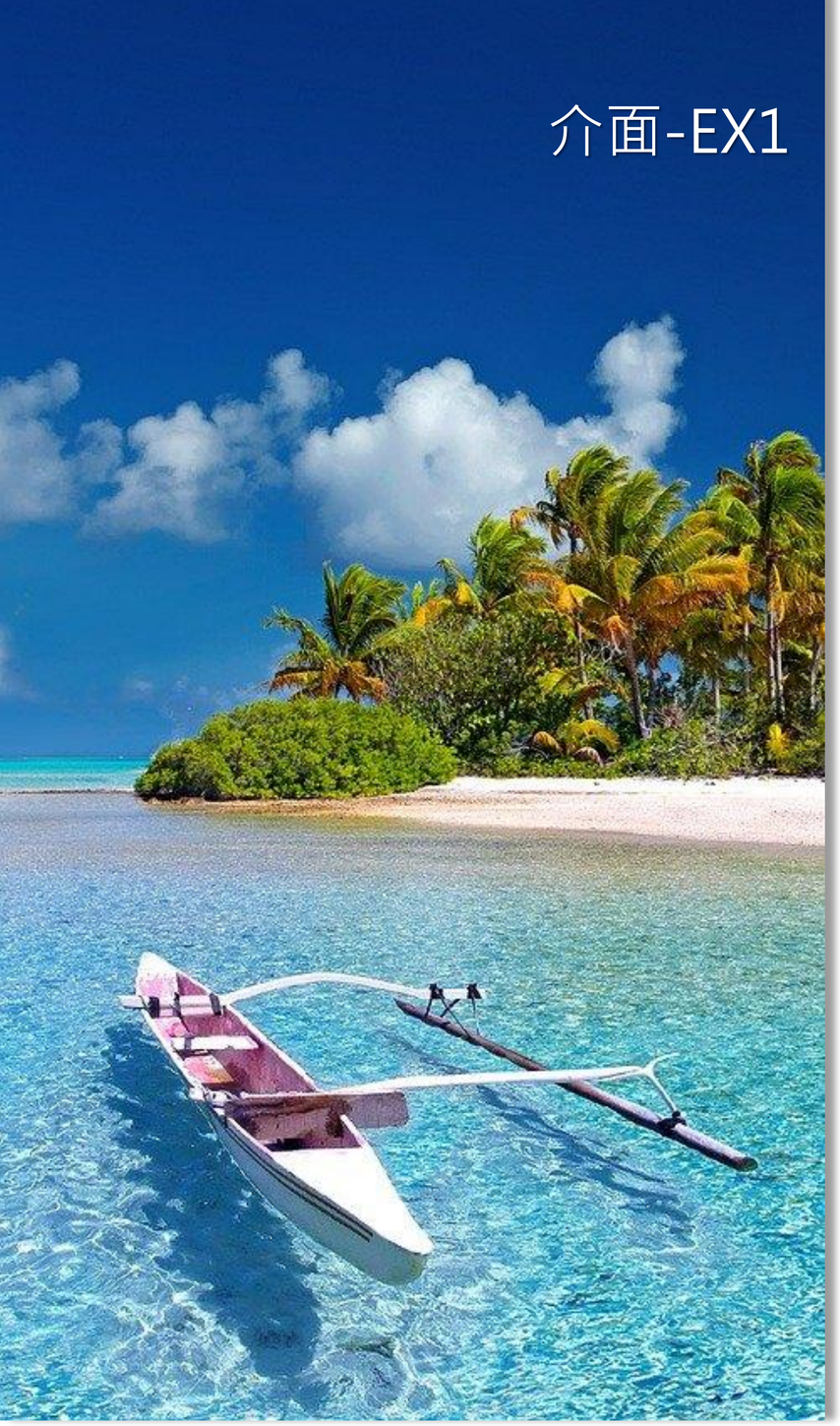
介面實現多型及匿名應用



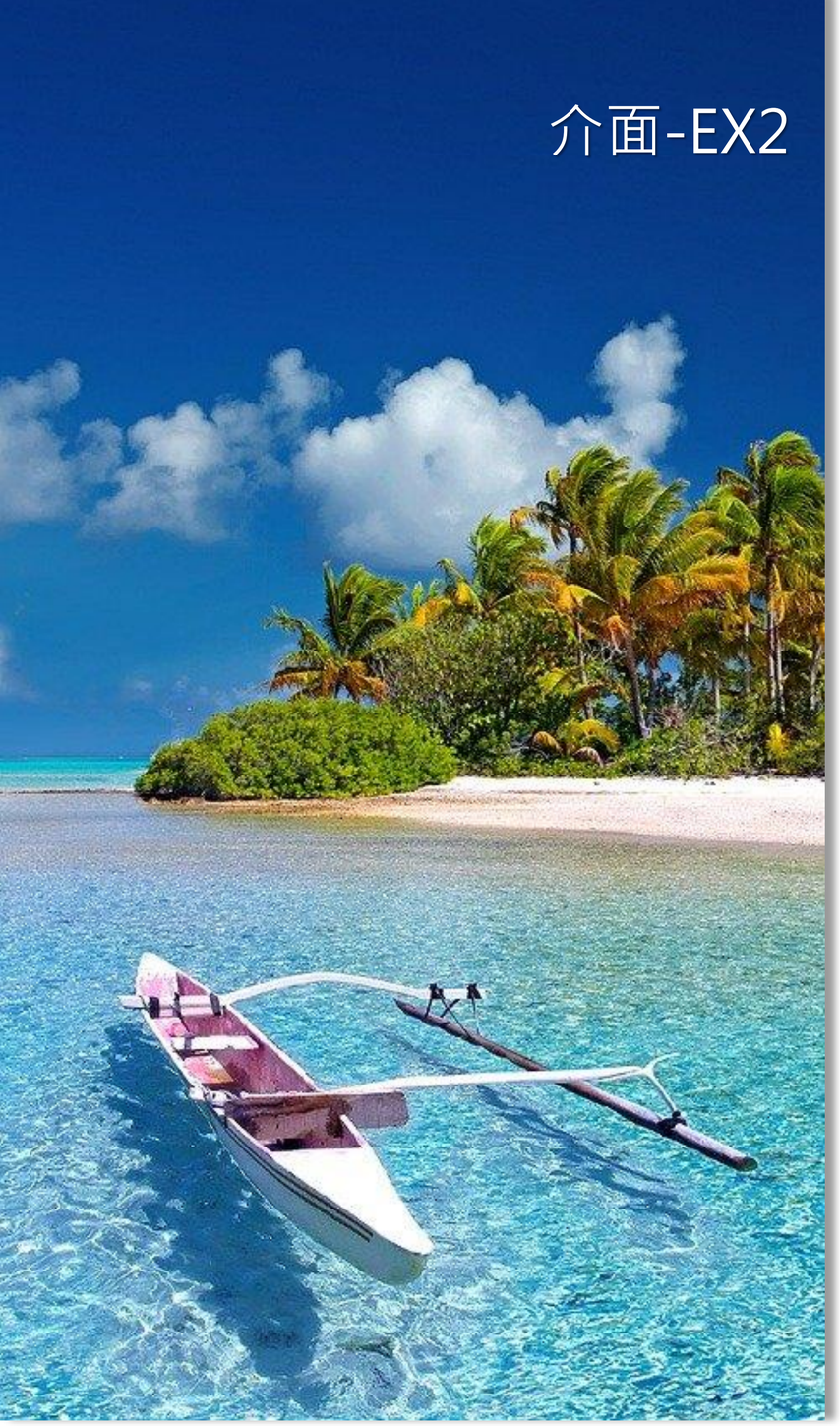
```
//1.建立實體物件
    Computer pc=new Computer();
    Flash flash=new Flash();
    pc.transData(flash);
    //2.建立匿名物件
    pc.transData(new Printer());//此即為介面具體實現多型的應用
    //3.建立實體物件但為匿名類別
    Usb mp3=new Usb(){
       @Override
       public void start() {
          System.out.println("mp3開始啟動");
       }

       @Override
       public void stop() {
          System.out.println("mp3結束運作");
       }
    };
    pc.transData(mp3);
    //4.匿名類別及匿名物件
    pc.transData(new Usb(){
       @Override
       public void start() {
           System.out.println("手機開始運作");
       }

       @Override
       public void stop() {
           System.out.println("手機結束運作");
       }

    });
}
}
```

```
class Computer{
   void transData(Usb usb){
      usb.start();
      usb.stop();
   }
}
interface Usb{
 //定義常數例長,寬,高,深度,....
   void start();
   void stop();
}
class Flash implements Usb{

   @Override
   public void start() {
      System.out.println("隨身碟開始運作");
   }
   @Override
   public void stop() {
      System.out.println("隨身碟結束運作");
   }
}
class Printer implements Usb{

   @Override
   public void start() {
      System.out.println("印表機開始運作");
   }
   @Override
   public void stop() {
      System.out.println("印表機 結束運作");
   }
}
```

介面-EX1



```
public class NewClass1 {
    public static void main(String[] args)        {
                                new C().test();
                }
}
interface A{
    int v=1;
}
class B{
    int v=2;
}
class C extends B implements A{
    void test(){
        System.out.println(v);
    }
```

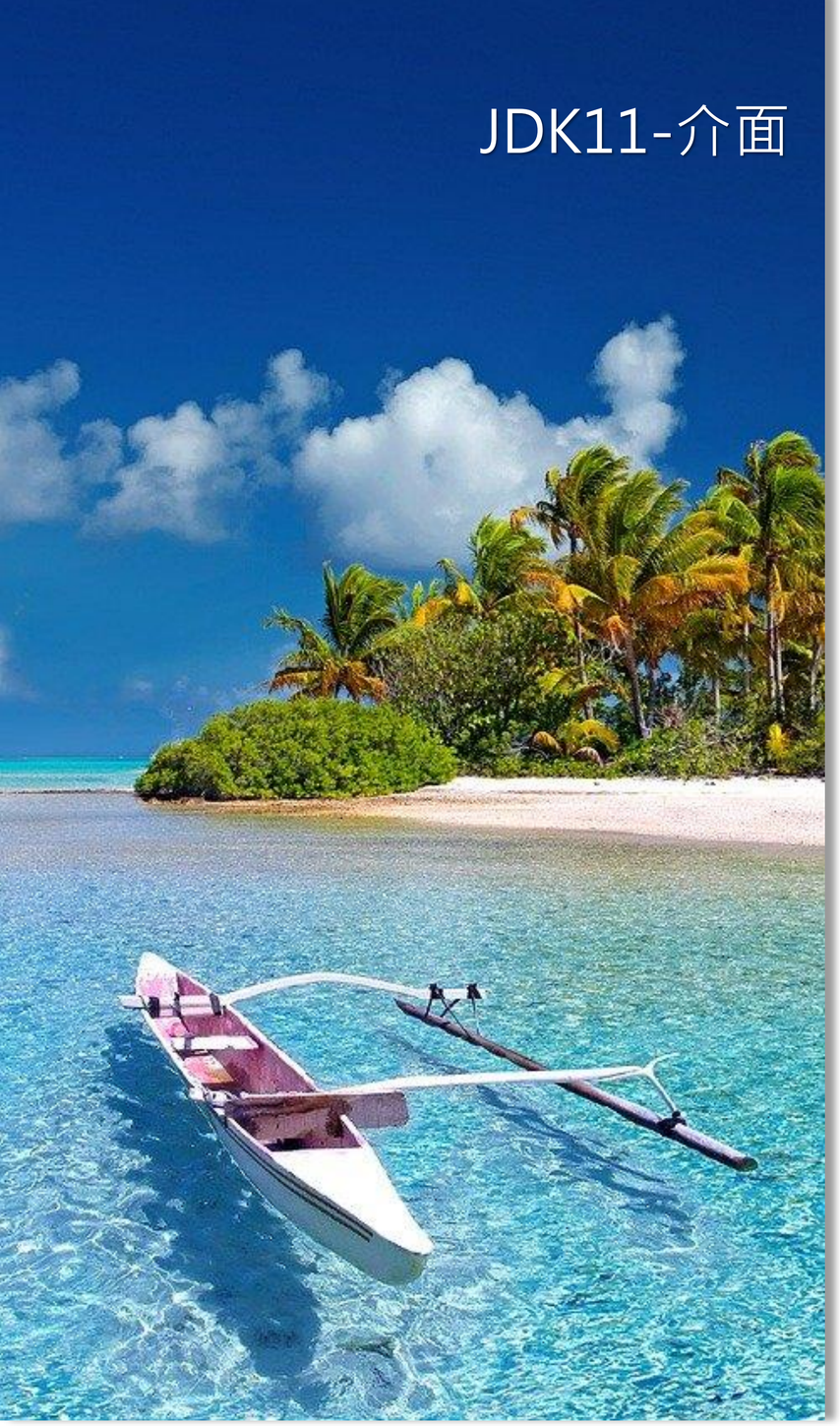


介面-EX2

```
public class NewClass1 {
    public static void main(String[] args) {
        new Ball("hello").play();
    }
}
interface Baseketball{
    void play();
}
interface Football{
    void play();
}
interface Game extends Baseketball,Football{
    Ball ball=new Ball("Go");
}
```

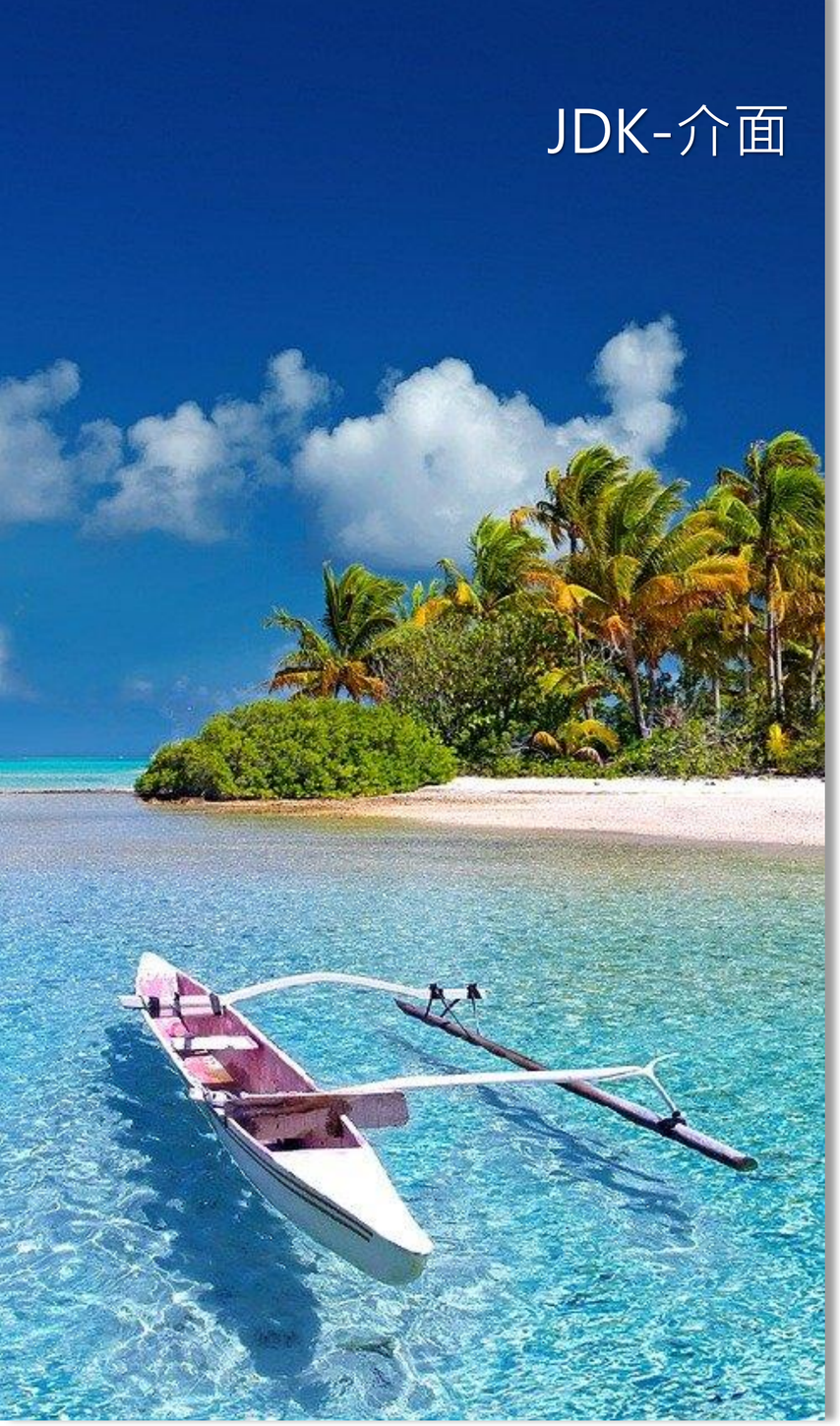
```
class Ball implements Game{
    private String name;
    public String getName(){
        return name;
    }
    public Ball(String name){
        this.name=name;
    }
    @Override
     void play(){
        ball=new Ball("Football");
        System.out.println(ball.getName());
    }
}
```

JDK11-介面



- JDK 11的介面
  - 除了定義常數及抽象方法外,還可以定義靜態方法和預設方法和私有方法
  - 介面中定義的靜態方法只能透過介面來存取，無法使用實作的子類呼叫
  - 介面中定義的預設方法，可以使用實作的子類呼叫，也可利用子類來覆寫
  - //靜態方法
  - public static void method1(){
  - System.out.println("IfaceA的靜態方法1");
  - }
  - //預設方法
  - public default void method2(){
  - System.out.println("IfaceA的預設方法2");
  - }
  - default void method3(){
  - System.out.println("IfaceA的預設方法3");
  - }
  - //私有方法
  - Private void method3(){
  - }

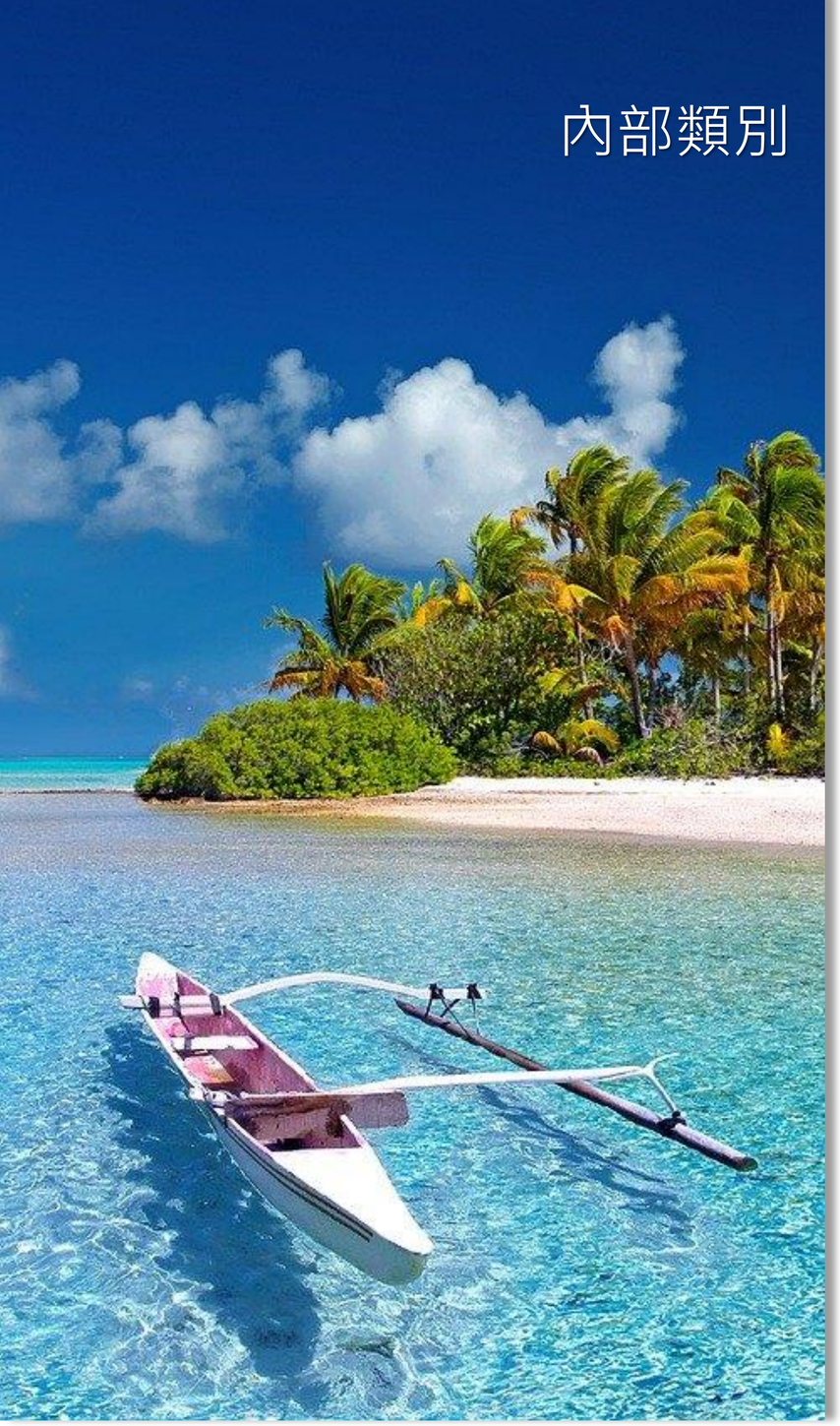
JDK-介面



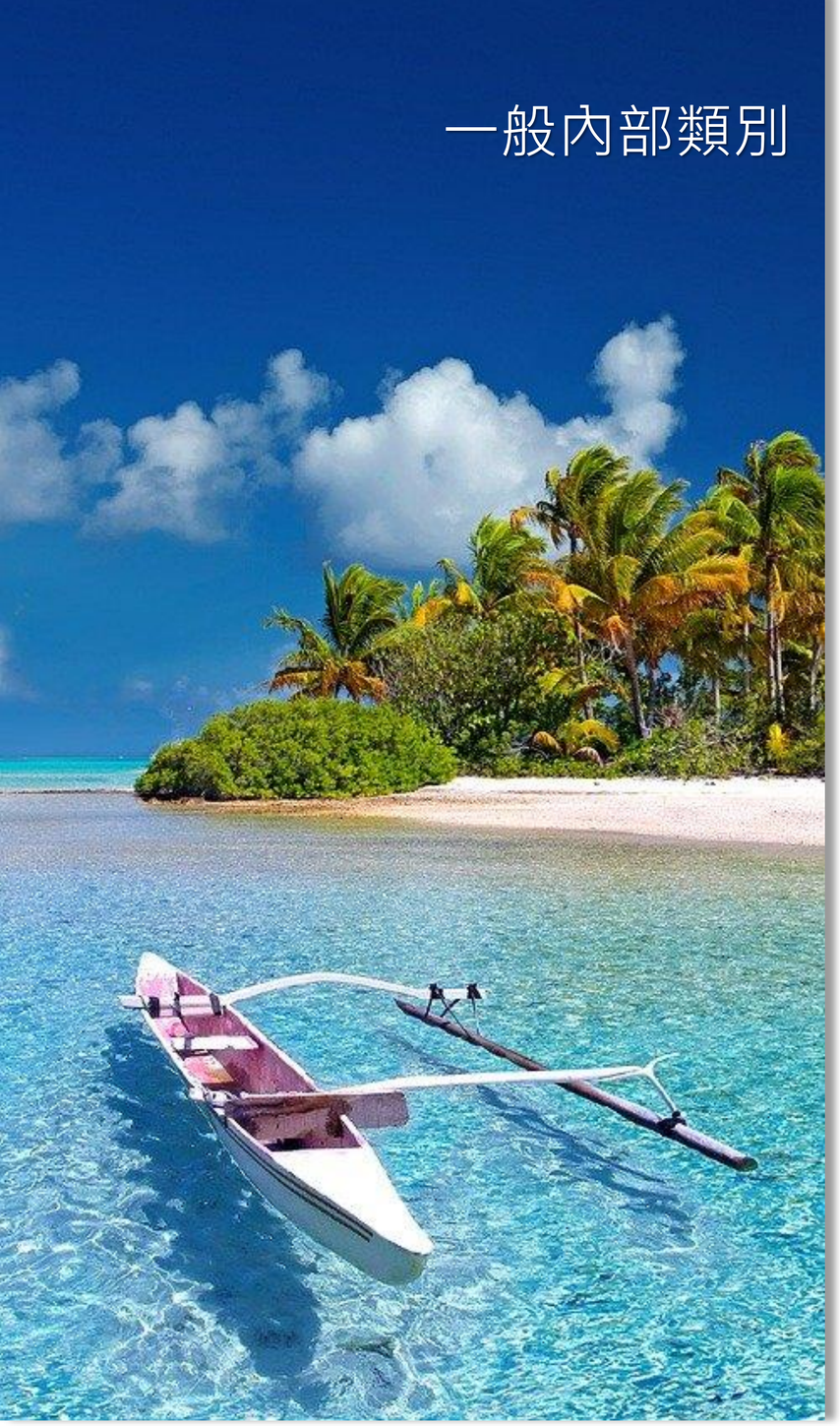
- 定義一個介面IfaceA宣告三個方法,一個靜態方法1,二個預設方法(no public)2,3
- 定義實作IfaceA的SubClass類別,覆寫一個預設方法2
- 產生SubClass類別物件,並呼叫方法2,3且試著呼叫介面的靜態方法
- 定義一個SuperClass類別,宣告一個方法method3,存取權限需為public
- 將SubClass繼承SuperClass,檢視SubClass呼叫的method3是介面的還是類別的
- 再宣告一個介面IfaceB並宣告一個預設方法3,並讓SubClass也實作介面IfaceB(二個實作的介面不能有同名方法,會衝突,只能利用子類別覆寫)
- 在SubClass中宣告一個方法method4
  - 呼叫自己的method3
  - 呼叫父類別的method3
  - 呼叫介面中的method3

內部類別 INNER CLASS

內部類別



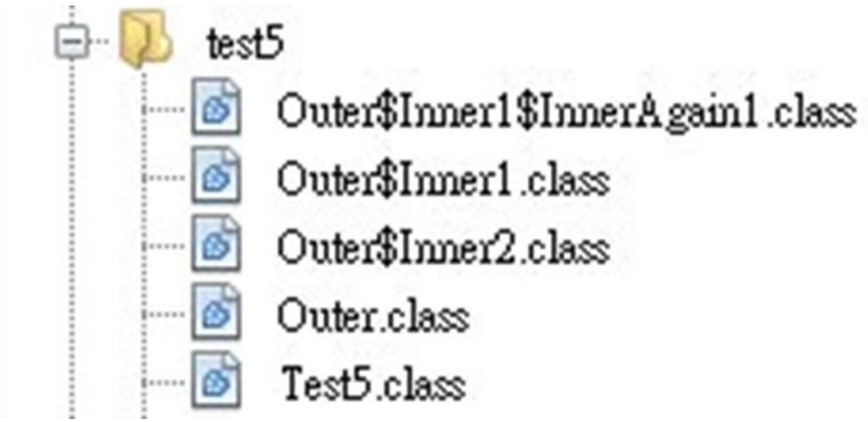
- 所謂的內部類別就是在類別中再定義一個類別，也可說是類別中的類別，其宣告位置及存取特性可分為以下四種
  - 成員內部類
    - 一般內部類別(regular inner class)
    - 靜態內部類別(static inner class)
  - 方法類別(method class) 修飾語只能為 default abstract final
    - 又稱為區域內部類別,可分別在
      - 方法內宣告
      - 建構子內宣告
      - 初始化區塊內宣告
  - 匿名類別(anonymouse class)



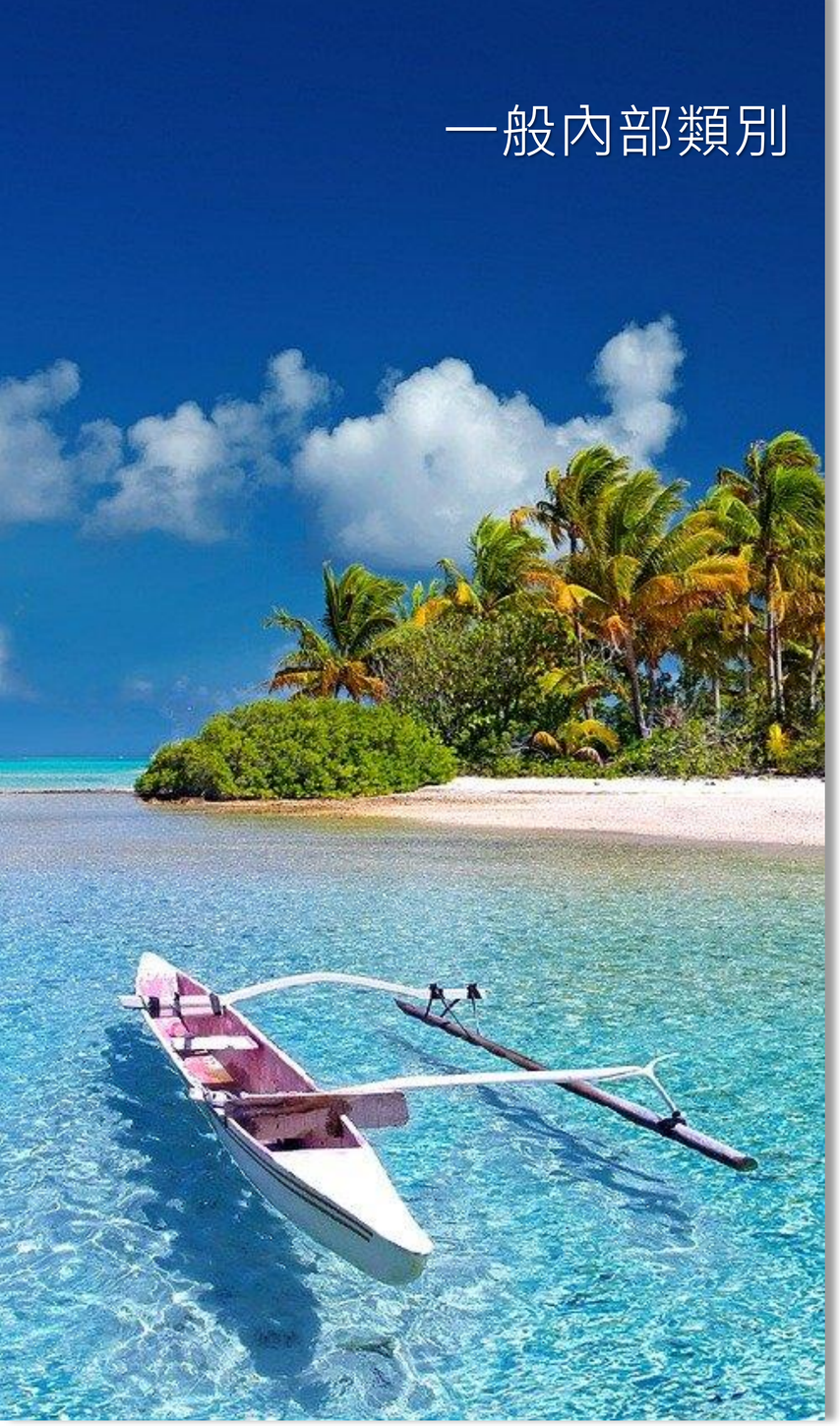
一般內部類別

- 一個外部類別可以有很多個內部類別，一個內部類別也可以有很多個次內部類別

```
public class Test5 {
    public static void main(String[] args) {
    }
}
class Outer{
    class Inner1{
        class InnerAgain1{
        }
    }
    class Inner2{
    }
}
```

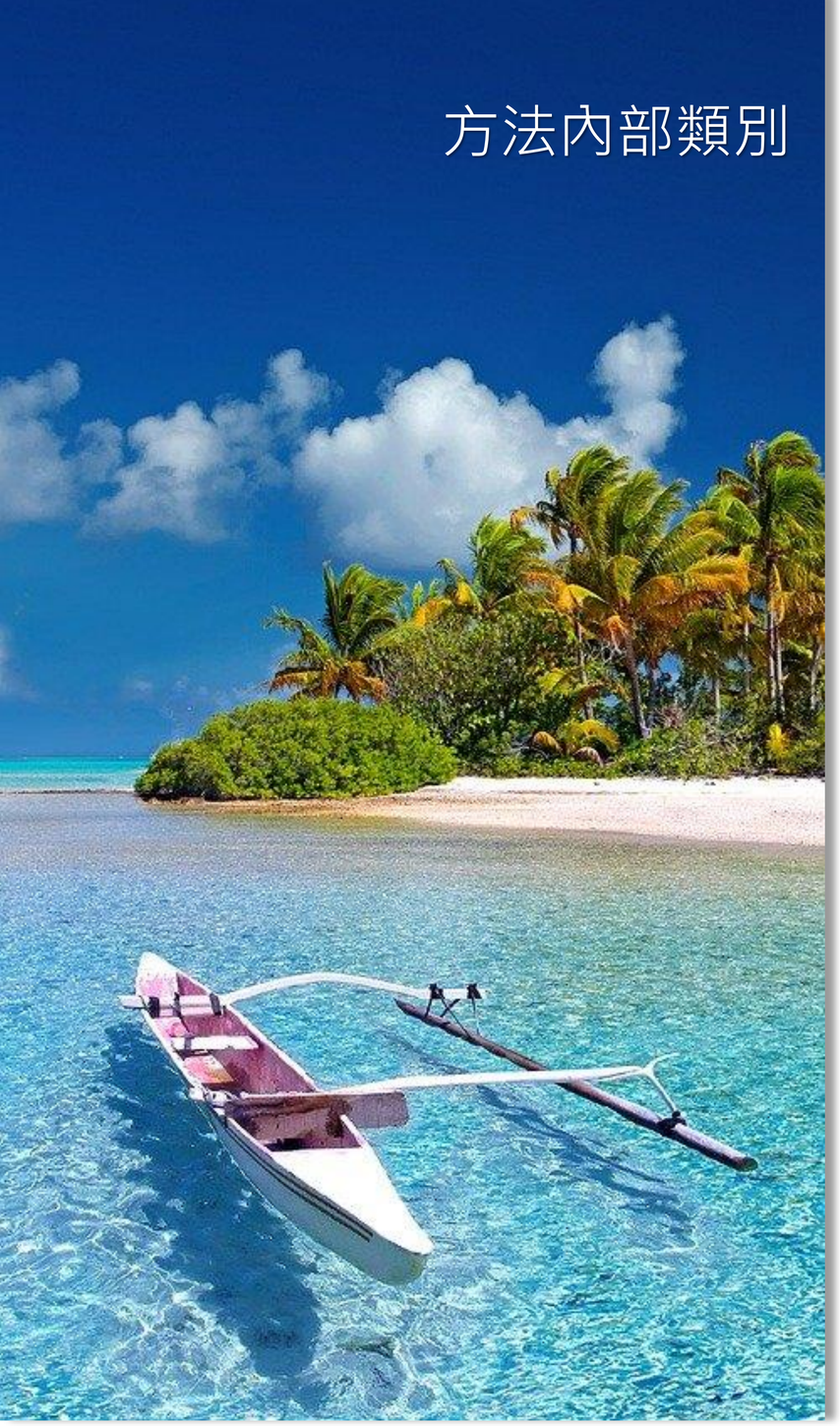


一般內部類別



```
public class PersonTest {
    public static void main(String[] args) {
        new Person().show();
        Person.Body body=new Person.Body();
        body.bodyShow();
         //在外部類別的方法中無法直接呼叫另一個類的非靜態內部類別
         //需先產生外部類別物件後,才可在產生內部類的實體
        Person person=new Person();
        Person.Brain pBrain=person.new Brain();

    }
}
class Person{
    void show(){
       //在外部類別的物件方法內可直接呼叫內部類
        Brain brain=new Brain();
        brain.brainShow();
        Body body=new Body();
        body.bodyShow();

    }
    static class Body{
        static int x=20;//靜態內部類別則可宣告類別變數
        final static int y=20;
        void bodyShow(){
            System.out.println("我是靜態的身體");
        }
    }
    class Brain{
        int x=20;
  void brainShow(){
            System.out.println("我是非靜態的大腦");
        }
    }
}
```

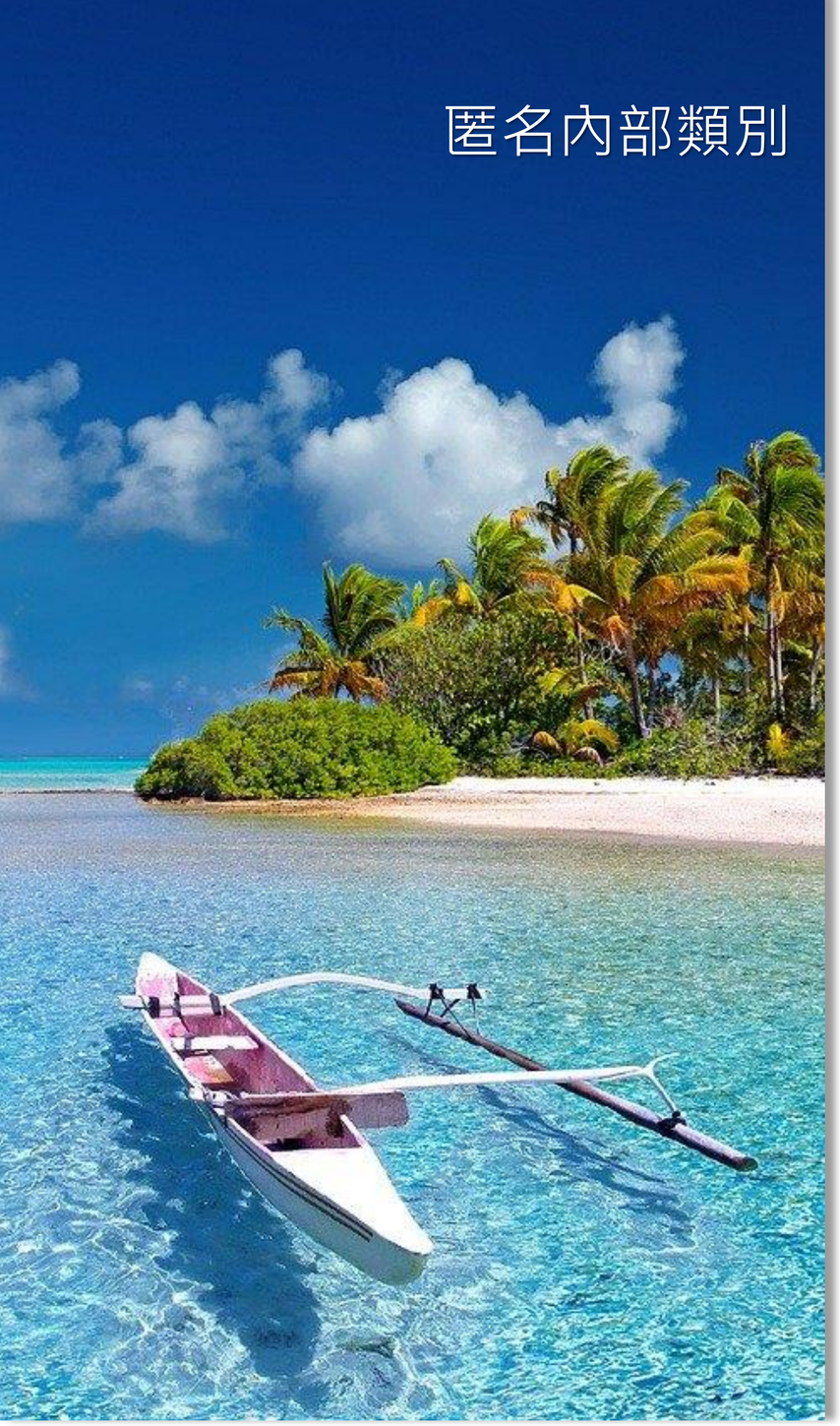
方法內部類別



- 方法內部類別顧名思義就是在方法中宣告的類別。
- 方法內部類別命名方式 Outer$1Inner1.class(1的流水序號是固定的,除非有名稱相同的的方法內部類別才會變2以做區別)

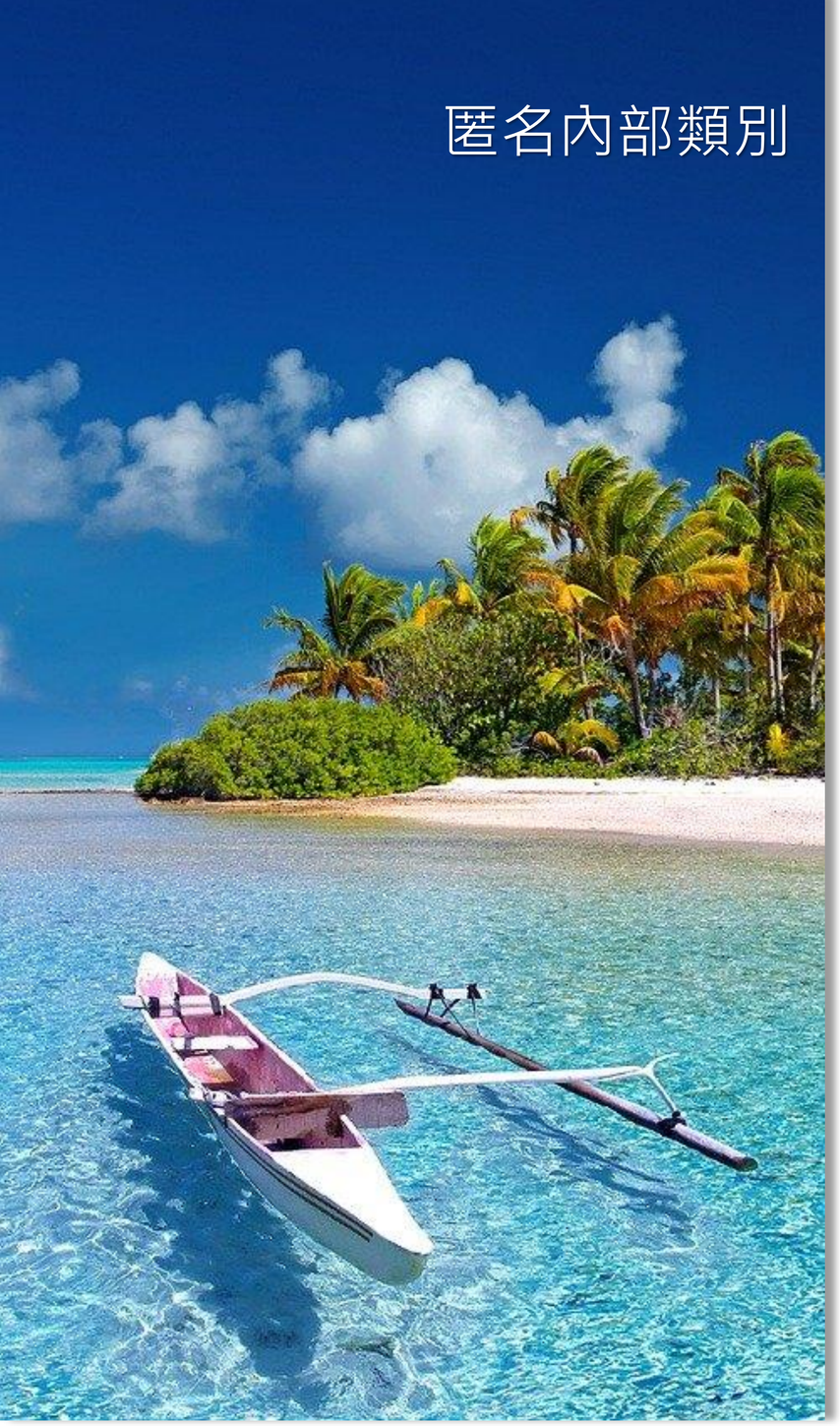
```
public class MethodClass {
    public static void main(String[] args) {
        new MethodClass().see();
    }
    void see(){
int num=50;//jdk7此行會報錯,jdk8會自動加上final
class InnerMethodClass{
            void foo(){
num=100;//此行編譯錯誤--2
System.out.println(num);//可直接調用--1
                System.out.println("Hello InnerMethodClass\n");
            }
        }
        InnerMethodClass imc=new InnerMethodClass();//要在方法類別定義後才可宣告
        imc.foo();
    }
}
```

匿名內部類別



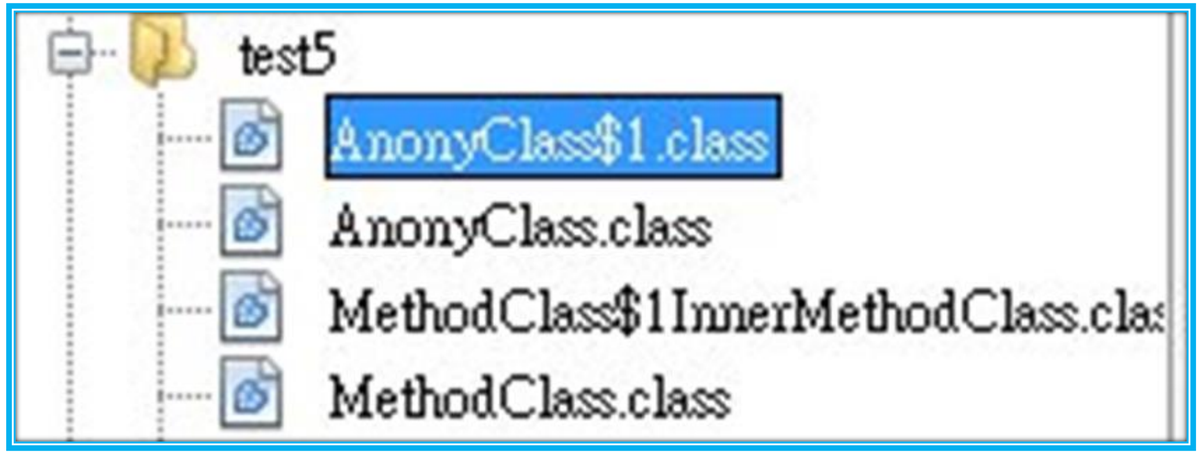
- 匿名類別簡單的說就是沒有宣告名稱的類別，直接以{....}來實作的類別程式碼。匿名類別是四大內部類別中最容易，也最常被使用的內部類別寫法，適用於只在程式中用到一次，之後更不需再使用的類別

匿名內部類別

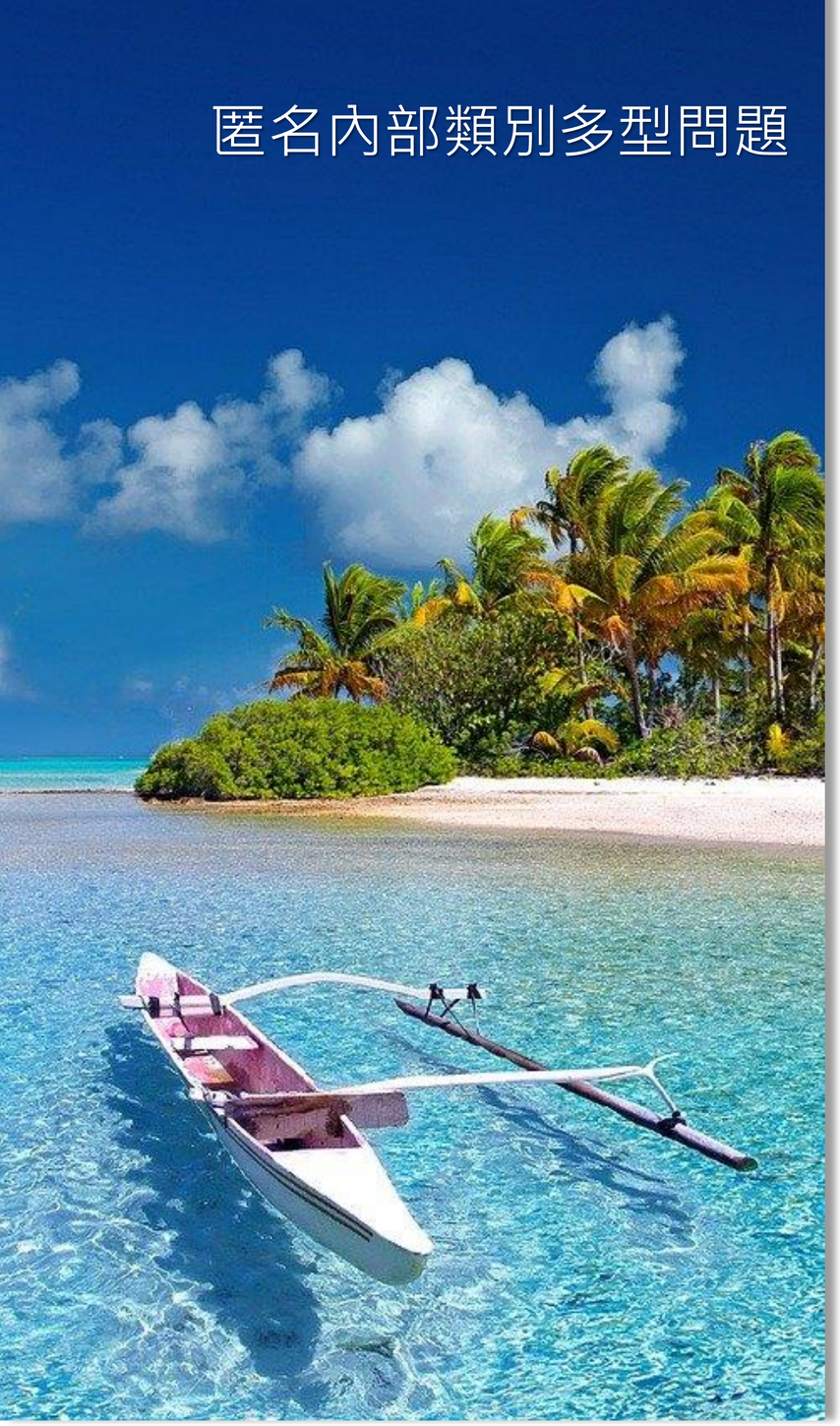


- 類別名稱以流水序號1~n依序表示下去

```
public class AnonyClass {
    public static void main(String[] args) {
        Pet p=new Pet(){//原先不能直接實體化介面,但透過匿名類別則可以
            public void skill(){
                System.out.println("我會唱歌");
            }
            public void move(){
                System.out.println("我會跑步");
            }
        };//這裡記得加上分號
        p.skill();
        p.move();
    }
}
interface Pet{
    String attr="cute";
    void skill();
    void move();
}
```

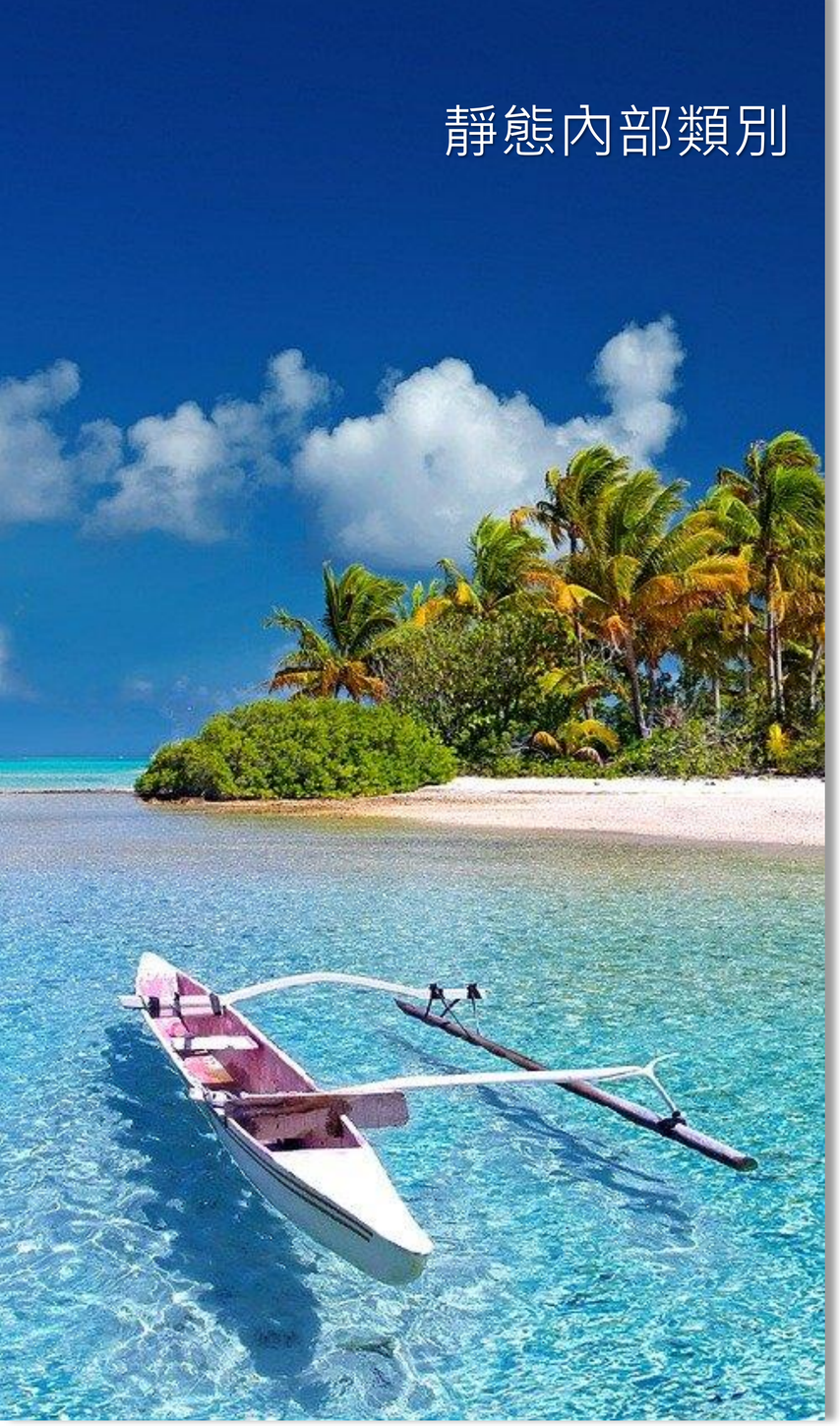


匿名內部類別多型問題



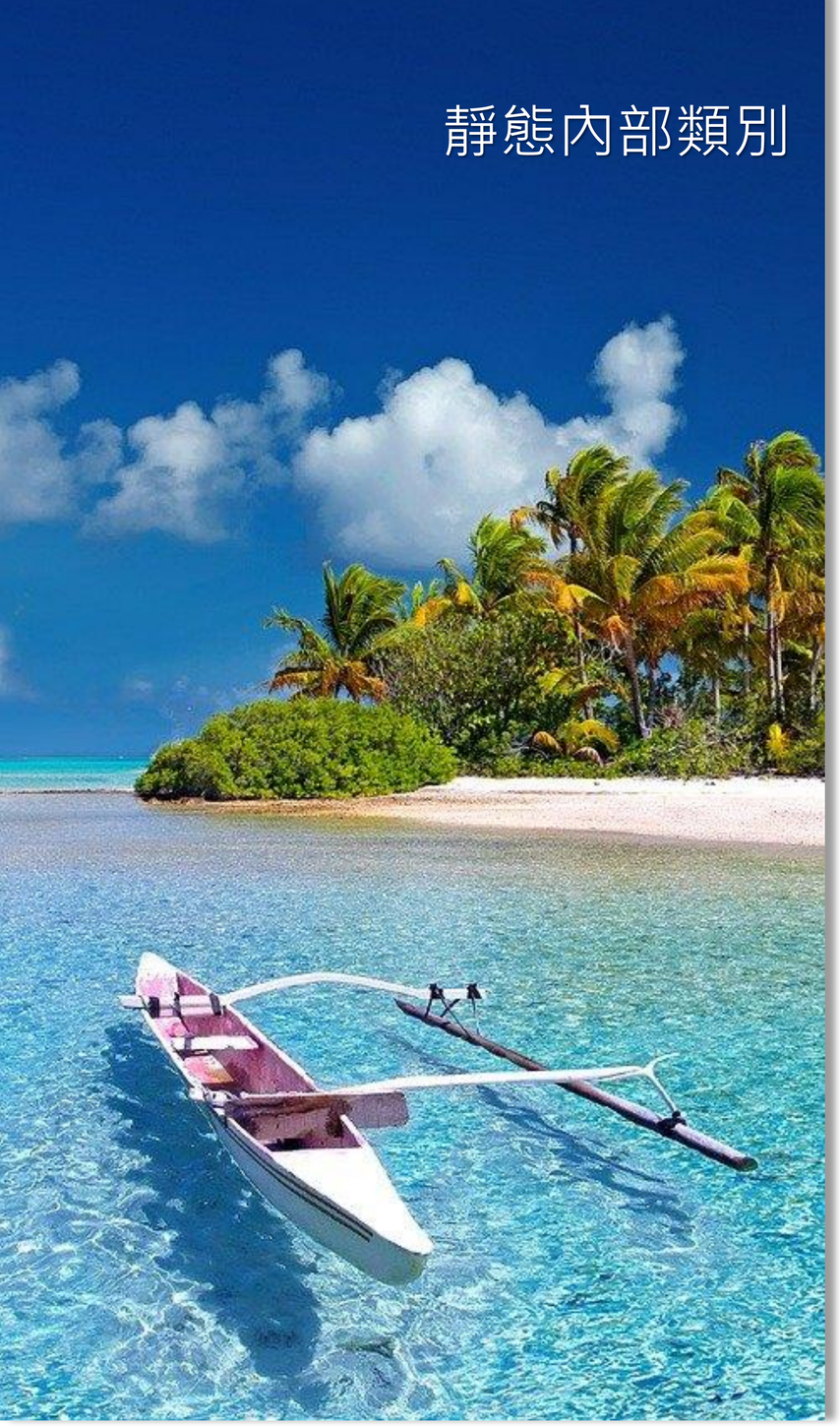
- 類別名稱以流水序號1~n依序表示下去
- public class AnonyClass {
- public static void main(String[] args) {
- Pet p=new Pet(){//原先不能直接實體化介面,但透過匿名類別則可以
- public void skill(){
- System.out.println("我會唱歌");
- }
- public void move(){
- System.out.println("我會跑步");
- }
- public void bite(){
- System.out.println(“我會咬人");
- }
- };//這裡記得加上分號
- p.skill();
- p.move();
- p.bite();//此行錯誤,父類別無法看到子類別,多型概念
- }
- }
- interface Pet{
- String attr="cute";
- void skill();
- void move();
- }

靜態內部類別



- 靜態內部類別是在宣告內部類別時加上static修飾字
- 使其物件實體配置在global區塊中,而不是heap區塊中
- 靜態內部類別沒有this參考指標指向外部類別
- 靜態內部類別可以同時定義static 與not-static成員
- 內部類別除了static內部類別可定義static 成員

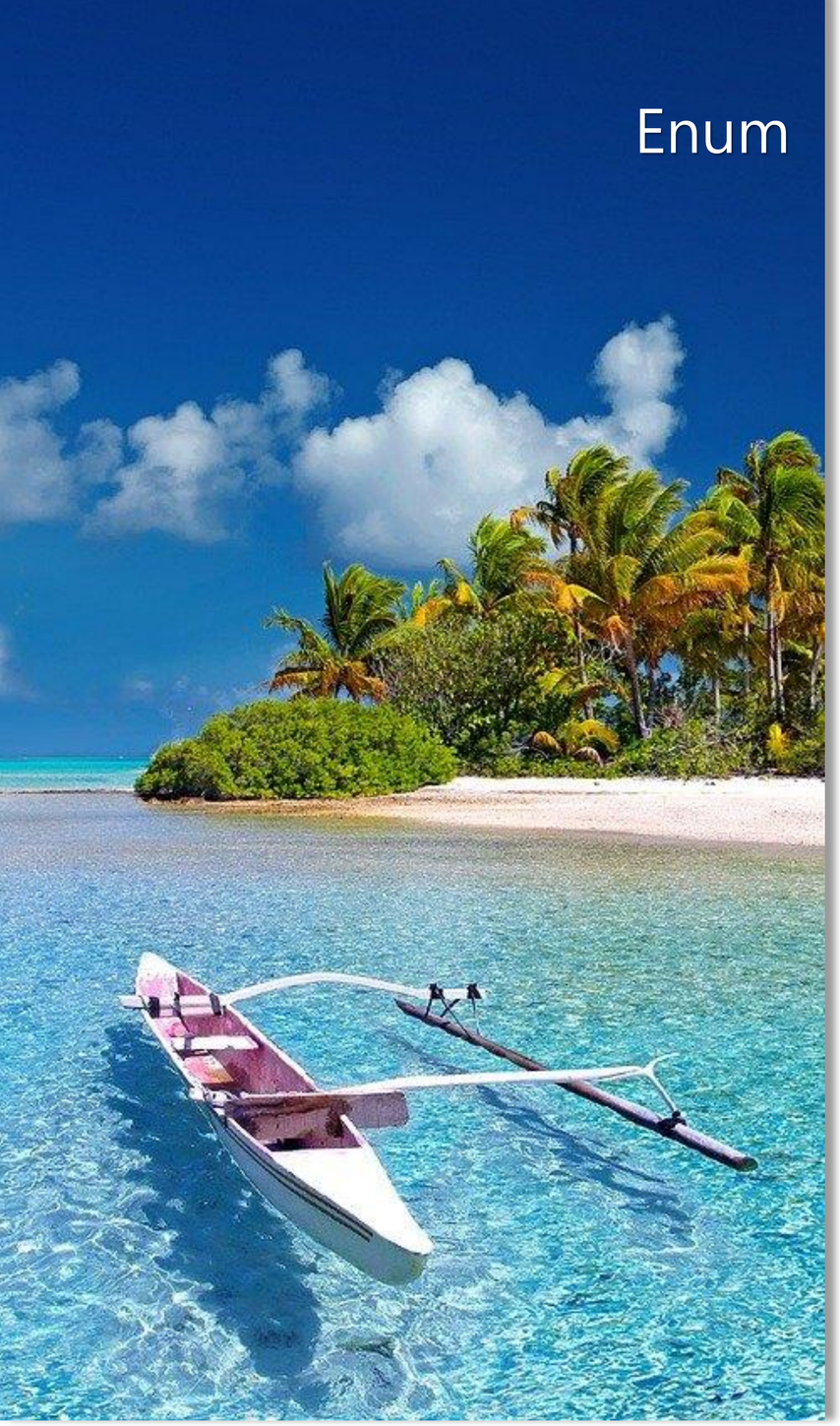
靜態內部類別



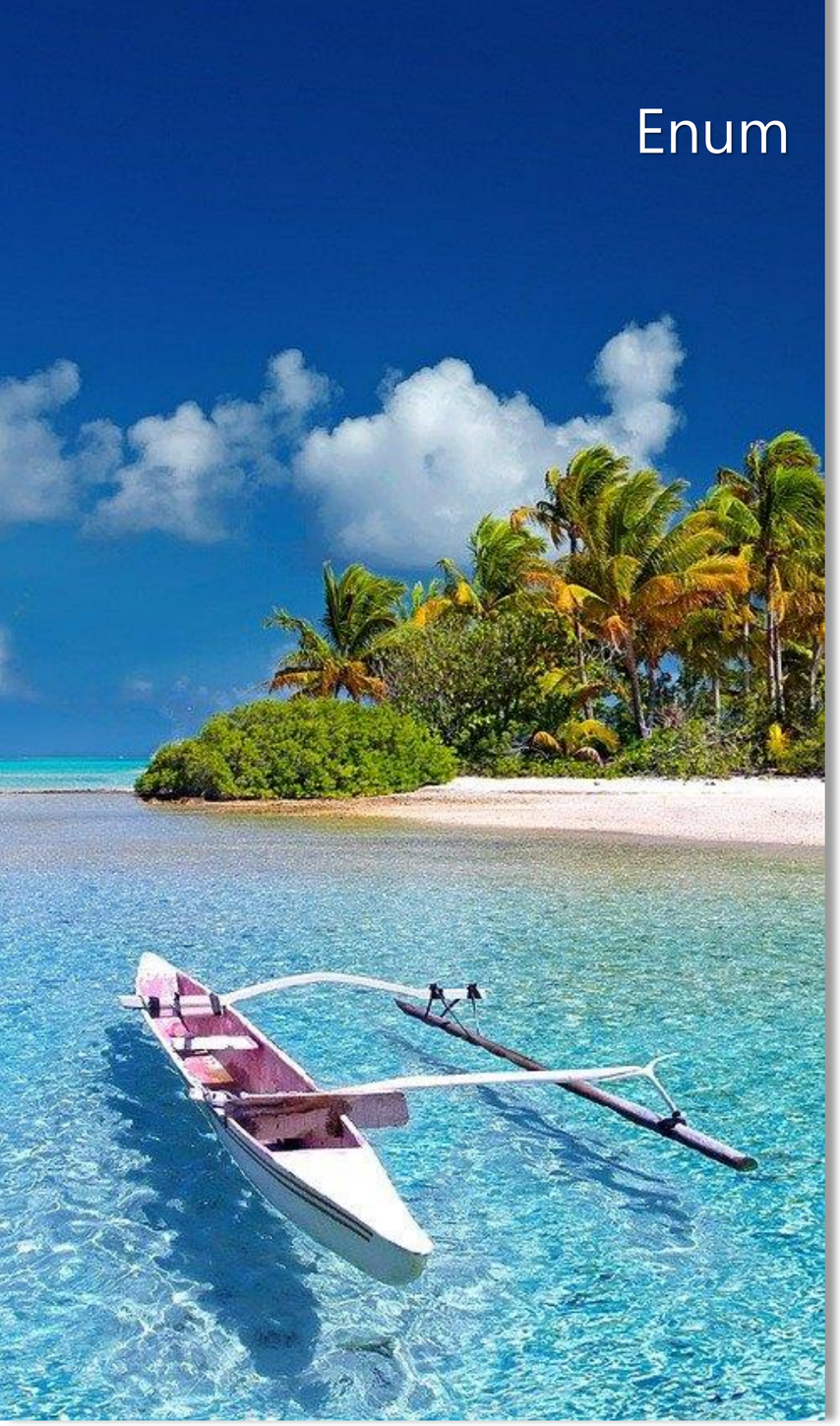
```
public static void main(String[] args) {
MyOuter1.MyInnerStatic ms=new
MyOuter1.MyInnerStatic();
        ms.fooA();
        ms.fooB();    }}
class MyOuter1{
    private int x=7;
    private static int sx=9;
    static class MyInnerStatic{
        private int x=77;
        private static int sx=99;
        public void fooA(){
            System.out.println(sx);
            System.out.println(MyOuter1.sx);
            System.out.println(x);
            System.out.println(this.x);
            System.out.println(new
MyOuter1().x);
        }
        public static void fooB(){
            System.out.println(sx);
            System.out.println(MyOuter1.sx);
        }    } }
```

Enum 列舉

Enum



- 以 enum 關鍵字來宣告 例　enum Season{ SPRING,SUMMER,AUTUMN,WINTER;}
- 一般會將有固定數目的值宣告為 enum ,若值不固定則不建議
- 所產生列舉類別具有 final 屬性 並繼承 Enum類別
- 列舉的建構子是由編譯器自動產生,且存取權限為 private 或 default
- 列舉的內容值如 SUNDAY,MONDAY.. 皆經過 public static final 修飾

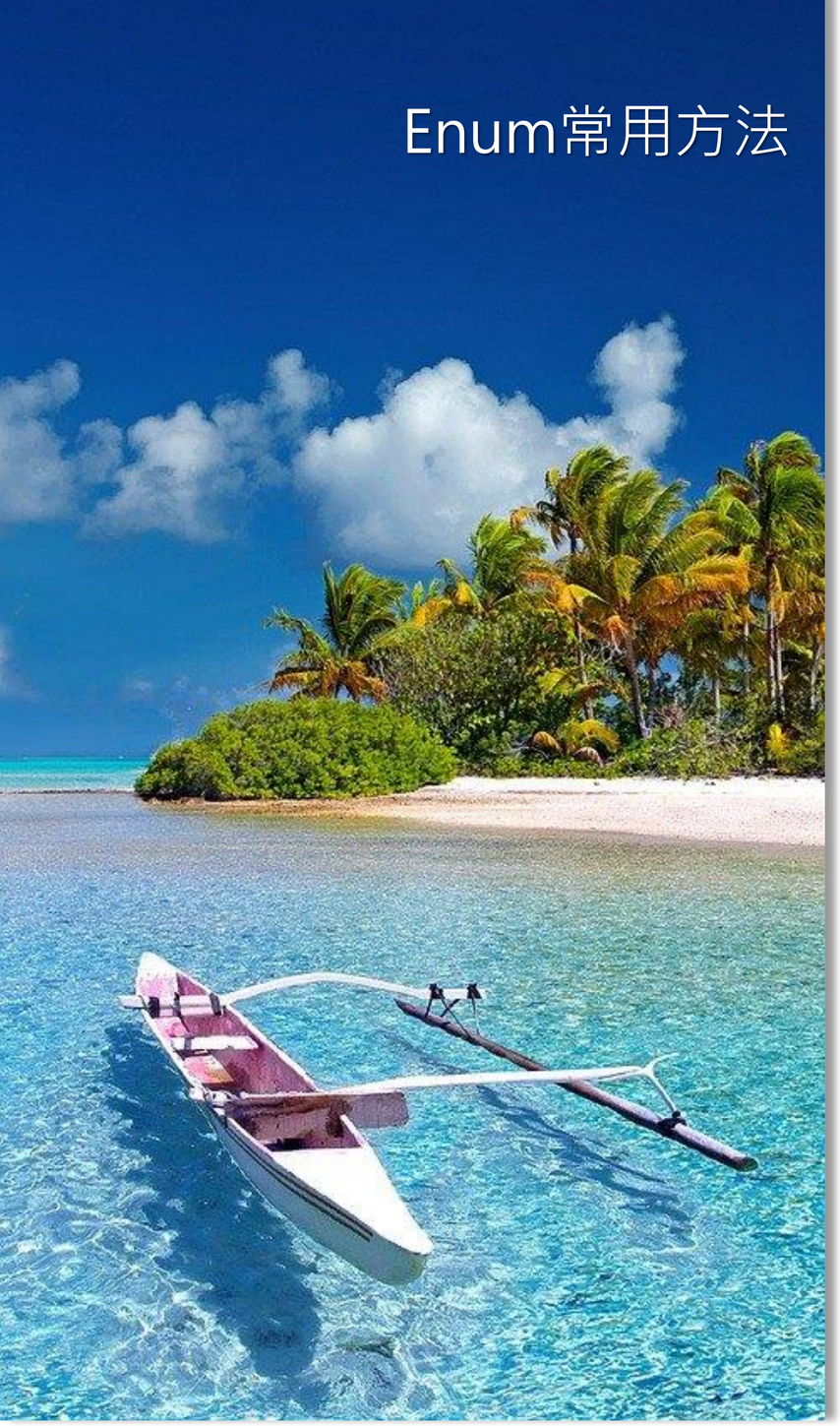


Enum

- 用來宣告已知數量之物件
- 使用enum關鍵字來宣告
- 語法
- enum 變數{
- 常數1,常數2.....常數n;
- }
- 自訂列舉類別

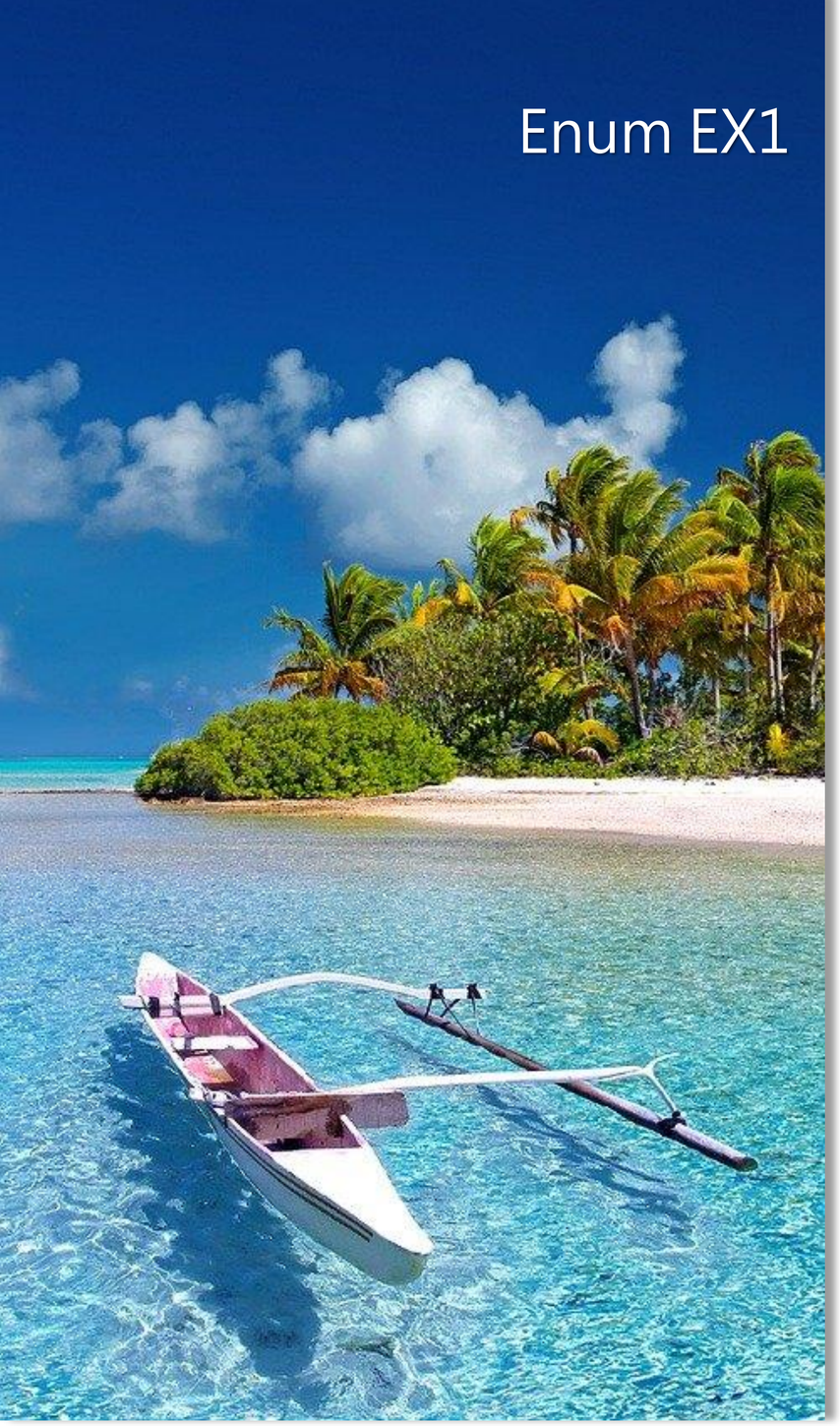
```
class JavaProgram {
	// 1.定義物件屬性
	private final String typeName;
	private final String typeDescription;

	// 2.私有化建構子,即不可再外面再新增實體,如此才有辦法確認有限個,因為列舉是固定的已知項目,如星期,四季
	// 利用建構子來指定宣告為final屬性的值
	private JavaProgram(String typeName, String typeDescription) {
		this.typeName = typeName;
		this.typeDescription = typeDescription;
	}

	// 3.提共當前列舉類的多個對象
	public static final JavaProgram JAVASE = new JavaProgram("標準版", "用於單機開發");
	public static final JavaProgram JAVAEE = new JavaProgram("企業版", "用於伺服器開發");
	public static final JavaProgram JAVAME = new JavaProgram("微型版", "用於行動裝置開發");

	// 4.取得列舉物件的屬性
	public String getTypeName() {
		return typeName;
	}

	public String getTypeDescription() {
		return typeDescription;
	}

	@Override
	public String toString() {
		return "JavaProgram [typeName=" + typeName + ", typeDescription=" + typeDescription + "]";
	}

}
```

Enum常用方法



- values() 回傳列舉元素
- toString()
- ordinal()返回列舉的索引值

Enum EX1

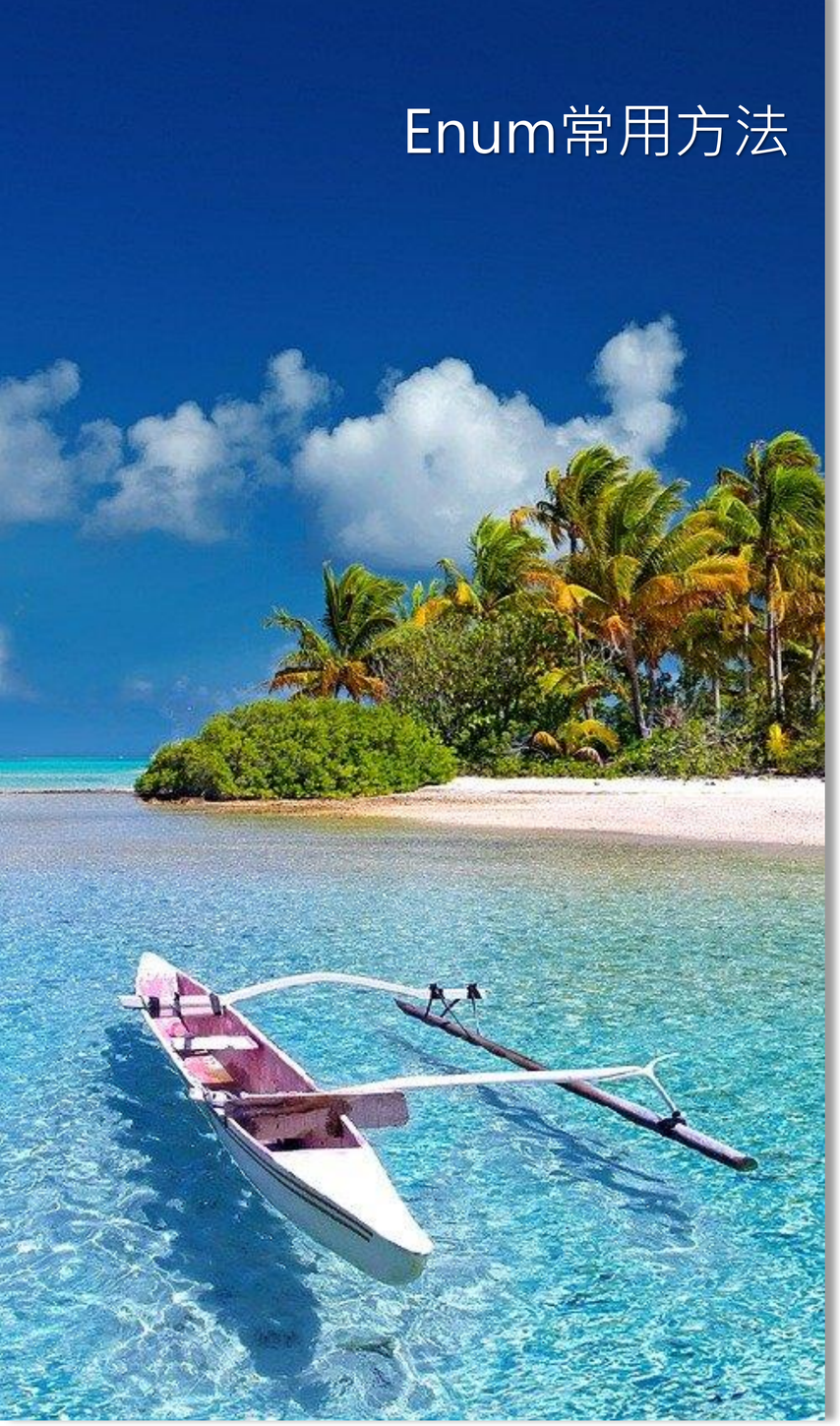


Which two modifications enable the given code to compile? *

```
USCurrency.java
public enum USCurrency {
        PENNY(1),
        NICKLE(5),
        DIME(10),
        QUARTER(25);
        private int value;

        public USCurrency(int value) {
                this.value = value;
        }
         public int getValue() {
                return value;
        }
}
Coin.java
public class Coin {
     public static void main(String[] args) {
                USCurrency usCoin = new USCurrency.DIME;
                System.out.println(usCoin.getValue());
        }
}
```

- [ ] A. Nest the USCurrency enumeration declaration within the Coin class.
- [ ] B. Make the USCurrency enumeration constructor private.
- [ ] C. Remove the new keyword from the instantion of usCoin.
- [ ] D. Make the getter method of value as a static method.
- [ ] E. Add the final keyword in the declaration of value.

Enum常用方法



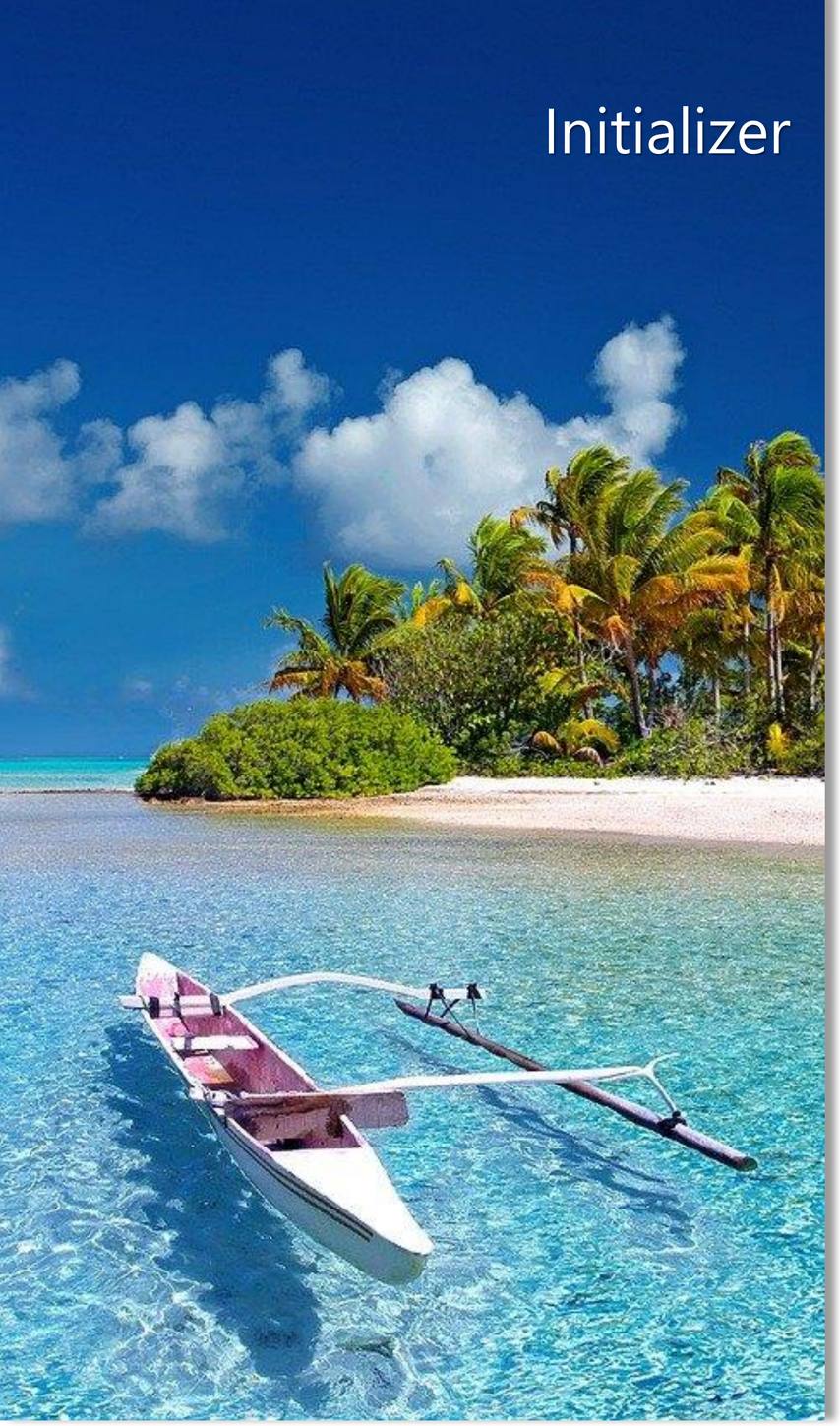
Examine this code: System.out.println(Letter.values()[1]);What code should be written at line 5 for this code to print 200?

```
public class Test10 {
        public static void main(String[] args) {
                System.out.println(Letter.values()[ 1]);
        }
}
enum Letter{
        ALPHA(100),BETA(200),GAMMA(300);
        int v;
        Letter(int v){this.v=v;}
        /*Insert code here */
}
```

- A. public String toString() { return String.valueOf(ALPHA.v); }
- B. public String toString() { return String.valueOf(Letter.values()[1]); }
- C. public String toString() { return String.valueOf(v); }
- D. String toString() { return “200”; }

Initializer 初始化區塊

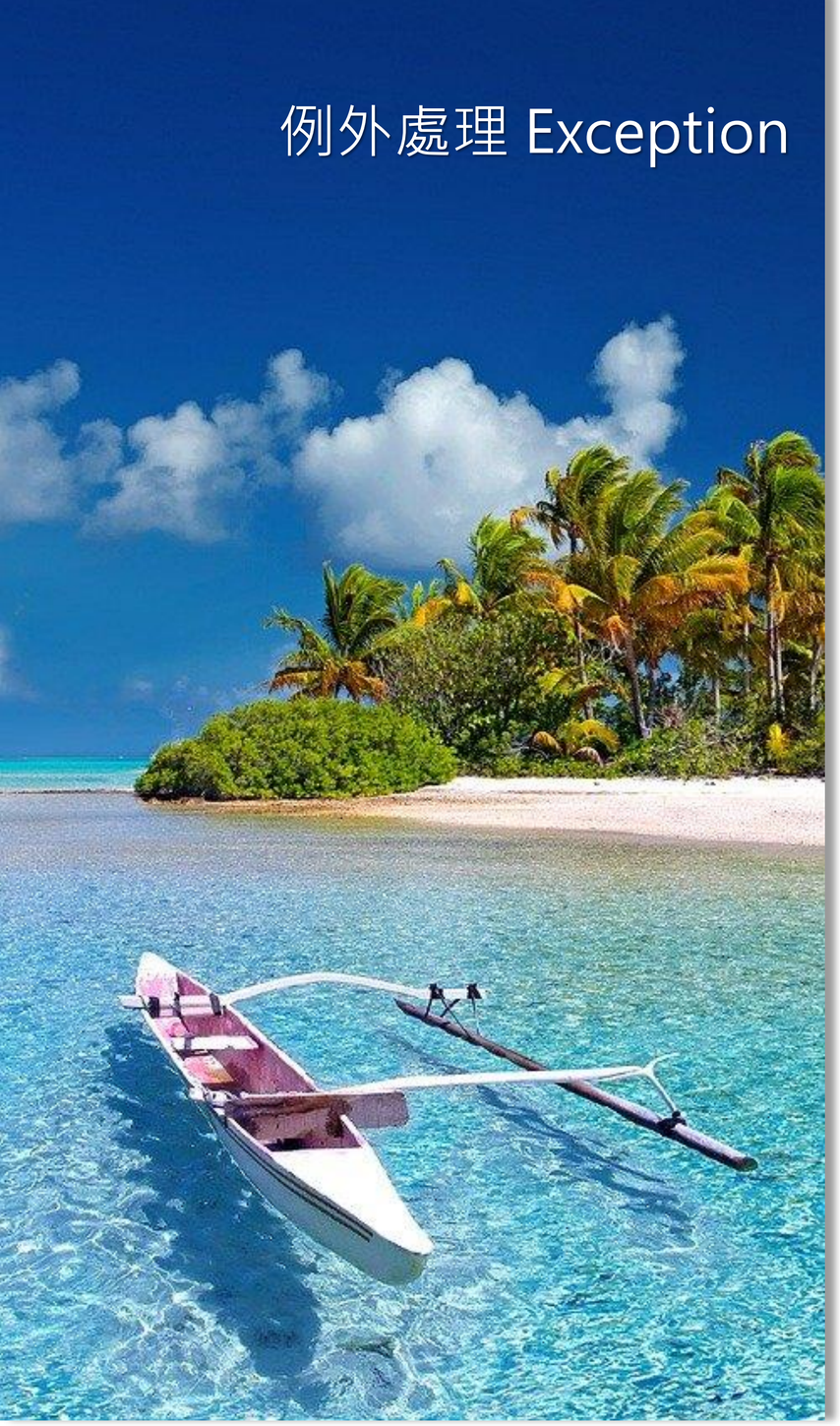
Initializer



- 初始化區塊分為
- **instance initializer** 物件初始化區塊
  - 可以初始化物件變數及初始化類別變數
  - 當上述物件初始化區塊與建構子初始化重覆時,會以建構子的初始化優先
- **static initializer** 類別初始化區塊
  - 只能初始化類別變數,無法初始化物件變數,且其不會將程式碼移至建構子內

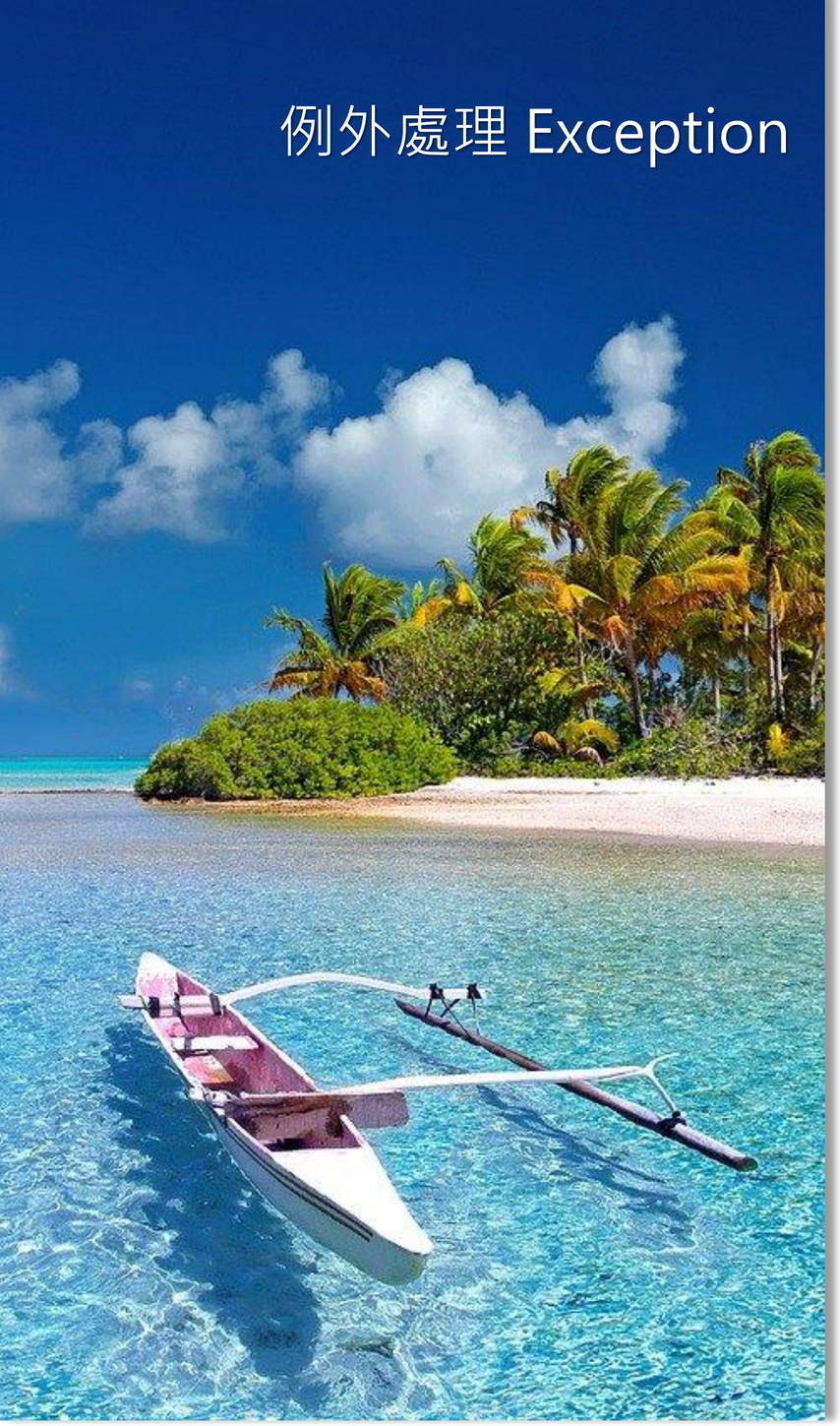
例外處理 EXCEPTION

例外處理 Exception



程式上的錯誤依性質可分為以下三種

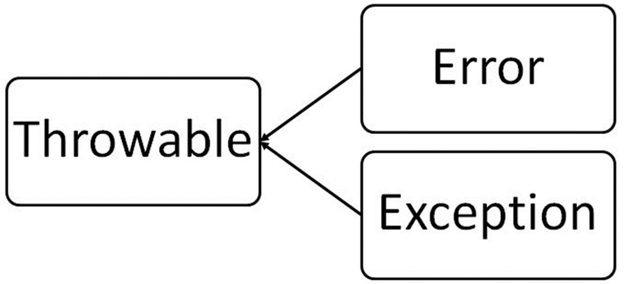
- 程式語法上的錯誤
  - 即為編譯時期的錯誤(Compile time error)程式人員可以很容易地找出錯誤位置，如char c="SCJP"
- 執行時期的錯誤
  - 執行時期錯誤，顧名思義就是在編譯時沒有問題，卻在執行的時候發生錯誤，最典型的就是陣列超出索引值，或是除以0
- 商業邏輯的錯誤
  - 此錯誤是最難找到的BUG，例如利息配發公式，折扣計算方式



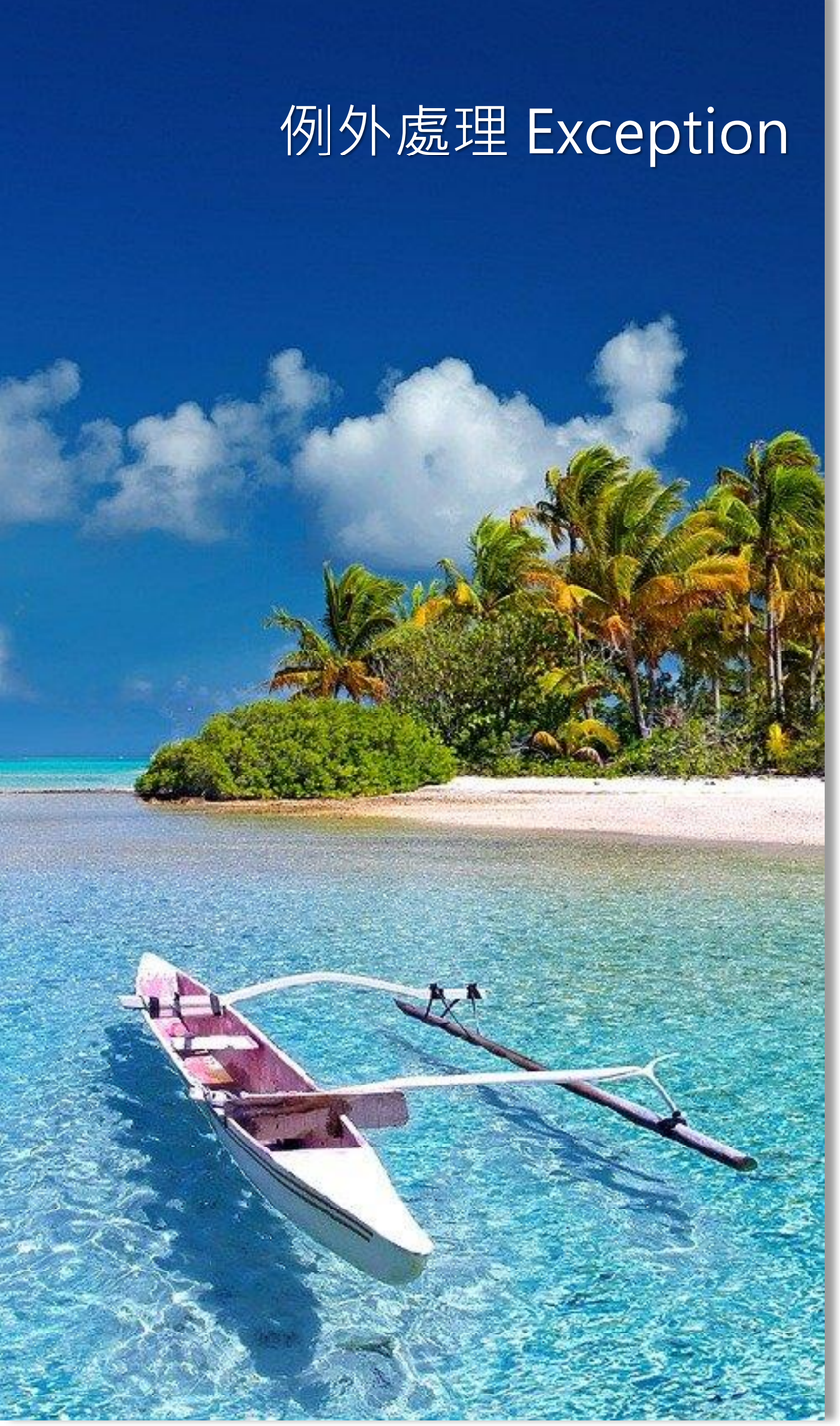
例外處理 Exception

在JAVA語言中,將程式執行中發生的不正常的情況稱為例外,可分為

- ERROR(UncheckedException編譯器不要求要檢查的例外)
  - JAVA 虛擬機器無法解決的嚴重問體,如系統內部錯誤,資源耗盡等問題,一般不編寫針對性的程式進行處理.
  - 例StackoverFlowError(main呼叫自己),OutOfMemory(new int[1024*1024*1024])
- Exception
  - RuntimeException(UncheckedException編譯器不要求要檢查的例外)
    - 執行時期錯誤，顧名思義就是在編譯時沒有問題，卻在執行的時候發生錯誤，最典型的就是陣列超出索引值，或是除以0
    - 例NullpointerException,ArrayIndexOutOfBoundsException,ArithmeticException
  - 除了RuntimeException外的其它子類皆為Checked Exception即編譯器要求一定要檢查例IOException FileNotFoundException……..



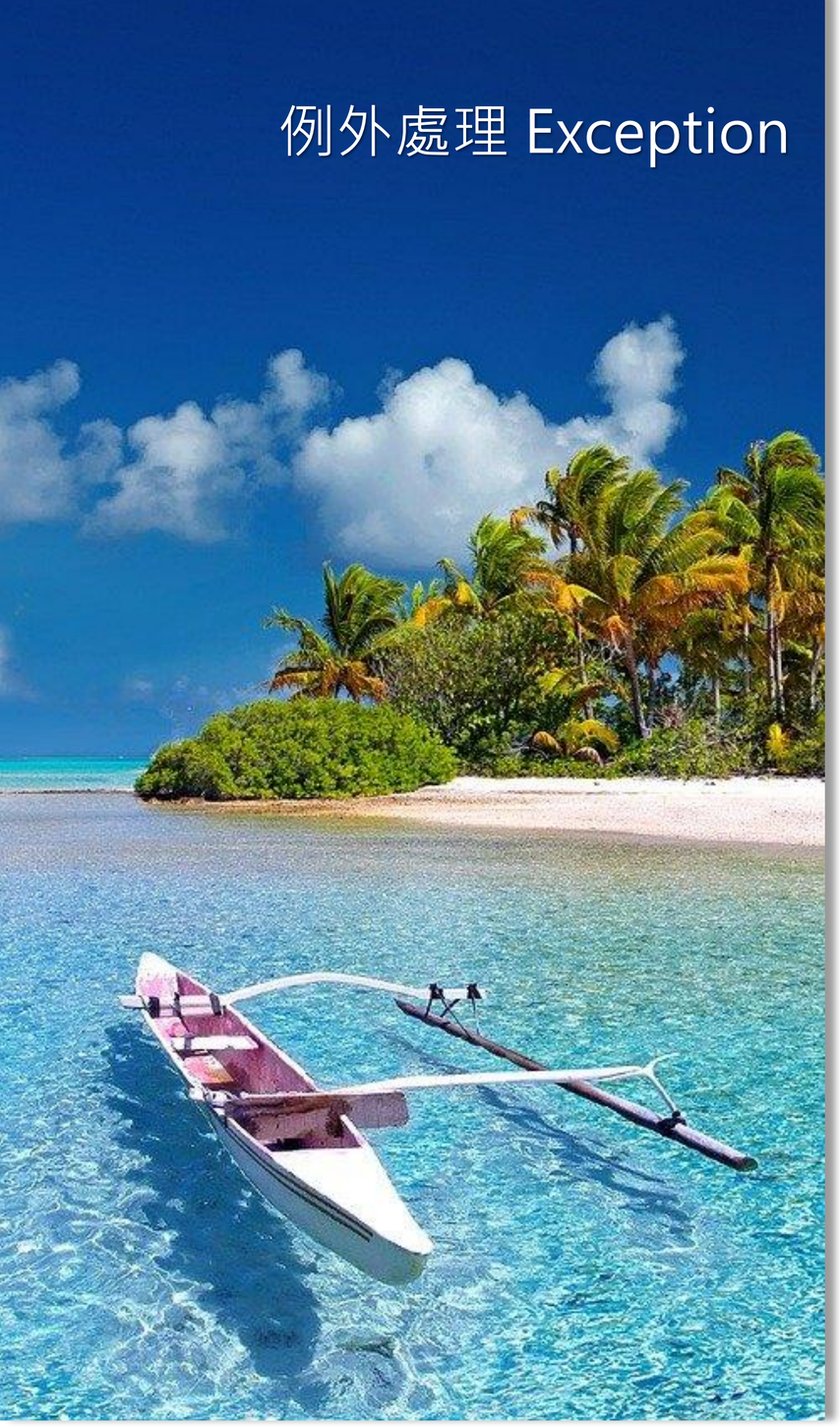
例外處理 Exception

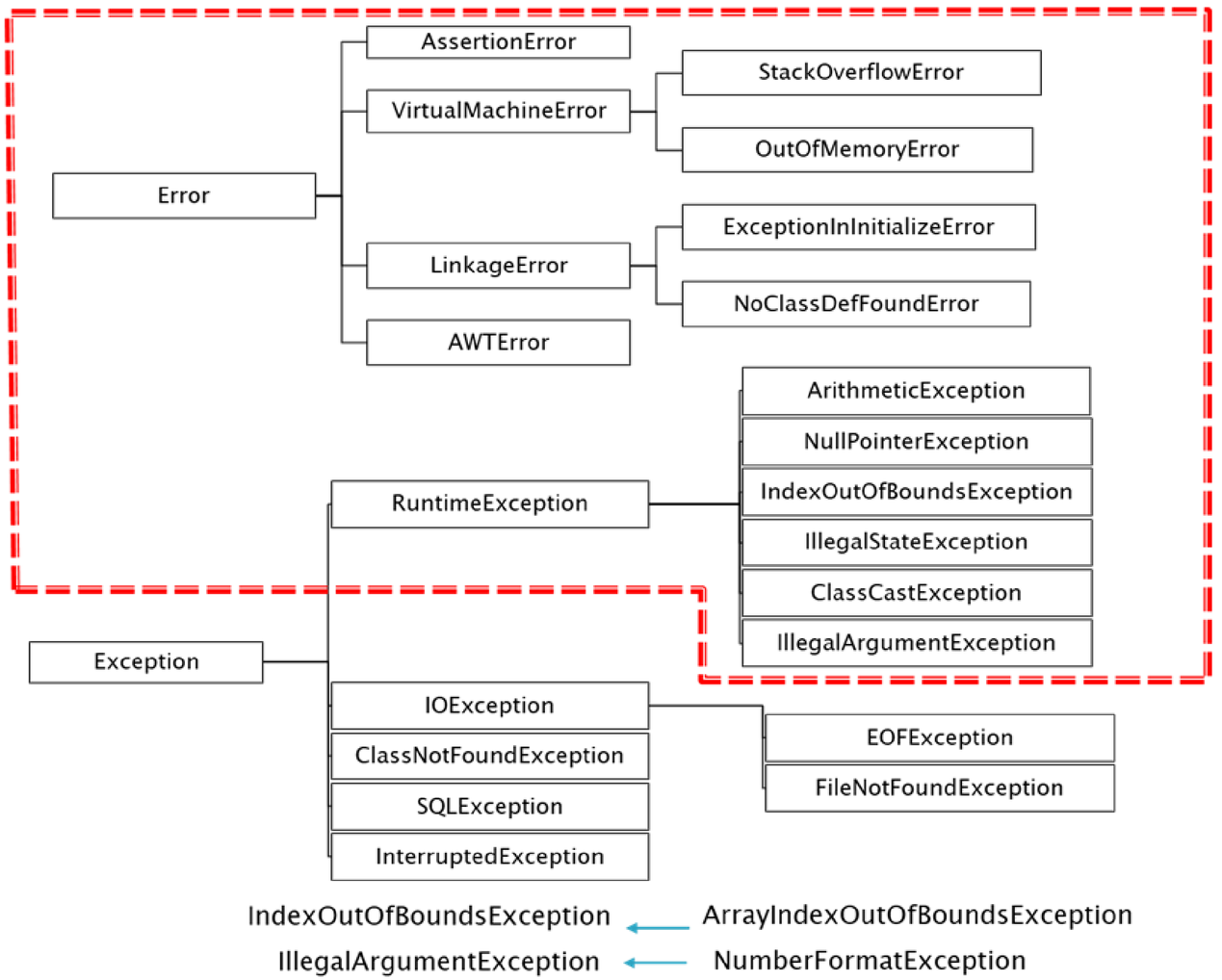


- 例外分類
  - 不需檢查的例外 (Unchecked Exception)
    - java.lang.Error
    - java.lang.RuntimeException
      - NullPointerException,ArithmeticException
  - 必需檢查的例外 (Checked Exception)
    - java.lang.Exception 除了RuntimeException
      - IOException

```
public static void main(String[] args)throws IOException{
    File file=new File("a- * bc.txt");
   try{
    file.createNewFile();
   }catch(IOException e){
       System.out.println("檔案無法存取");
   }
}
```

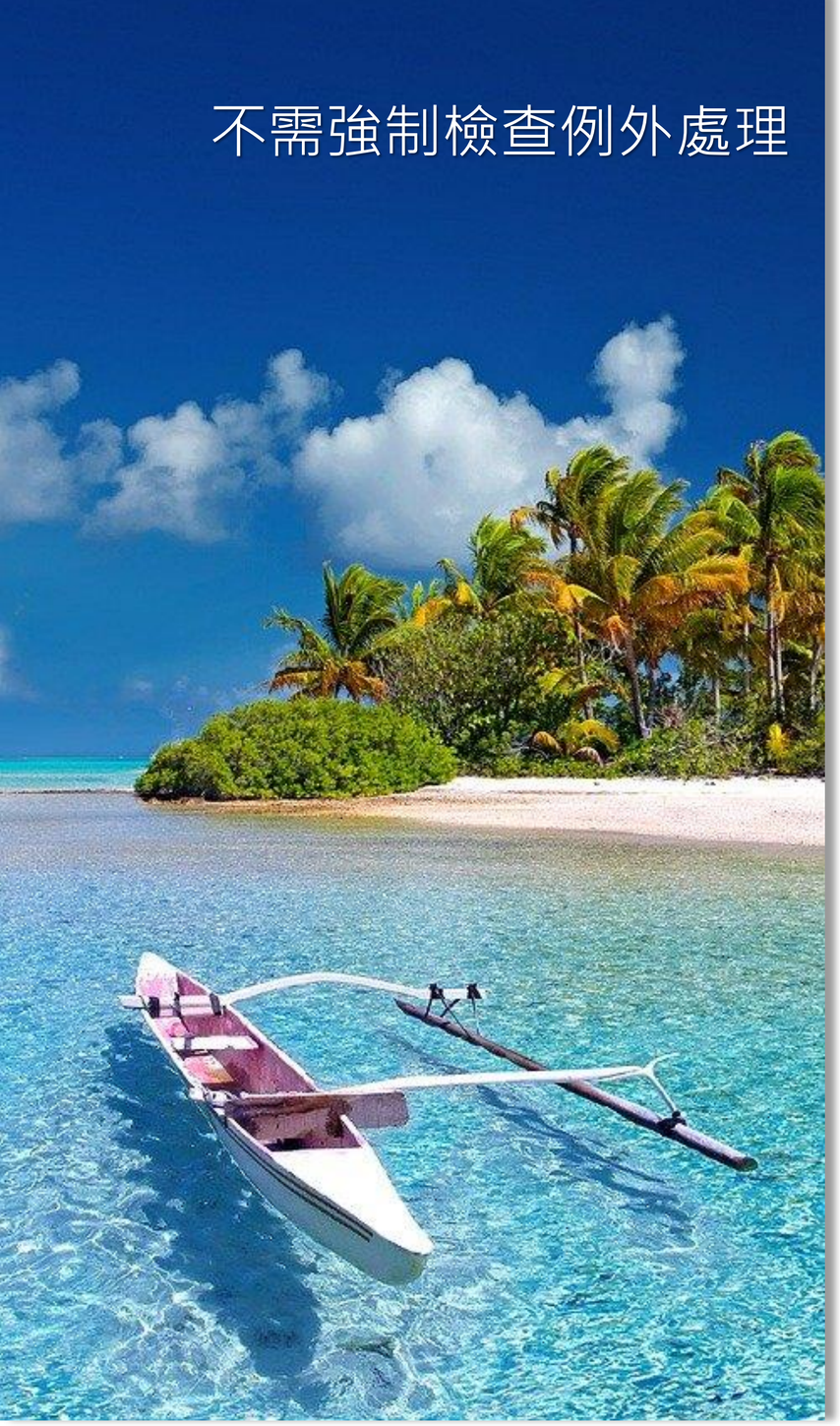
例外處理 Exception





IndexOutOfBoundsException ← ArrayIndexOutOfBoundsException

IllegalArgumentException ← NumberFormatException

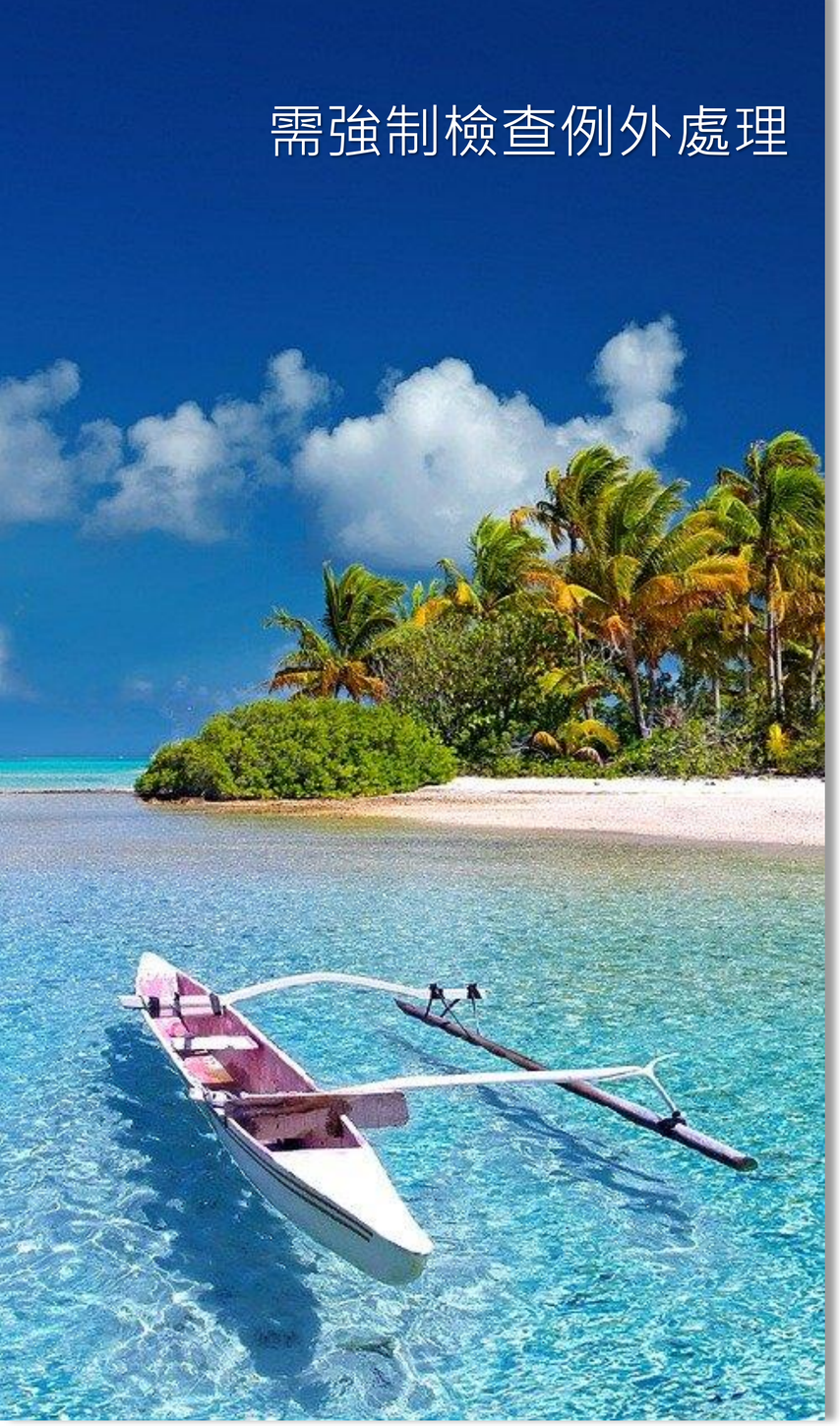


Unchecked Exception

- RuntimeException
  - 程式設計上疏忽造成
  - 此類例外雖可用例外處理機制保護，但不建議這麼做
  - 應該修正程式中的 Bug

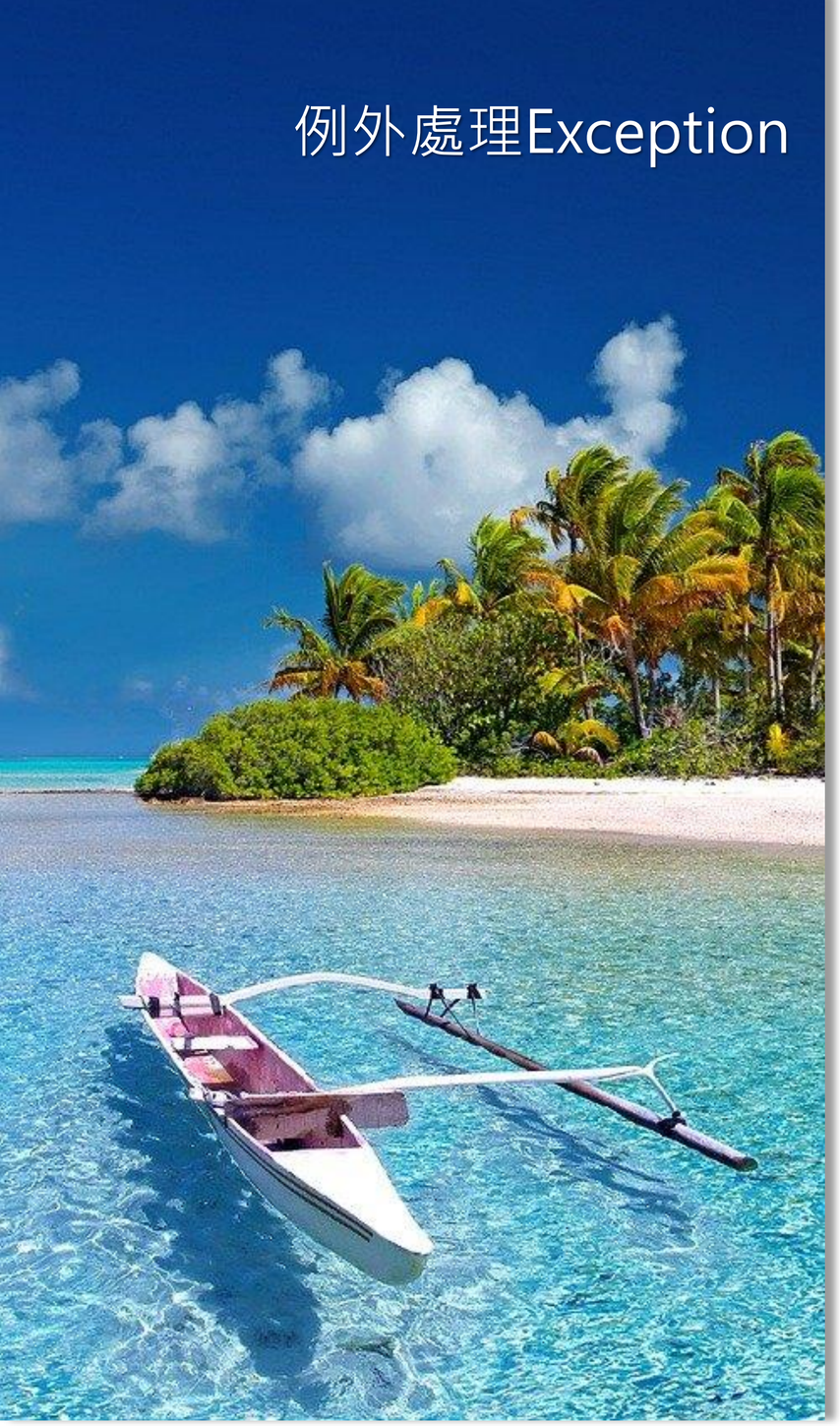
執行期例外	說明
ArithmeticException	數學運算時的例外。例如：某數除以 0。
ArrayIndexOutOfBoundsException	陣列索引值超出範圍。
NegativeArraySizeException	陣列的大小為負數。
NullPointerException	物件參照為 null，並使用物件成員時所產生的例外。
NumberFormatException	數值格式不符所產生的例外。

需強制檢查例外處理



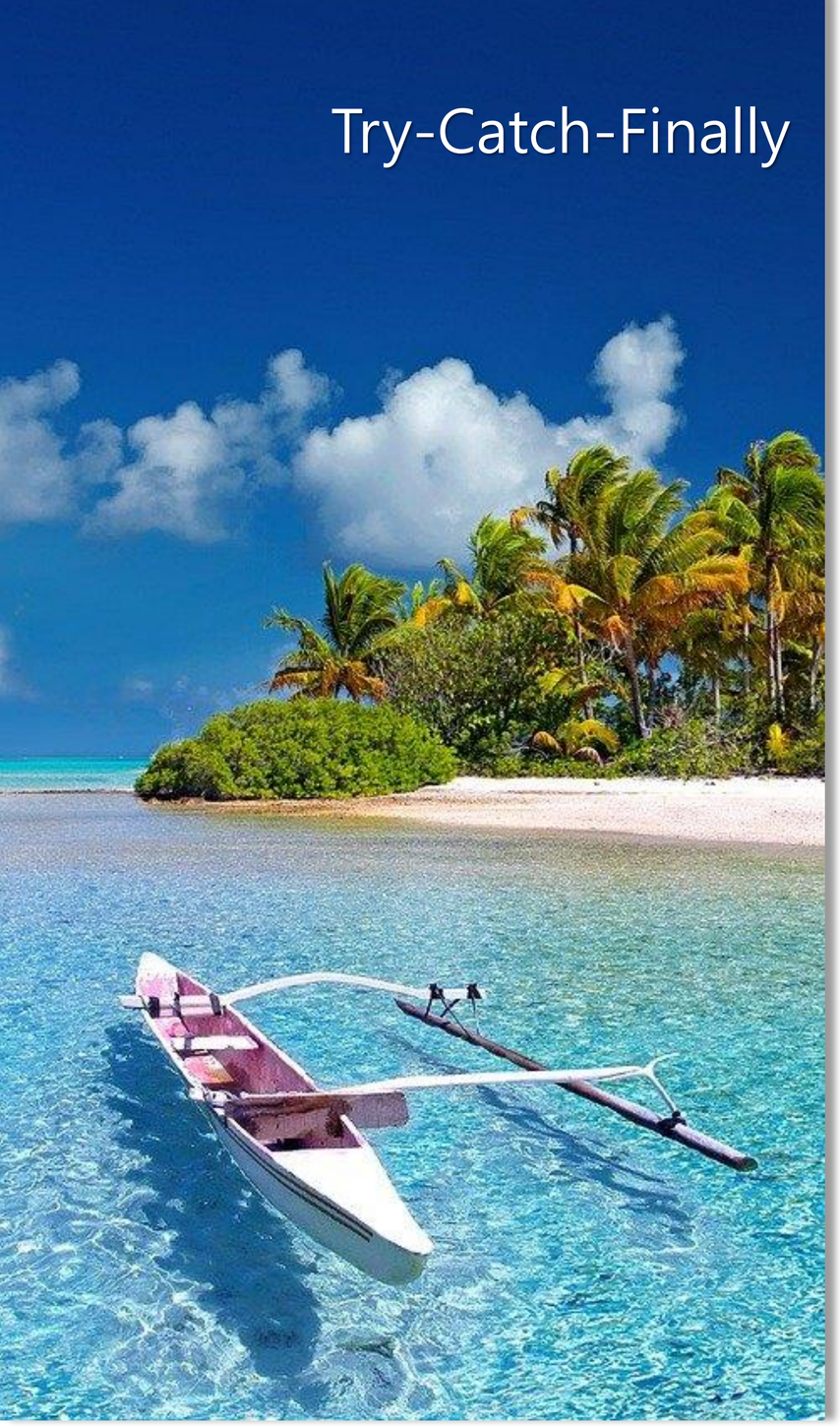
- Checked exception
  - Exception 的子類別，但不是 RuntimeException 的子類別
  - 可預期的外部因素造成的例外
    - 檔案不存在錯誤→FileNotFoundException
    - 輸出入處理錯誤→IOException
    - 資料庫處理錯誤→SQLException
    - 網路連結錯誤→SocketException
  - 系統強制程式中一定要作例外處理，否則編譯失敗

例外處理Exception



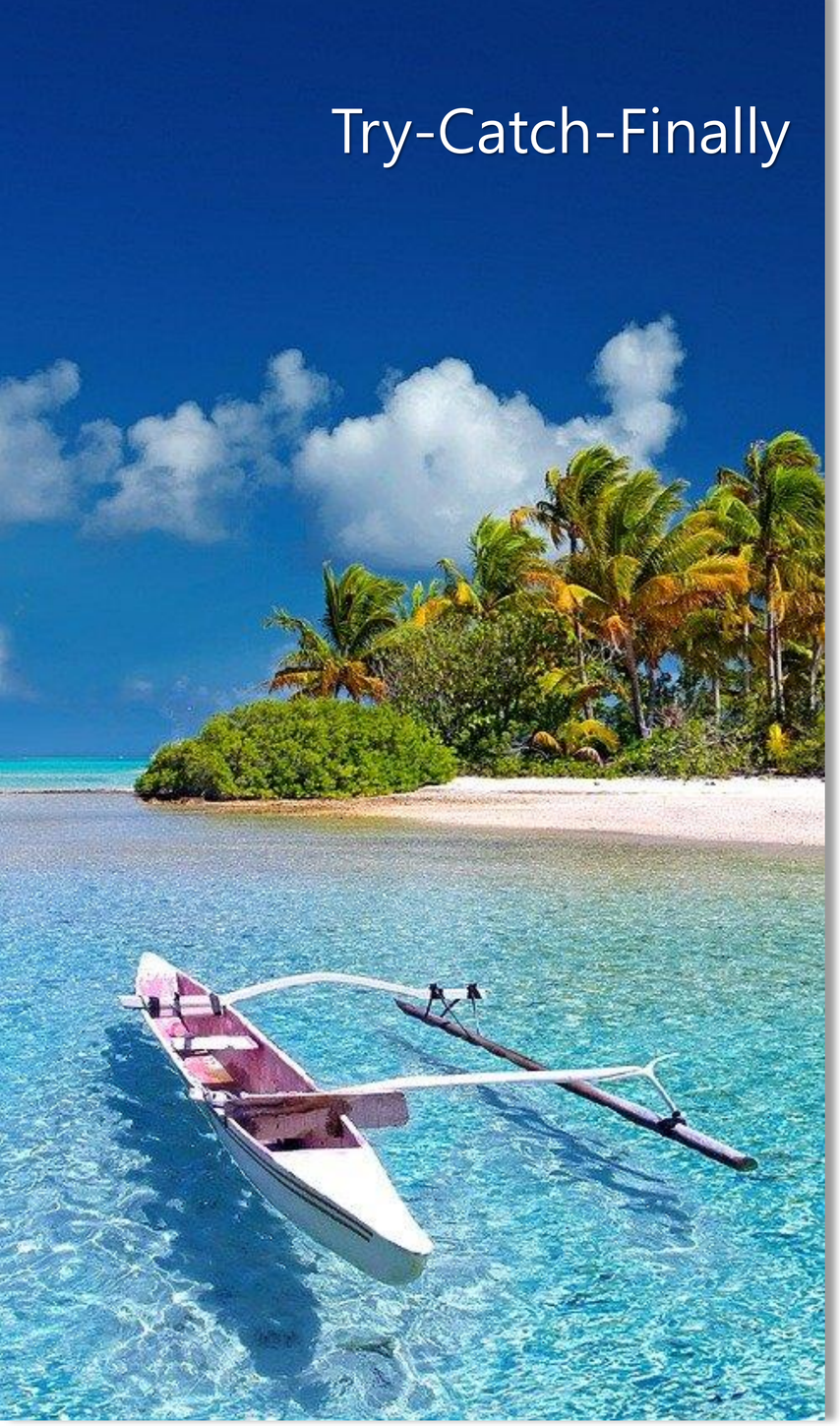
- 產生例外(丟)
  - 由系統自動產生
  - throw-吃飽沒事,無病申吟,自己產生,程式一定撰寫在方法內.
- 處理例外(接)
  - try-catch-finally(勇於負責,全交由我來處理)
  - throws(擺爛,交給上面的人來處理),程式撰寫在方法簽章

Try-Catch-Finally



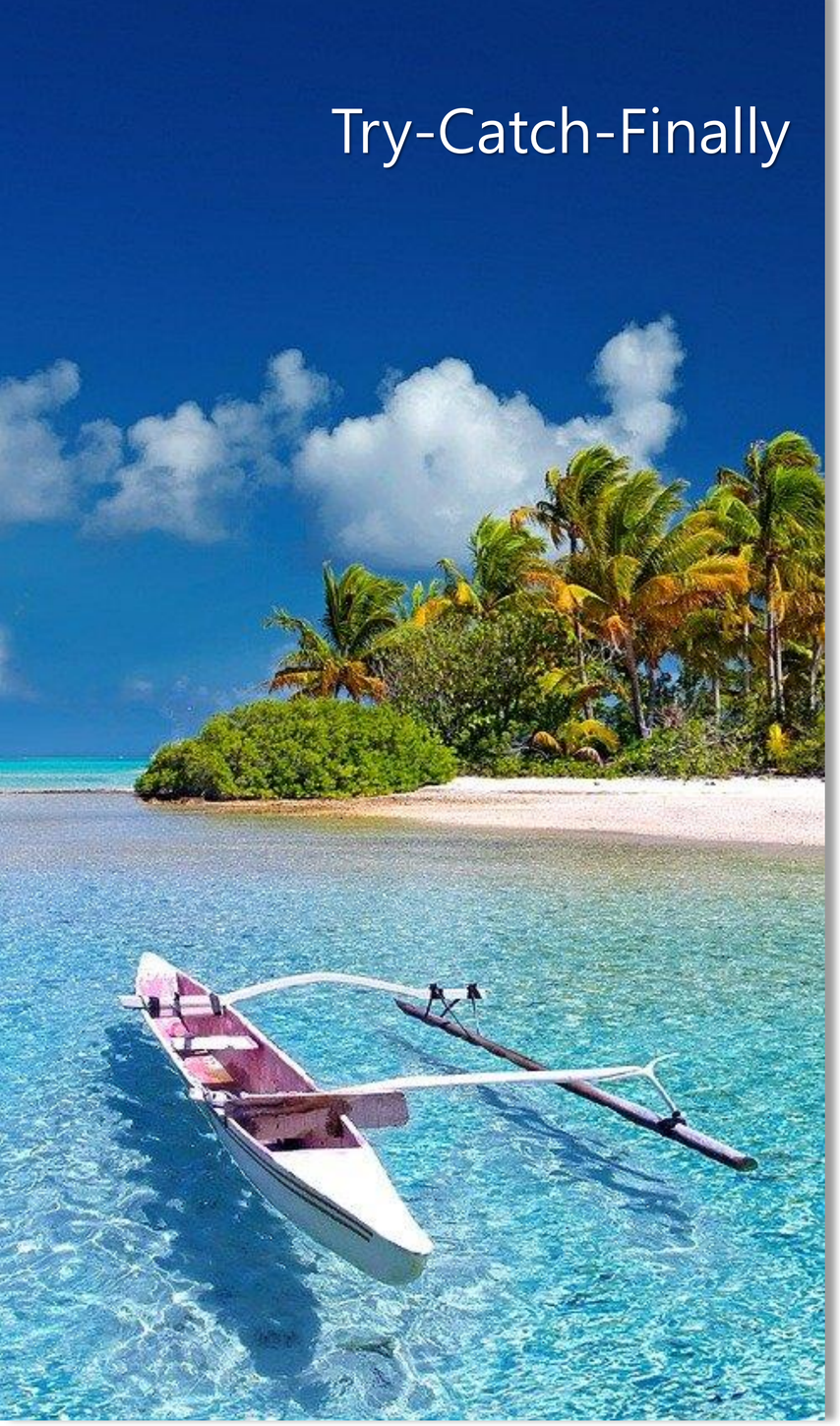
- 在Java程式語言中，try 程式區塊可與 catch或finally區段搭配使用，不過catch與finally一定要有try區塊才可執行，但try只要有catch或finally擇一即可
- Try與finally的區塊只能有一個，但catch的區塊可以有多個
- 當使用多個catch區塊時，若catch的類別彼此有直系繼承關係，在撰寫時必須先寫直系子類別，再寫直系父類別（編譯時期即會自動檢查)
- Finally此區段之程式是無論如何一定會執行到的程式.
- 例外顯示方式
  - System.out.println(e.getMessage());//回傳錯誤訊息
  - e.printStackTrace()//此方法為void,所以直接輸出

Try-Catch-Finally



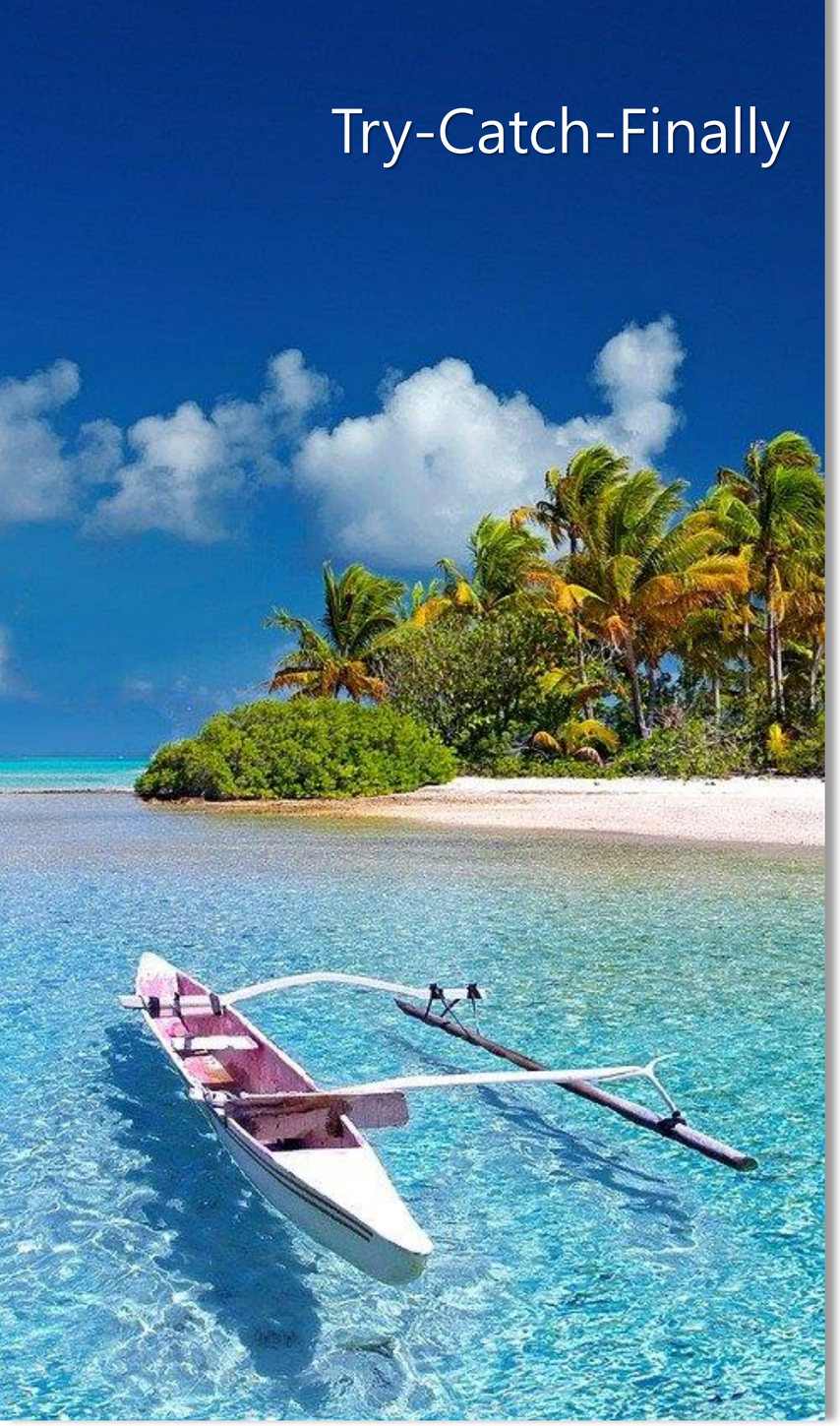
```
        Scanner sc=new Scanner(System.in);
        int denominator,numerator,max;
        int array[]={1,2,3,4,5,6};
        try{
        System.out.println("請輸入分子");
        numerator=sc.nextInt();
        System.out.println("請輸入分母");
        denominator=sc.nextInt();
        System.out.println("除法結果為:"+numerator/denominator);
        System.out.println("請輸入陣列最大值");
        max=sc.nextInt();
        for(int n=0;n<max;n++)
            System.out.print(array[n]+"\t");
        System.out.println();
        }
         catch(InputMismatchException e){
            System.out.println("輸入的格式錯誤");
            System.out.println(e.toString());//
        }
        catch(ArithmeticException e){
            System.out.println("分母不能為零");
            System.out.println(e.toString());//最簡單錯誤說明為e.getMessage,再來是e.toString要看詳細結果為e.printStackTrace(System.out);會包行錯誤行號
        }
        catch(ArrayIndexOutOfBoundsException e){
             System.out.println("\n陣列超出索引值");
            System.out.println(e.toString());
        }

        finally{
            System.out.println("這是一定會執行到的區塊 ,不管有無return 皆一定會執行到 ");
        }
```

Try-Catch-Finally



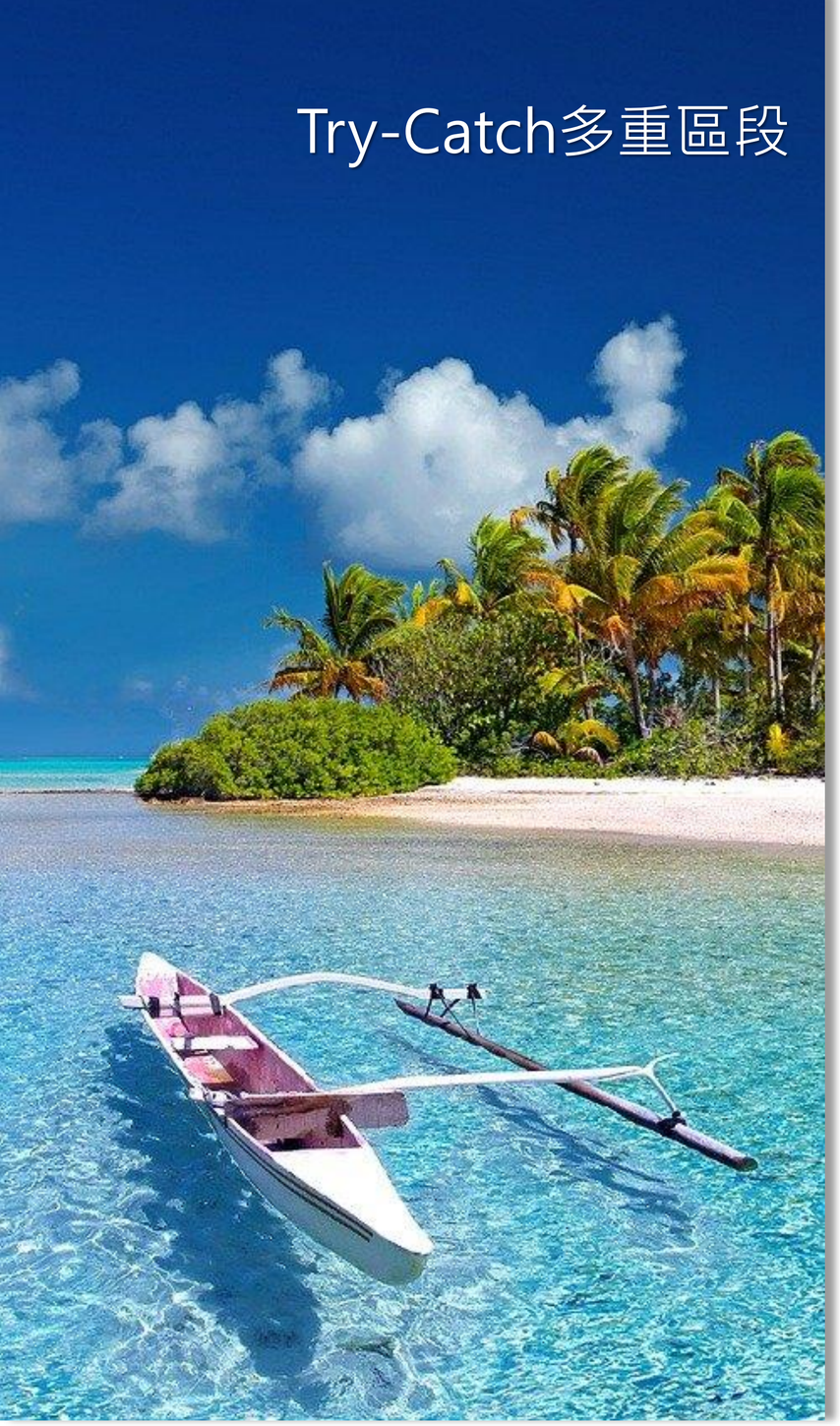
```
public static void main(String[] args) {
      int num=test();
      System.out.println(num);
   }
   public static int test(){
      try{
         int a[]=new int[5];
         System.out.println(a[5]);//此行可註解掉看程式執行結果
         return 1;
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("陣列超出索引值");
         return 2;
      }
      finally{
         System.out.println("我一定會執行");
         return 3;//練習時可先將此行註解掉,看其return1或return 2,但無論如何皆會執行到finally
      }
   }
```

Try-Catch-Finally



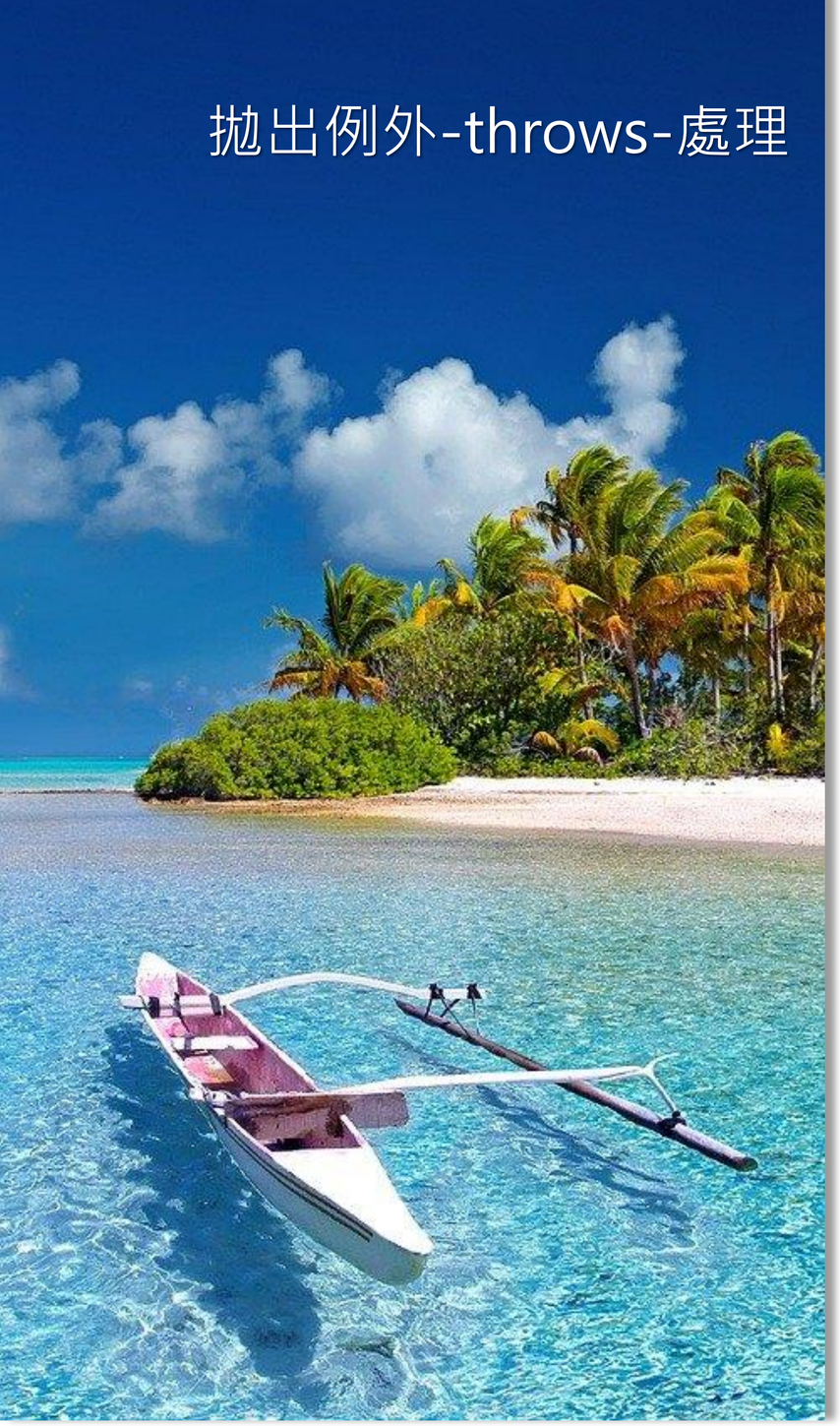
```
public class ReturnExceptionTest {
    static void methodA(){
    try{
        System.out.println("進入方法A");
        throw new RuntimeException("方法A沒事拋出例外");
    }finally{
        System.out.println("用方法A的finally區段,且一定會執行到");
    } }
    static void methodB(){
      try{
        System.out.println("進入方法B");
        return;
    } finally{ System.out.println("用方法 \"B\" 的finally區段,且一定會執行到");  }   }

 public static void main(String[] args) {
     try{
         methodA();
     }catch(Exception e){ System.out.println(e.getMessage());   }
     methodB();
 }}
```

Try-Catch多重區段



- catch(InputMismatchException ｜ArithmeticException e){
- System.out.println("輸入的格式錯誤");
- }
- 在多重區段內不能有繼承關係之例外

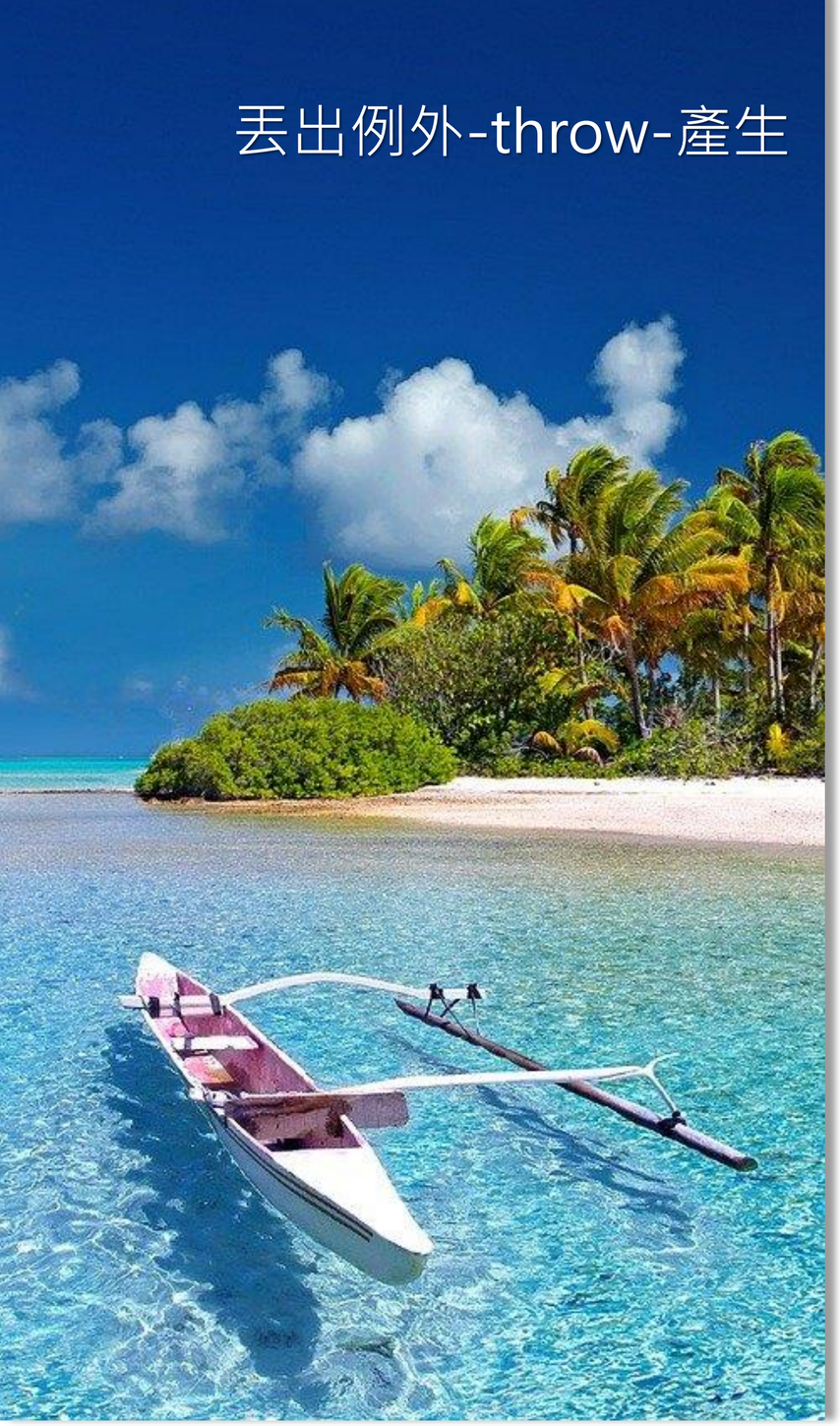
拋出例外-throws-處理



- 若是方法內不想處理例外,可於方法簽章中向上拋出例外
- 所以呼叫它的方法則需處理,若不處理也可再向上拋,直到由JVM處理
- 另外呼叫端的throws例外不能小於拋出端,即不能為拋出端的子類別
- 例外的覆寫-子類別的拋出例外需小於等於父類別或沒有拋出

  - 語法
  - 方法名稱(參數列) throws exception1,exception 2...{
  - }

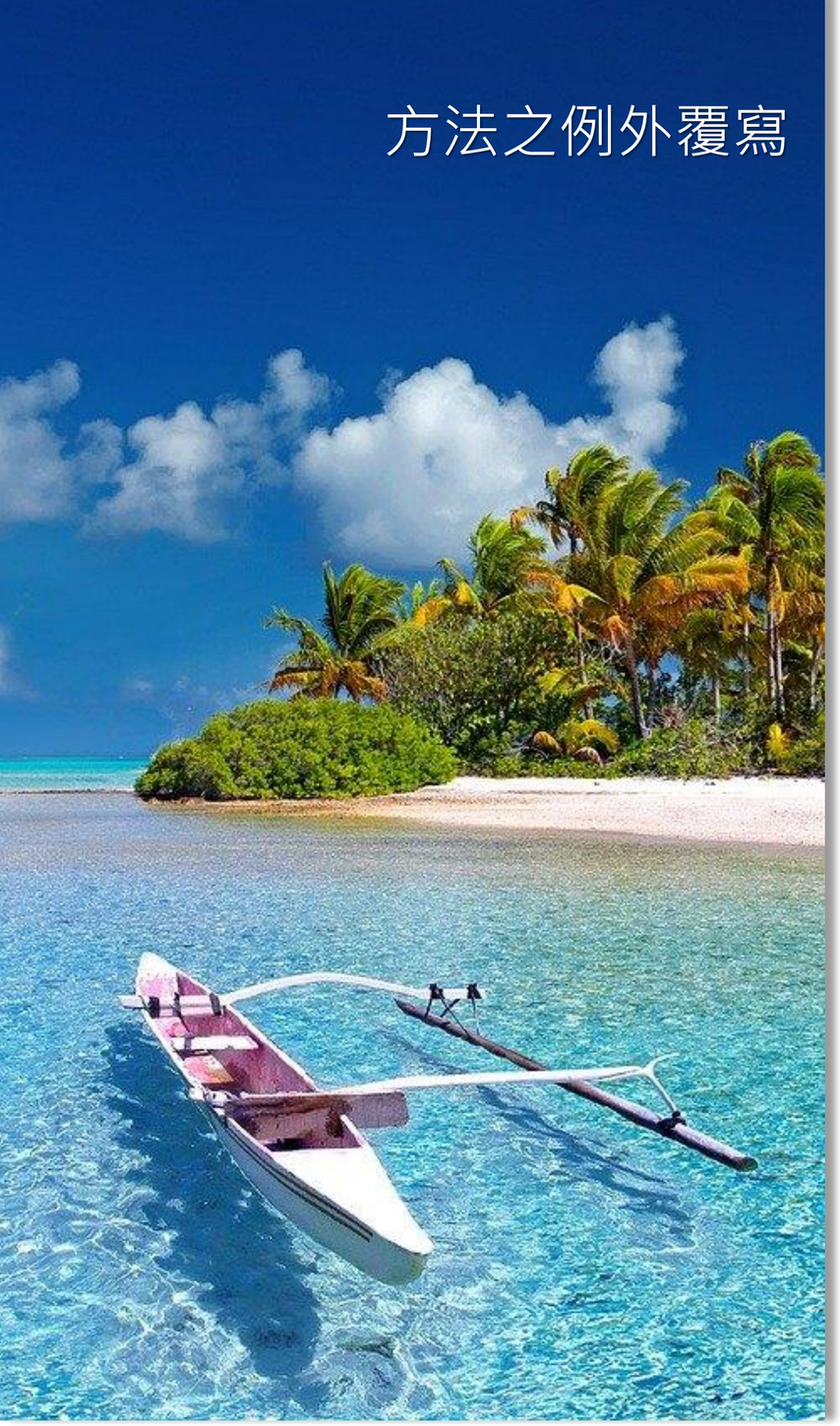
丟出例外-throw-產生



```
public void method( )
{
        throw new XXXException( );
}
```

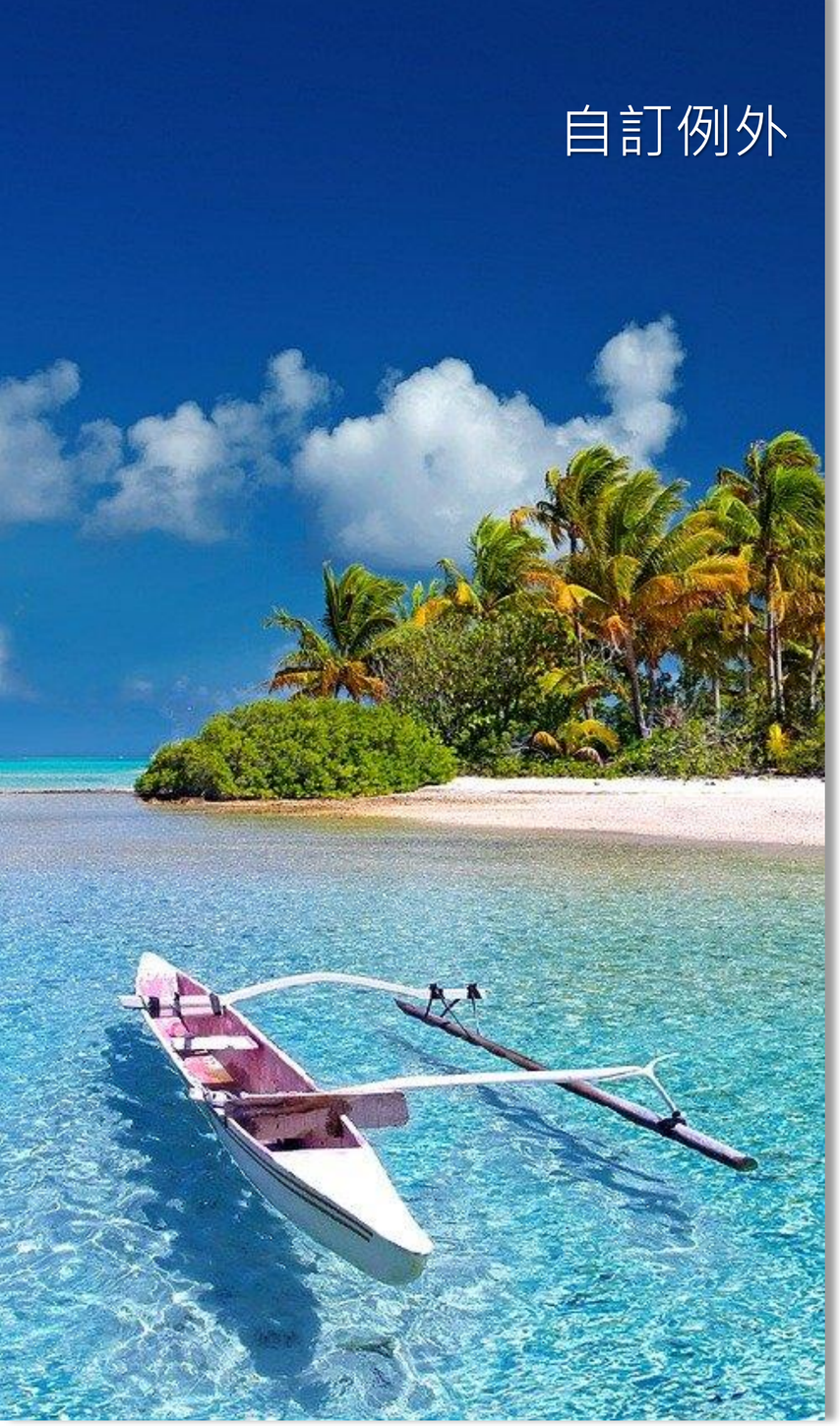
- Checked vs. Unchecked Exception
  - 方法內丟出 CheckedException，必須在方法簽章宣告 throws 此例外型別
  - 方法內丟出 Unchecked Exception (Error/RuntimeException)，可以不在方法簽章宣告 throws 此例外型別
  - 在方法簽章的例外需為同類別或父類別

方法之例外覆寫



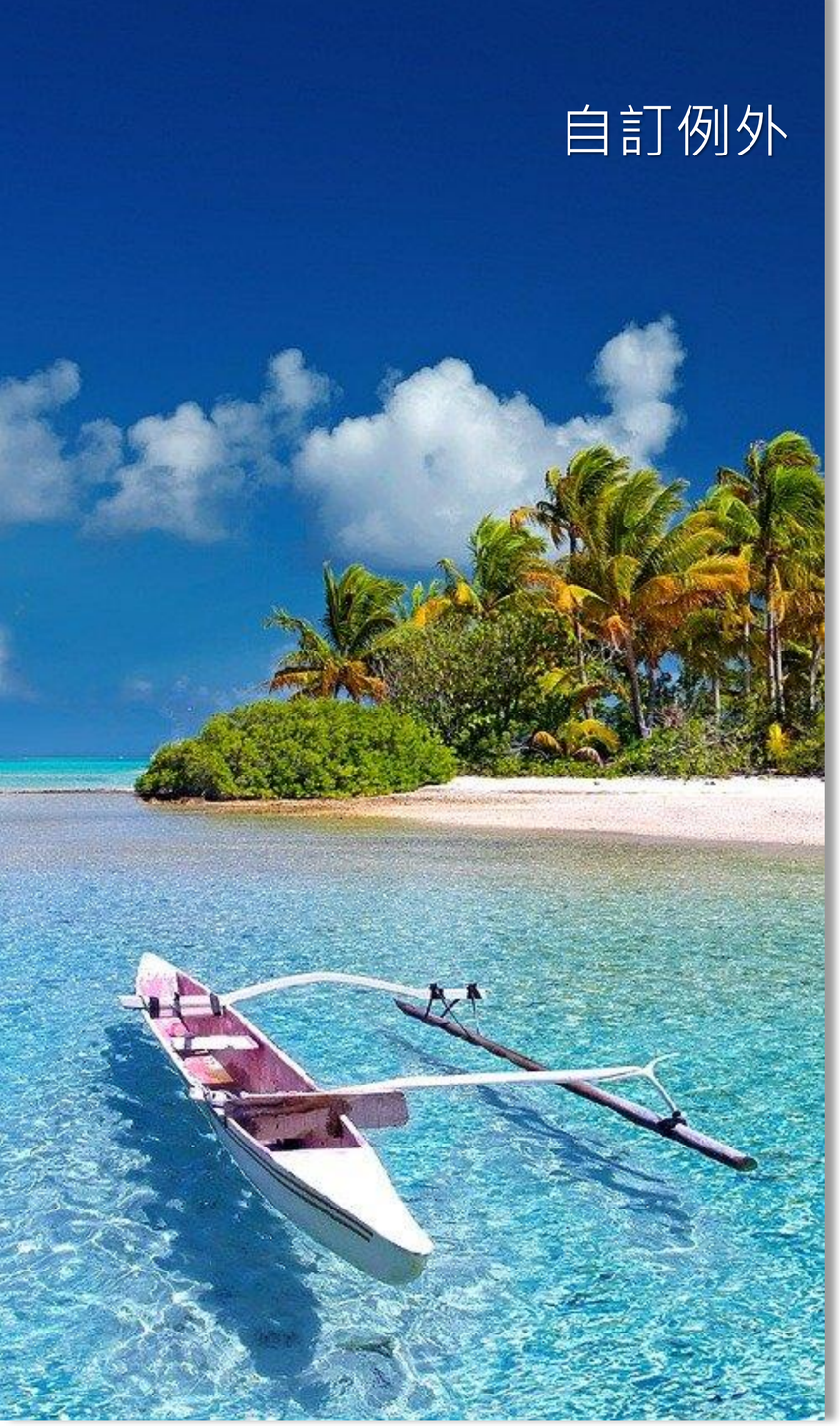
```
class Father{
    public void aMethod()throws Exception{}
    public void bMethod()throws Exception{}
    public void cMethod()throws IOException{}
}
class Son extends Father{
    public void aMethod(){}
    public void bMethod()throws IOException,SQLException{}
    public void cMethod()throws Exception{}
}
```

- 可以不使用**throws**來覆寫如**aMethod**
- 覆寫之例外需為子類別或是同類別如**cMethod**就無法執行



自訂例外

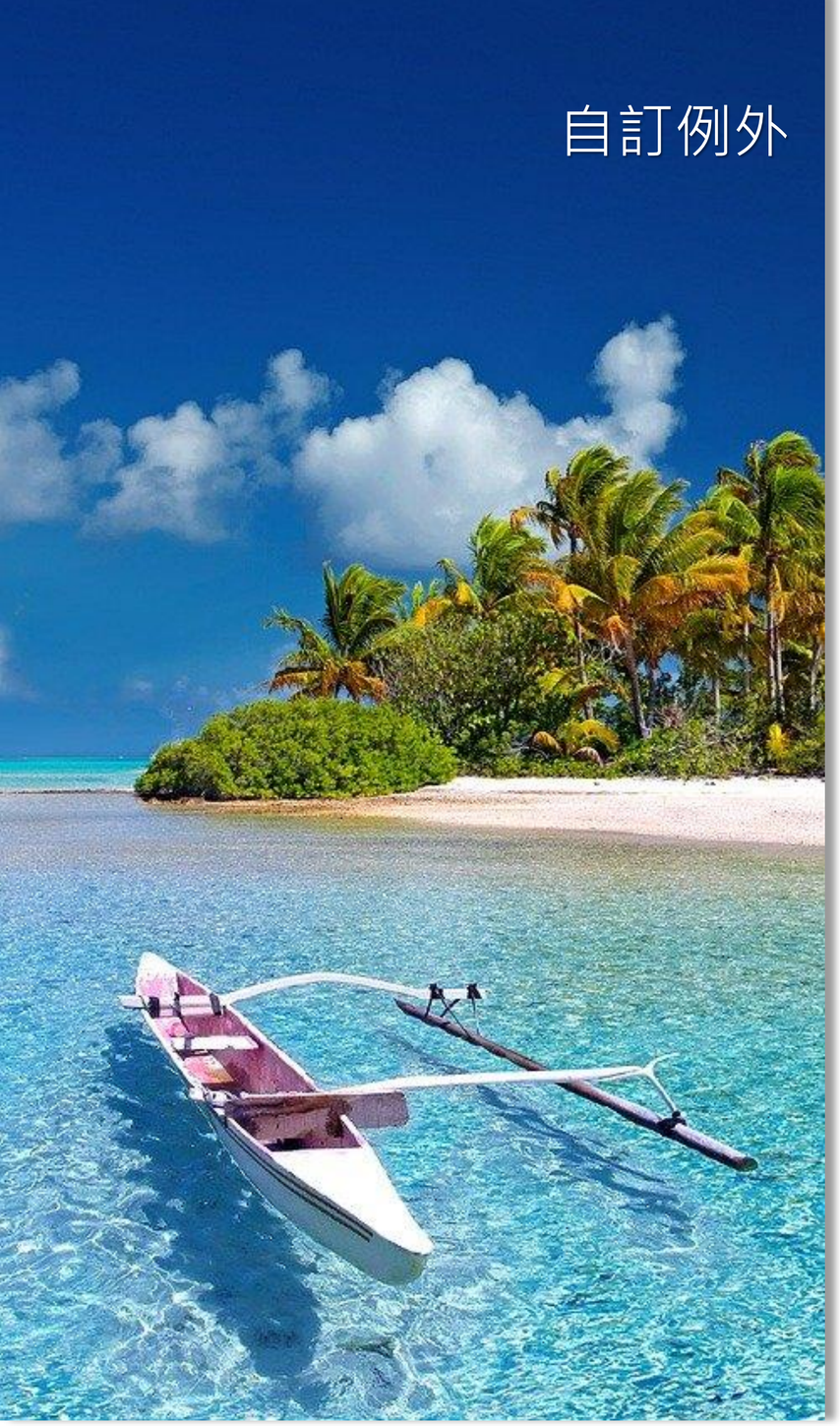
- 若要在程式中自訂例外類別，可利用extends關鍵字，將自身類別繼承Throwable類別或其子類別的方式來定義自定business例外類別
  - 例外錯誤類別的宣告方式
  -
  - 存取修飾字　修飾語　class　使用者自定例外類別名稱 extends 基礎例外類別{ //程式區塊 }
- 撰寫自訂例外的四個步驟
  - 產生自定例外類別
  - 在宣告方法時加入throws 修飾
  - 在方法中使用 throw new 自定例外類別 xxx 以丟出例外
  - 在 try-catch 程式區塊內呼叫該方法或是再 throws 出去



自訂例外

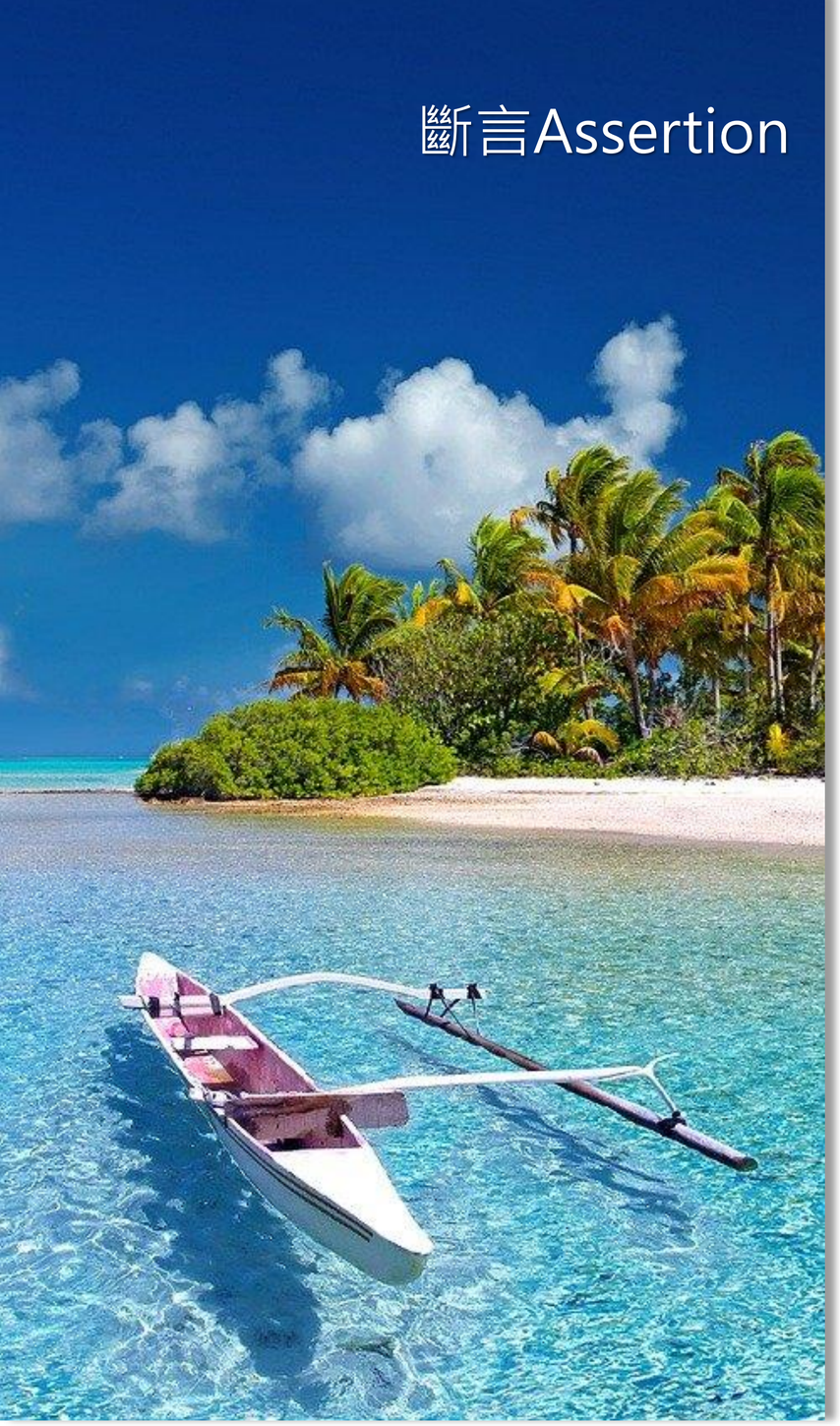
- 撰寫類別ExceptionTest.java,接收命令列的二個參數,要求不能輸入負數,並計算二數相除.
- 需針對下列例外進行處理
  - 資料型態不一致(NumberFormatException)
  - 缺少參數數目(ArrayIndexOutOfBoundsException)
  - 除零錯誤(ArithmeticException)
  - 輸入負數(自定例外類別 MinusException)
- 提示
  - 在類別中定義divMethod完成二數相除功能
  - 在main方法中使用例外處理,處理例外

自訂例外



```
public class ExceptionTest {
//宣告divMethod方法用來求二數相除(第二步驟)
   public static int divMethod(int a,int b) throws
MinusException{
      if(a<0 || b<0)
         throw new MinusException("輸入不能為負數");
      return a/b;
   }
public static void main(String[] args) {
   try{ int a=Integer.parseInt(args[0]);(第三步驟)
    int b=Integer.parseInt(args[1]);
      int result=divMethod(a,b);
      System.out.println(result);
   }catch(NumberFormatException e){
      System.out.println("必須輸入為數字");
   }catch(ArrayIndexOutOfBoundsException e){
      System.out.println("必須輸入二個數字");
   }catch(ArithmeticException e){
      System.out.println("除數不能為零");
   }catch(MinusException e){
      System.out.println(e.getMessage());
   }

   }
}
//自訂例外類別(第一步驟)
class MinusException extends Exception{
   MinusException(){};
   MinusException(String msg){
      super(msg);
   }

}
```

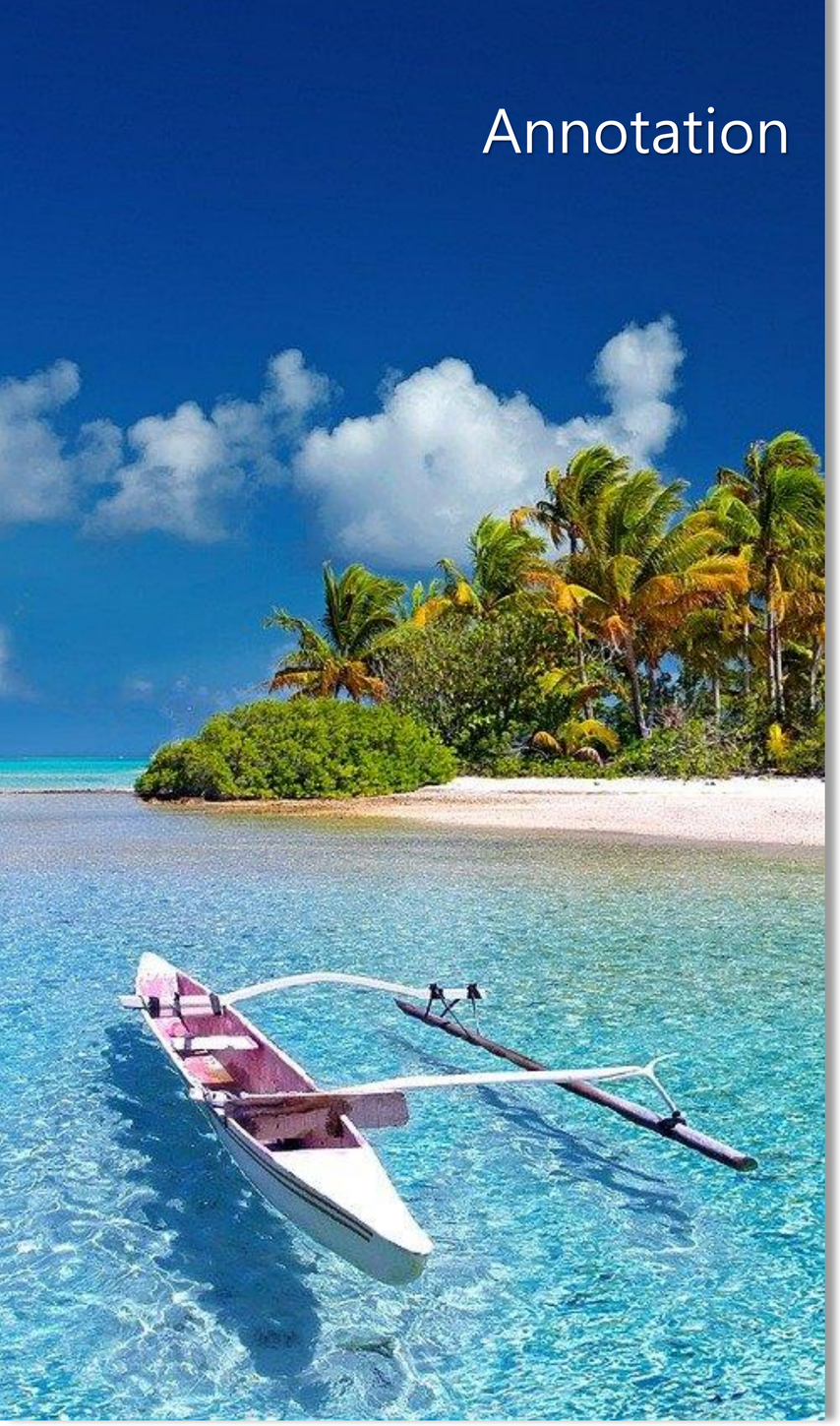
斷言Assertion



- 斷言
  - 用來斷定程式執行的某種結果，斷言若失準則被認為是嚴重的的問題，因為表示程式執行結果和預期有出入
- 語法
  - assert <boolean expression>;
  - assert <boolean expression> : <detail_expression>
  - 若布林運算式為false則拋出assertionError
  - 例 int x=5;assert(x<=0):"x一定大於等於零";
  - assertionError為Error的子類別，是屬於uncnecked exception
  - 啟動斷言　於properties中之run處加上-ea之參數,此時執行才會報錯

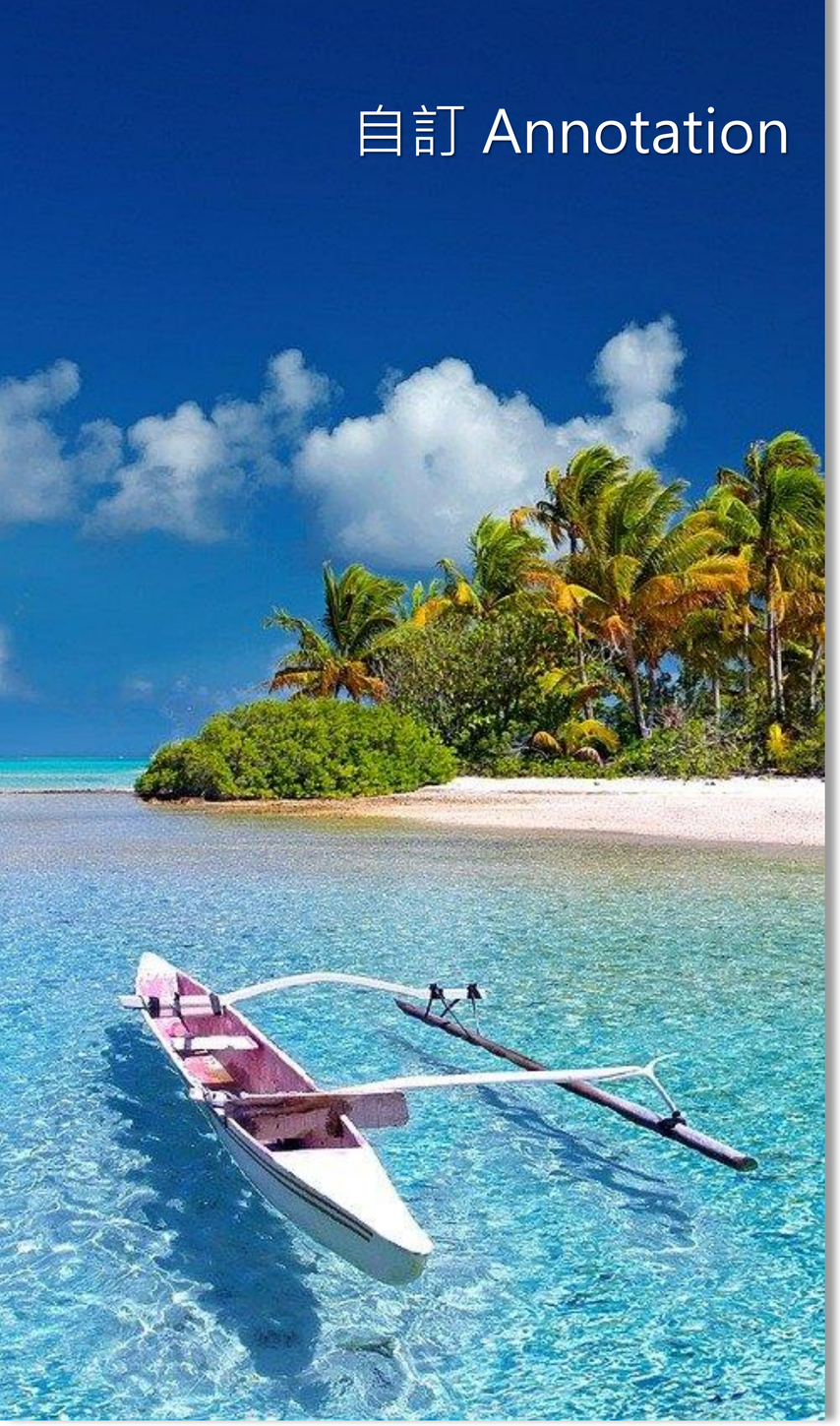
註解 Annotation

Annotation



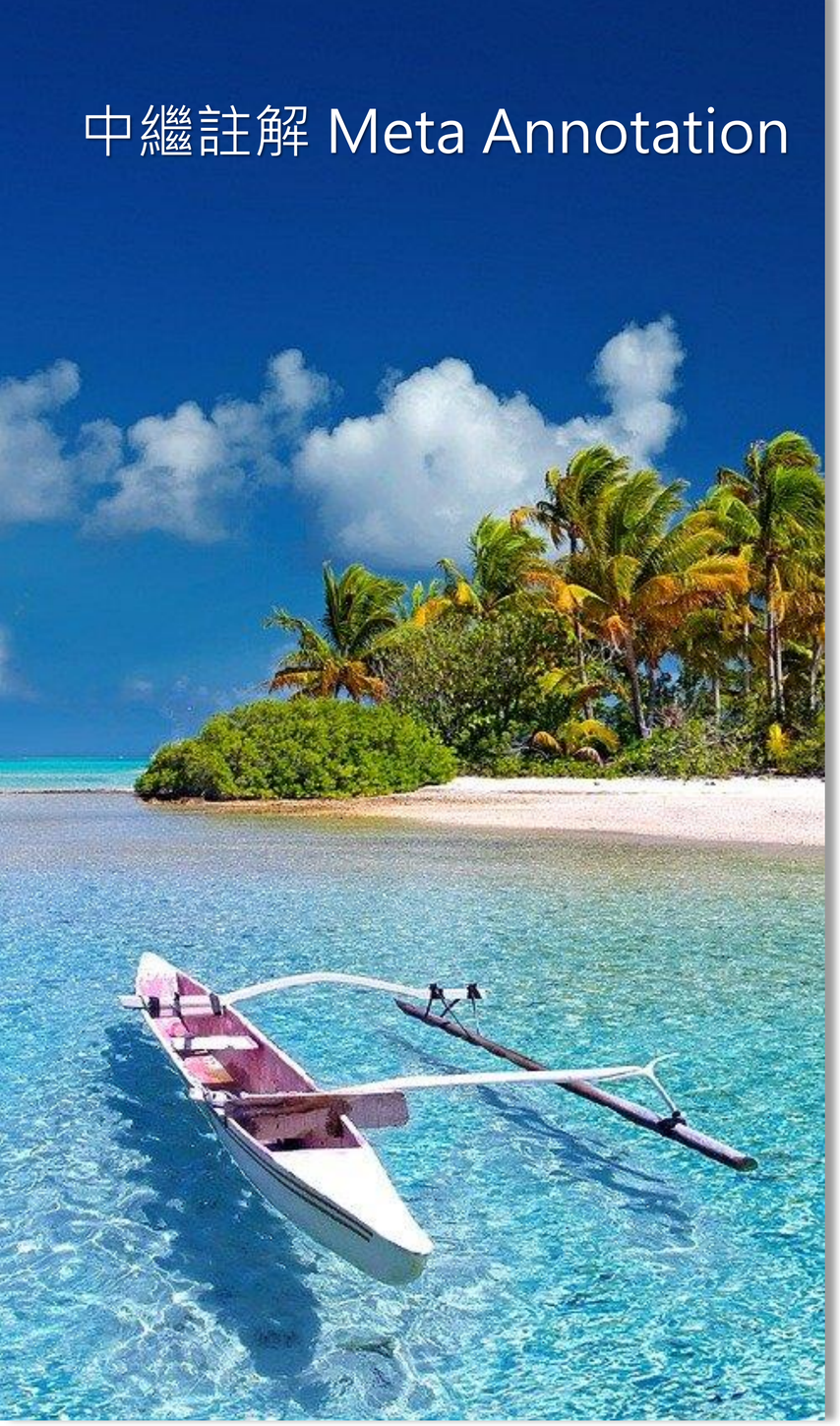
- Annotation 其實就是代碼的特殊標記,這些標記可以在編譯,類別加載,運行時被讀取,並執行其相對應的處理
- Annotation可以像修飾語一樣被使用,可用於修飾套件,類別,建構子,方法,成員變數,參數,區域變數的宣告,這些訊息會被保存在Annotation的name=value 中
- 產生說明文件的相關註解
  - @author 作者,多個作者以, 分開
  - @version 版本
  - @see 參考相關
  - @since 開發於那個版本
  - @param 對方法中參數的說明,如果無參數則不能加此註解
  - @return 對方法中返回值的說明,如果void則不能加此註解
  - @exception 對方法中可能拋異常進行說明,如果沒有throws則不能加此註解
- 在編譯時進行格式檢查
  - @Override 覆寫
  - @FunctionInterface 函數式介面
  - @Deprecated 過期
  - @SafeVarargs 抑制不特定長度參數所發生的 heap pollution 警告
    - class MyUtils{
    - @SafeVarargs
    - public static <T> void add(T...a)
    - { //do something } }

自訂 Annotation



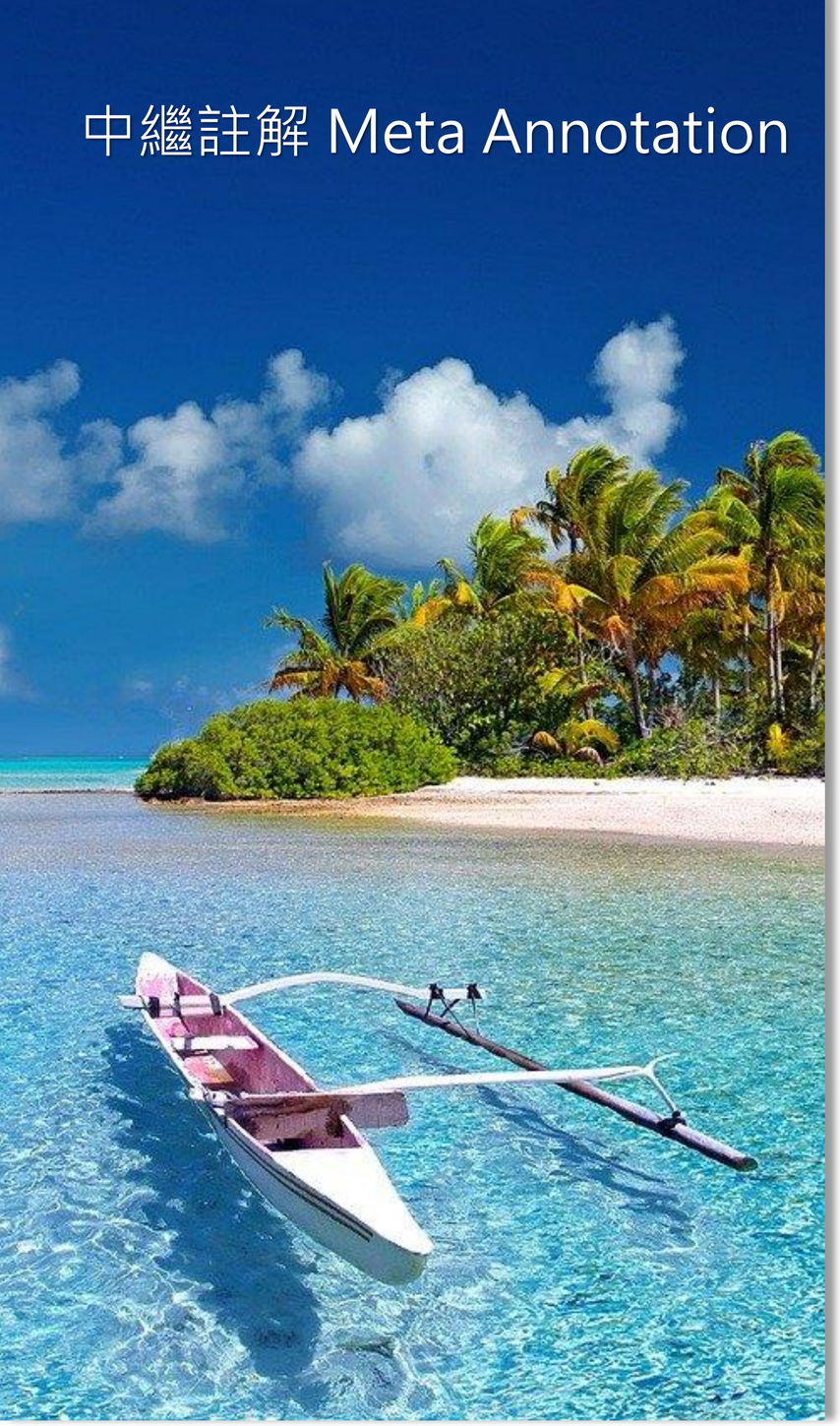
- 使用@interface關鍵字來自訂註解 @interface 註解名稱
- 自訂註解自動繼承了java.lang.annotation.Annotation介面
- Annotation的成員變數在Annotation定義中以無參數方法的形式來宣告,其方法名和回傳值定義了該成員的名字和類型,我們稱為配置參數
- 例 String value( ) default "預設值";
- 類型只能是八種基本資料型態,和String,Class類型,enum類型,Annotation類型,以上所有類型的陣列格式
- 可以在定義成員變數時設定初始值,也可以使用default關鍵字
- 如果只有一個參數成員,建議名為value
- 如果定義的註解含有配置參數,那麼使用時必須指定參數值,除非它有預設值,格式是 "參數名稱=參數值", 如果只有一個參數變數,且名稱為value,則可以省略value=
- 例
- public @interface MyAnnotation{
- String name( ) default "預設值";
- int value( ) default 25;
- int[] score() default {25,35,45};
- //Integer age() default 3; 不可以是包覆類別
- }
- //套用MyAnnotation
- @MyAnnotation(name="amy", score= {11,22,33})//不能直接寫 "amy" 因為其名稱不是 value
- class Test{}
- @MyAnnotation(15)//只有整數不用參數名稱,因為其參數名稱為 value
- class Test1{}
- @MyAnnotation //皆不加參數可以,因為二個參數皆有 default
- class Test2{}

中繼註解 Meta Annotation



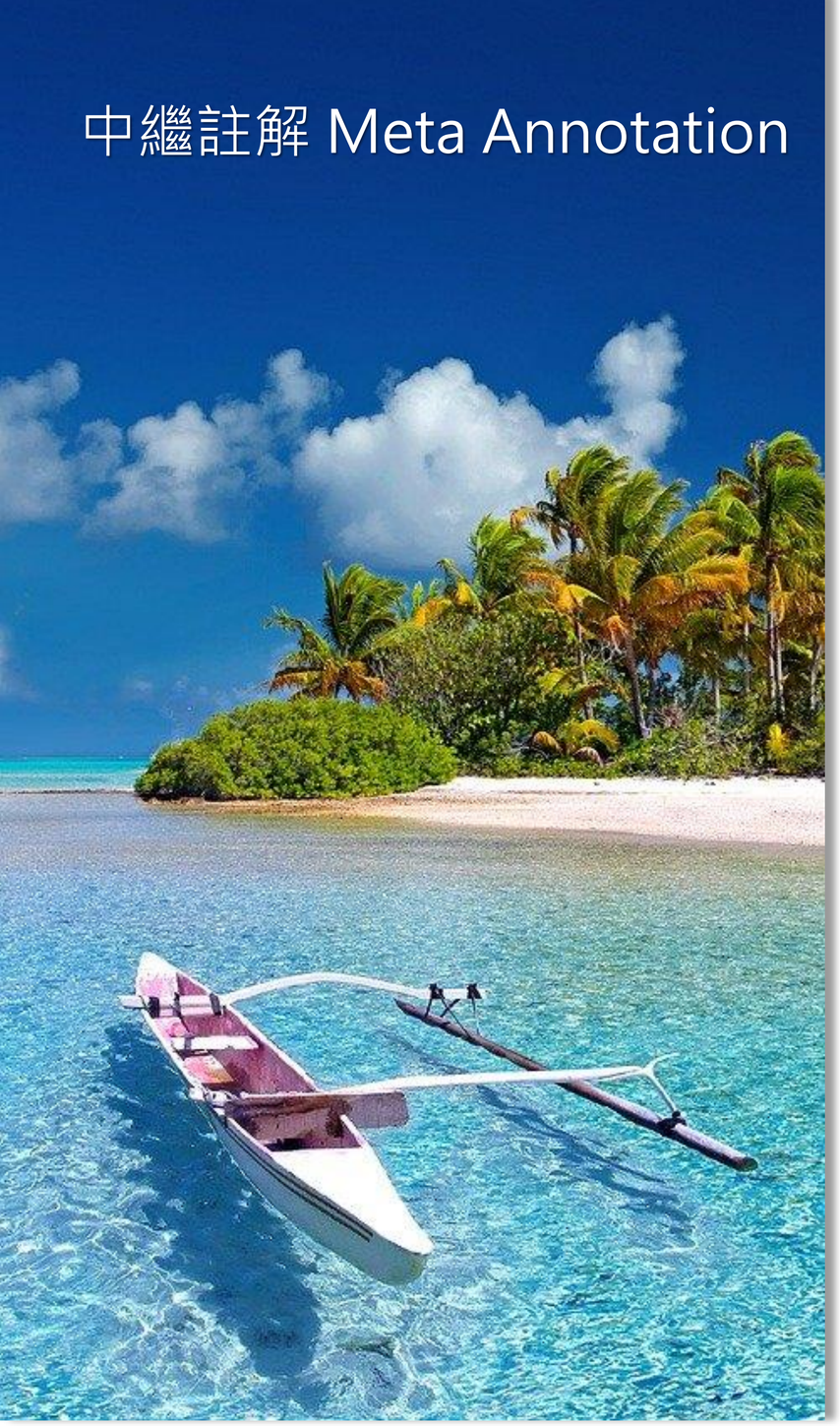
- 即是用來修飾註解用的註解，在jdk提供了四種中繼註解,需加在自訂註解類的上方如下所示
  - @Retention( ) 中括號內才撰寫下方的三種值
    - 其中Rentention為註解要保留到什麼時候，其中三個值皆存於RetentionPolicy屬性中，另外它有三個屬性
    - RetentionPolicy.SOURCE 註解只存於原始碼中
    - RententionPolicy.CLASS 註解存於編譯的class檔中(預設值)
    - RententionPolicy.RUNTIME註解存的生命最久，執行時還在,反射機制只能使用此種
  - @Target({x,x,x}) //可按Ctrl點註解看source code
    - 作用在特定類型的中繼註解
    - 其參數為陣列,所以寫值需以大括號括住
    - @Target({TYPE, FIELD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, MODULE})//TYPE 即類別
    - 需import static java.lang.annotation.ElementType.*才可直接寫欄位,否則需寫完整ElementType.FIELD
  - @Documented
  - @Inherited
  - @Repeatable 可重覆
  - Target類型新增二個
    - TYPE_PARAMETER 用來修飾泛型
    - TYPE_USE用來修飾變數的轉型宣告
    - ArrayList<@MyAnnotation("hello") String> al=new ArrayList<>();//此時只要加上TYPE_USE
    - class Generic<@MyAnnotation("hello") T>{//需加上TYPE_PARAMETER
    - int num=(@MyAnnotation int) 10.5; //此時只要加上TYPE_USE

中繼註解 Meta Annotation



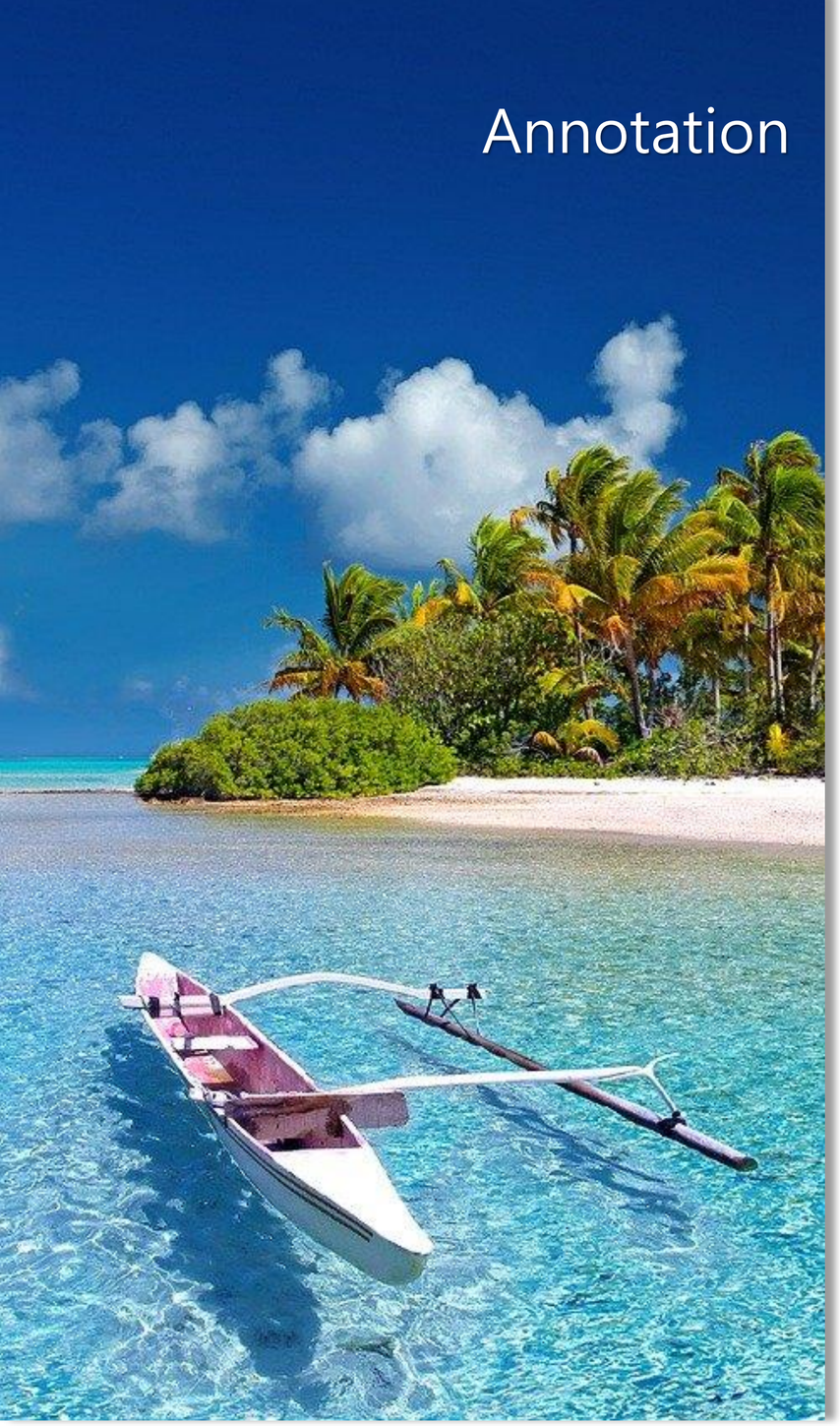
- //@MyAnnotations({@MyAnnotation("hello"),@MyAnnotation("again")}) jdk8 以前的寫法
- //jdk8 以後的寫法
- @MyAnnotation("hello")
- @MyAnnotation("again")
- **class ProcessOrders{ }**
- @Target({**TYPE,FIELD,METHOD,PACKAGE,LOCAL_VARIABLE**})
- @Retention(**RUNTIME**)
- @Repeatable(MyAnnotations.**class**)
- **@interface MyAnnotation{**
- String[] value();
- }
- @Target({**TYPE,FIELD,METHOD,PACKAGE,LOCAL_VARIABLE**})
- @Retention(**RUNTIME**)
- **@interface MyAnnotations{**
- MyAnnotation[] value();
- }

中繼註解 Meta Annotation



```
//@Inherited
public class ForTest {
        public static void main(String[] args) {
                Class t=Son.class;
                Annotation[] ann=t.getAnnotations();
                System.out.println(Arrays.toString(ann));
        }
}

//Annotation--begin
@Inherited
//保留政策要為RUNTIME執行時期才可以
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation{
        String value( ) default "java";
        int age() default 50;
}
//Annotation--end

//若有二個值,則參數名稱雖為 value 也是要寫
@MyAnnotation(value="inheritance test",age=120)
class Father{}
class Son extends Father{}
```

Annotation

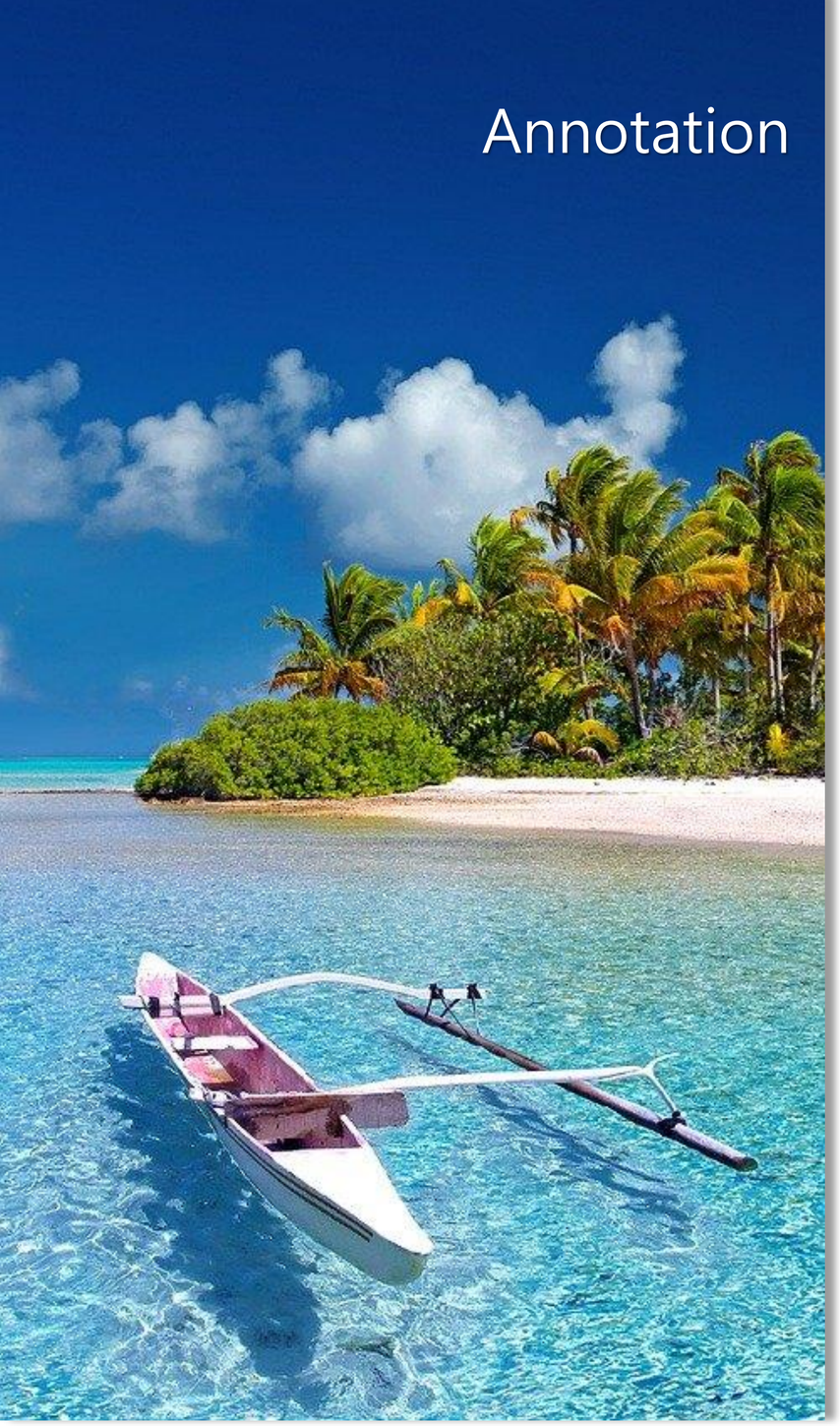


Which two annotations may be applied at Loc1 in the code fragment ?

```
@interface Resource{
    String[] value();
}
Examine this code fragment:
/*Loc1 */ class ProcessOrders{ ... }
```

- [ ] A. @Resource( )
- [ ] B. @Resource(value={{ }})
- [ ] C. @Resource("Customer1")
- [ ] D. @Resource
- [ ] E. @Resource({"Customer1" , "Customer2" });

Annotation

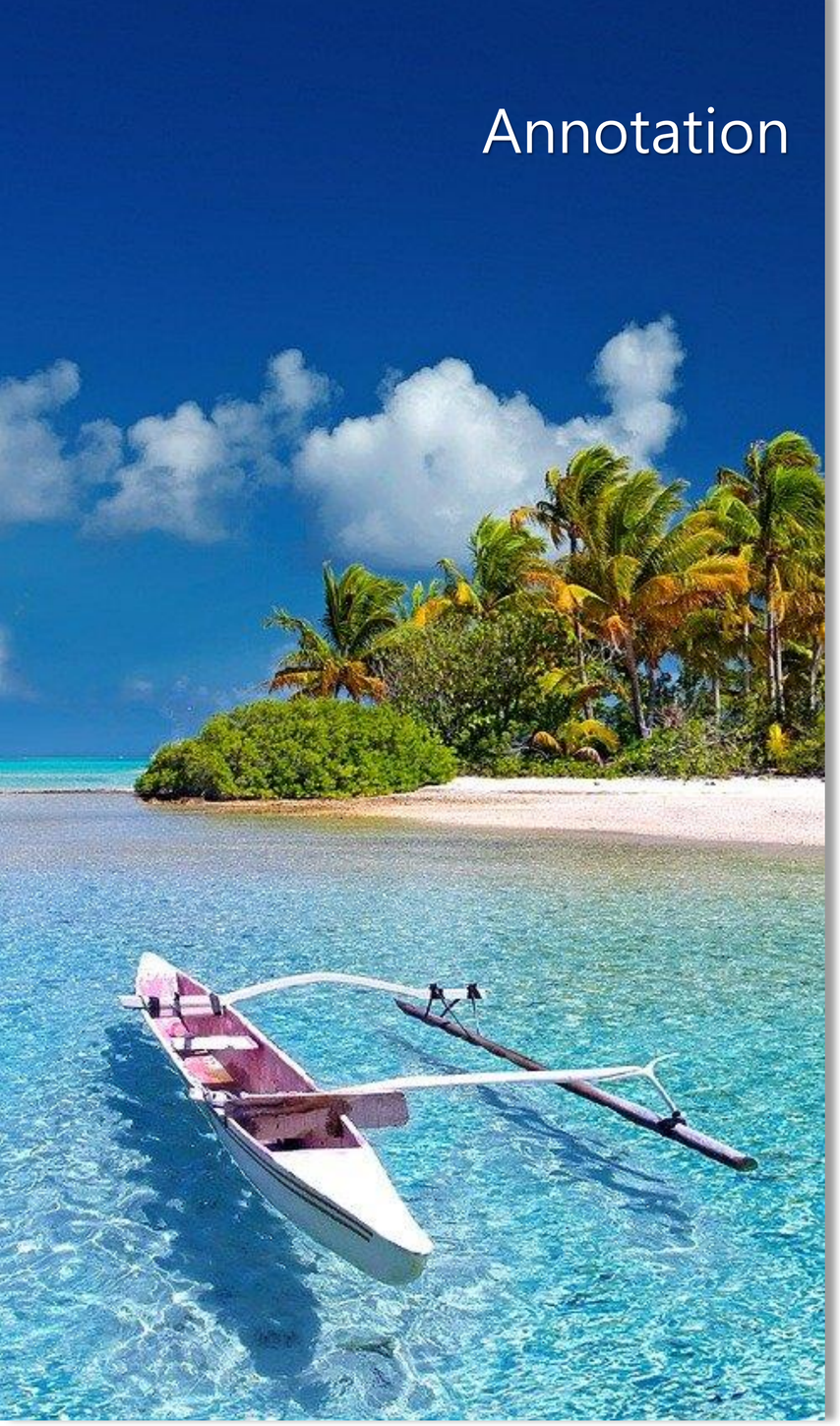


Which two annotations may be applied at Loc1 in the code fragment ?

```
@interface Resource{
        String name();
        int priority() default 0;
}
/*Loc1*/class ProcessOrders{
}
```

- [ ] A. @Resource(priority=100)
- [ ] B. @Resource(priority=0)
- [ ] C. @Resource(name="Customer1", priority=100)
- [ ] D. @Resource(name="Customer1")
- [ ] E. @Resource

Annotation



Which two are correct ? *

A.

```
@AuthorInfo(date="1-1-2020",comments= {null})
class Hello{
        public void func() {}
}
```

B.

```
class Hello{
        @AuthorInfo (date="1-1-2020",comments= {"world"})
        public void func() {}
}
```

C.

```
class Hello{
        @AuthorInfo
        public void func() {}
}
```

D.

```
@AuthorInfo (date="1-1-2020")
class Hello{
        public void func() {}
}
```

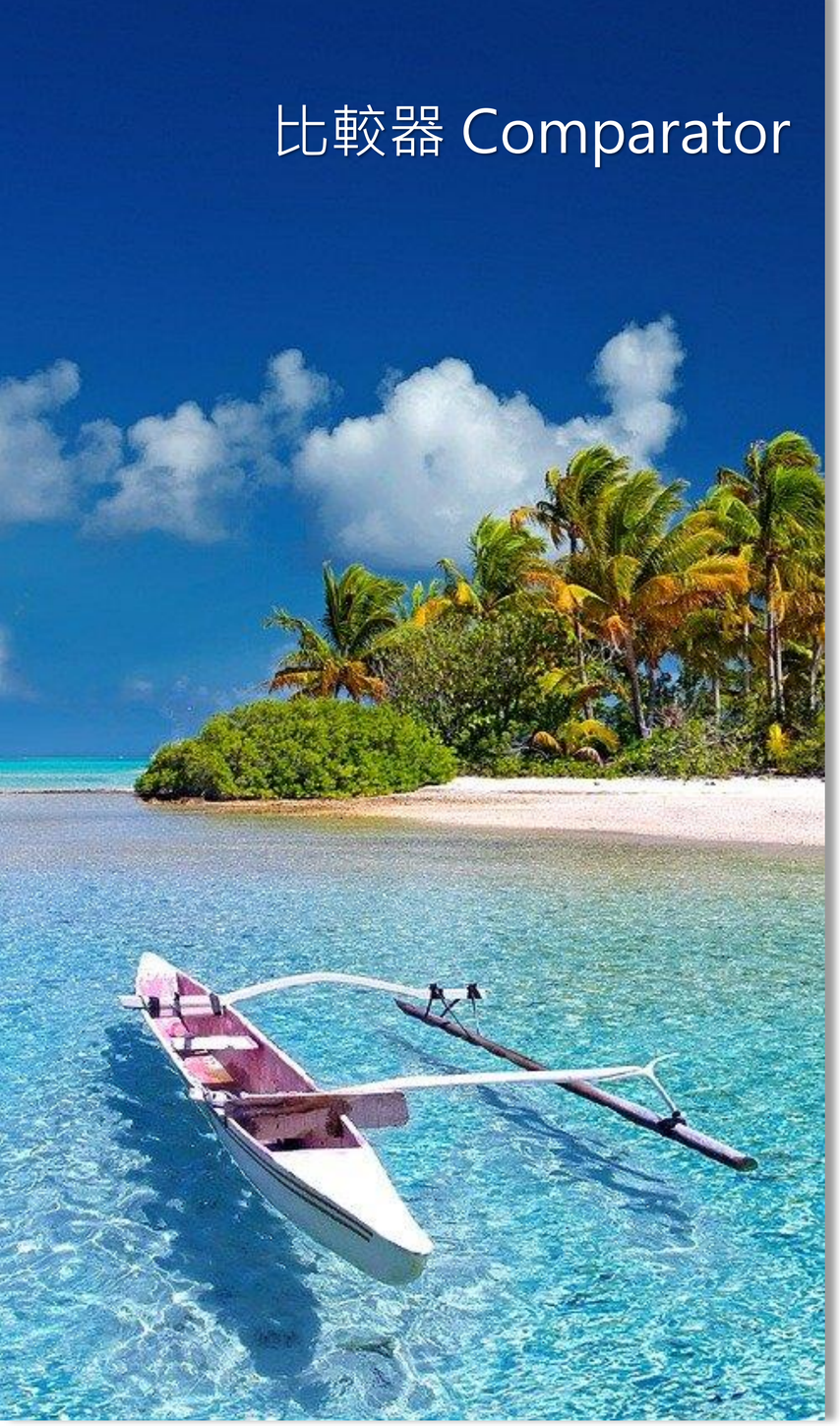
E.

```
class Hello{
@AuthorInfo (date="1-1-2020",author="James",comments= {"world"})
public void func() {}
}
```

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@interface AuthorInfo{
        String author() default "";
        String date();
        String[] comments() default {};
}
```

比較器 COMPARATOR

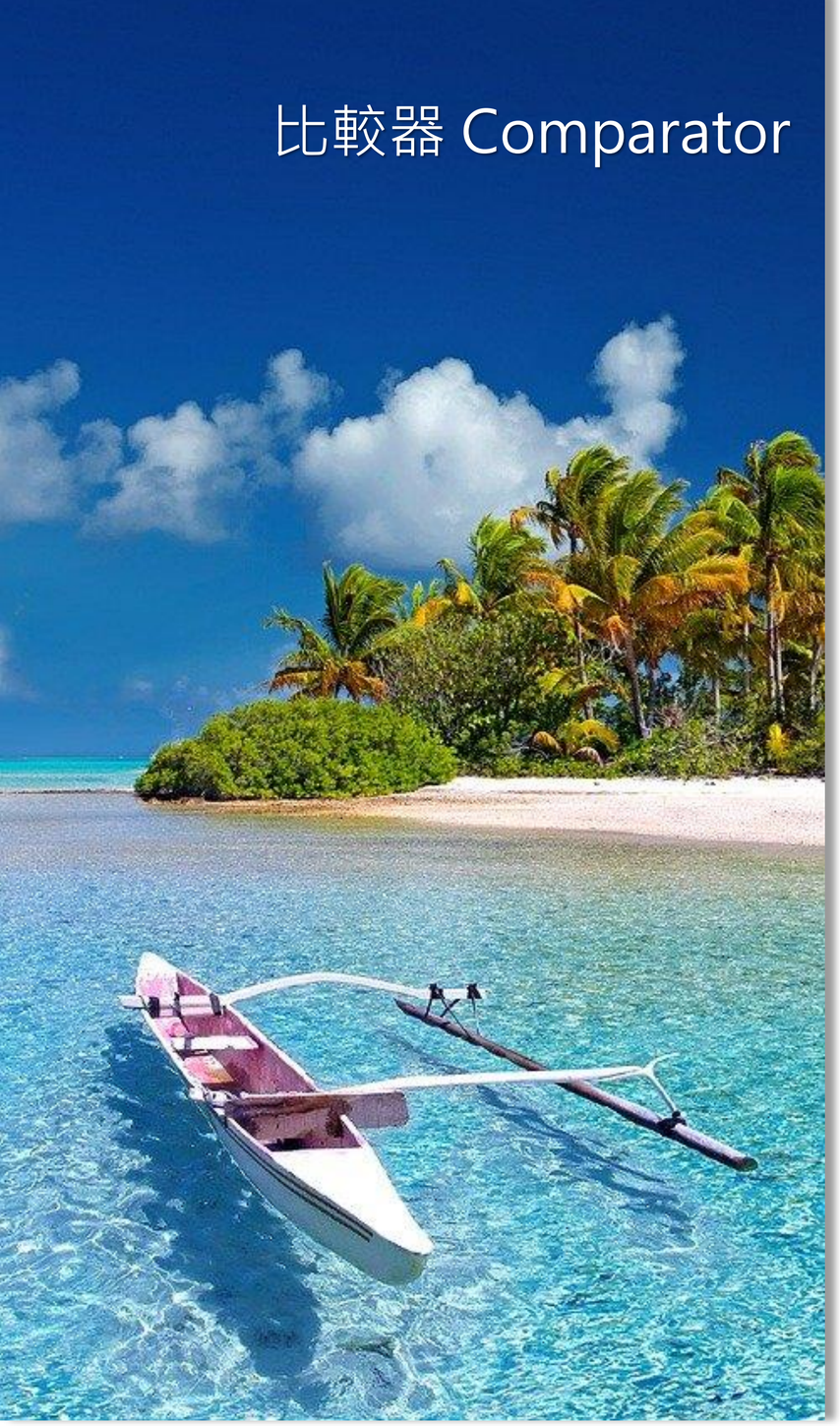
比較器 Comparator



java的物件,正常情形下只能進行比較 == 或 != ,無法使用 > , < 的運算子,若需要對多個物件進行排序,則需使用比較器,如String 實作Comparable介面,並覆寫了其compareTo的方法

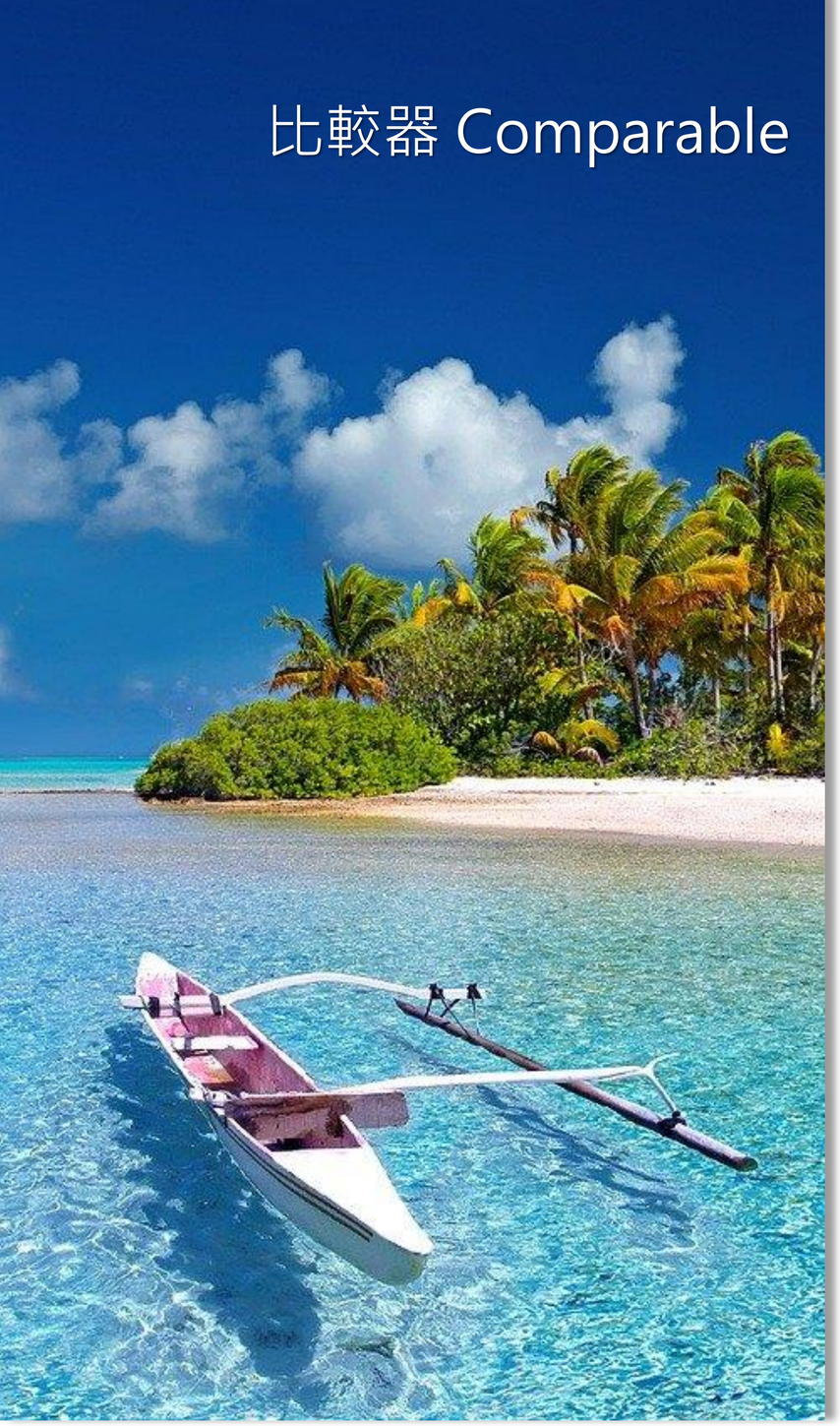
- Comparable介面
  - 必需實作compareTo方法
- Comparator介面
  - 必需實作compare方法
- 二個介面的方法回傳值皆是整數
  - 0表示相等
  - >0表示大於
  - <0表示小於

比較器 Comparator



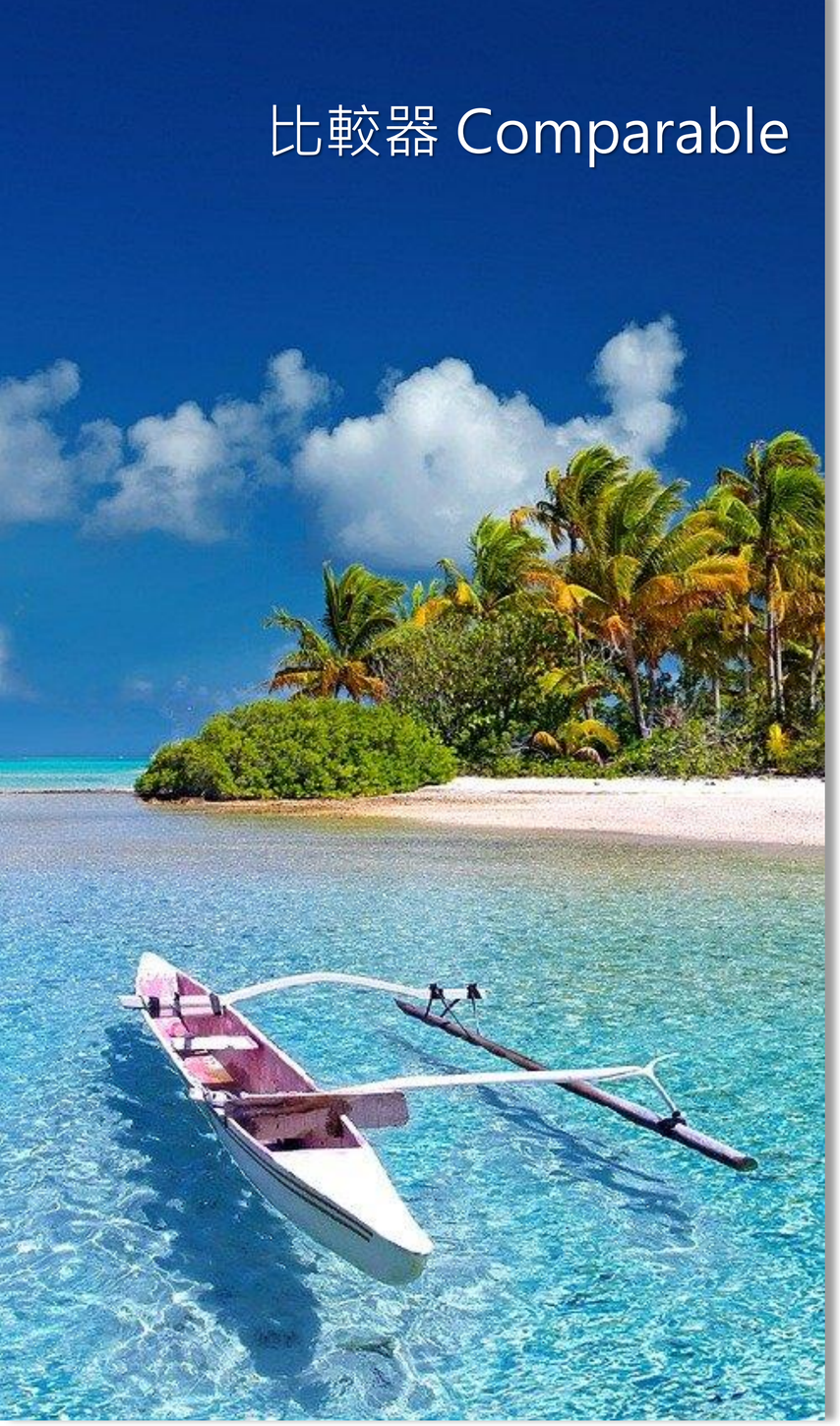
```
String color[] = {"mike", "amy", "zoe", "tom", "james"};
        //預設由小到大
        Arrays.sort(color);
        System.out.println(Arrays.toString(color));
        //利用匿名類別改寫排序準則
        Arrays.sort(color, new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                return o2.compareTo(o1);
            }
        });
        System.out.println("=====由大到小排序====");
        System.out.println(Arrays.toString(color));
```

比較器 Comparable



```
public class JavaApplication15 {
    public static void main(String[] args) {
        Person person[] = new Person[5];
        person[0] = new Person("amy", 25);
        person[1] = new Person("mike", 15);
        person[2] = new Person("zoe", 35);
        person[3] = new Person("james", 15);
        person[4] = new Person("Tina", 18);
        Arrays.sort(person);
        System.out.println(Arrays.toString(person));
    }
}
class Person implements Comparable {
    String name;
    int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Object o) {
        if (o instanceof Person) {
            Person person = (Person) o;
            if (this.age > person.age) {
                return 1;
            } else if (this.age < person.age) {
                return -1;
            } else {
                //年齡相同再針對姓名排序
                return
person.name.compareTo(this.name);
            }
        }
        throw new RuntimeException("資料型態不符合");
    }
    @Override
    public String toString() {
        return "[姓名=" + name + ", 年齡=" + age + "]";
    }
}
```

比較器 Comparable



A). new Comparator<String>() {

```
    @Override
    public int compare(String str1, String str2) {
        // TODO Auto-generated method stub
        return str1.compareTo(str2);
    }
};
```

B). new Comparator<String>() {

```
        @Override
        public int compareTo(String str1, String str2) {
            // TODO Auto-generated method stub
            return str1.compareTo(str2);
        }
    };
```

C). class Comps implements Comparator{

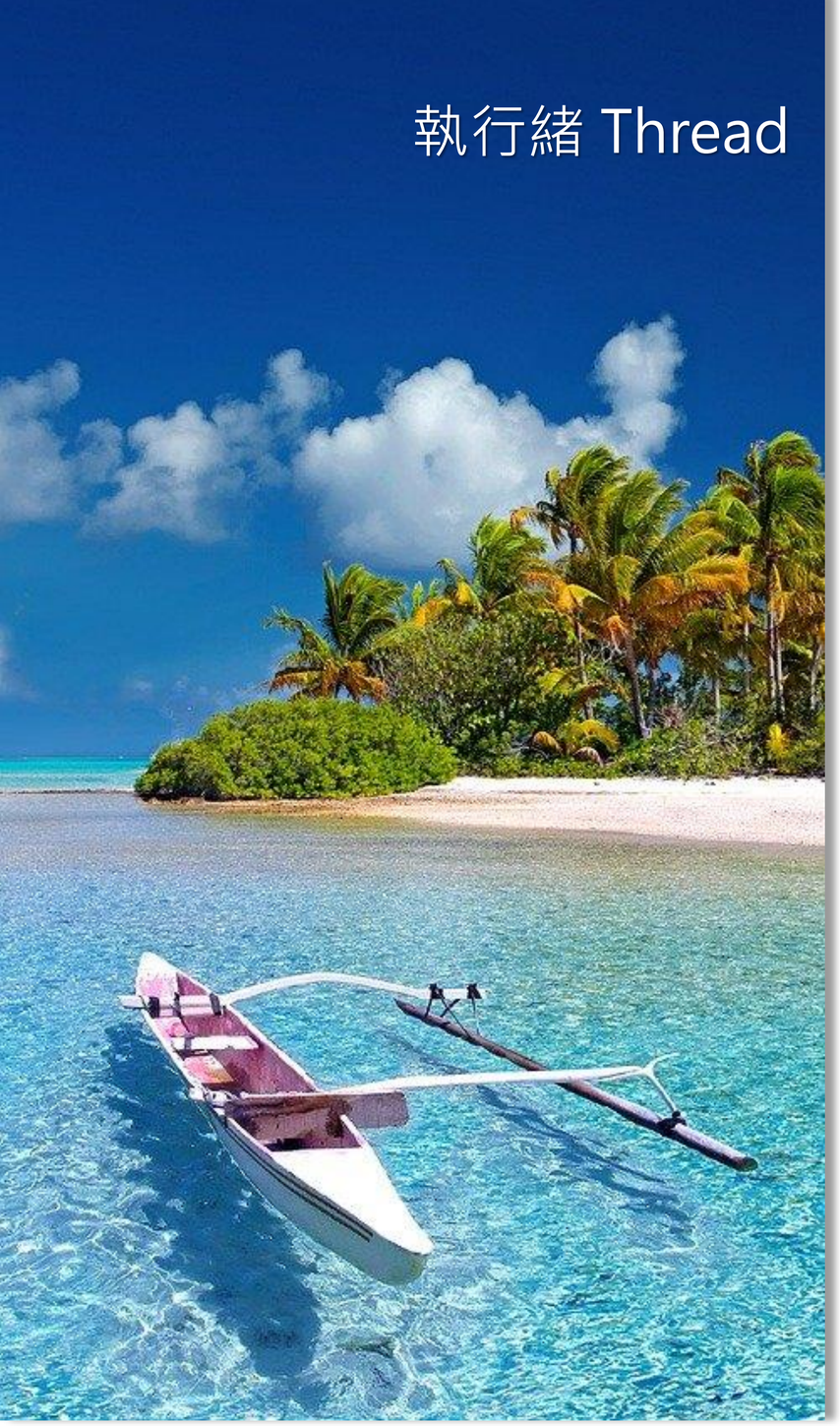
```
    @Override
    public int compare(String str1, String str2) {
        // TODO Auto-generated method stub
        return str1.length()-str2.length();
    }
}
```

D).class Comps implements Comparator{

```
    @Override
    public boolean compare(Object o1, Object o2) {
        // TODO Auto-generated method stub
        return o1.equals(o2);
    }
}
```

執行緒 THREAD

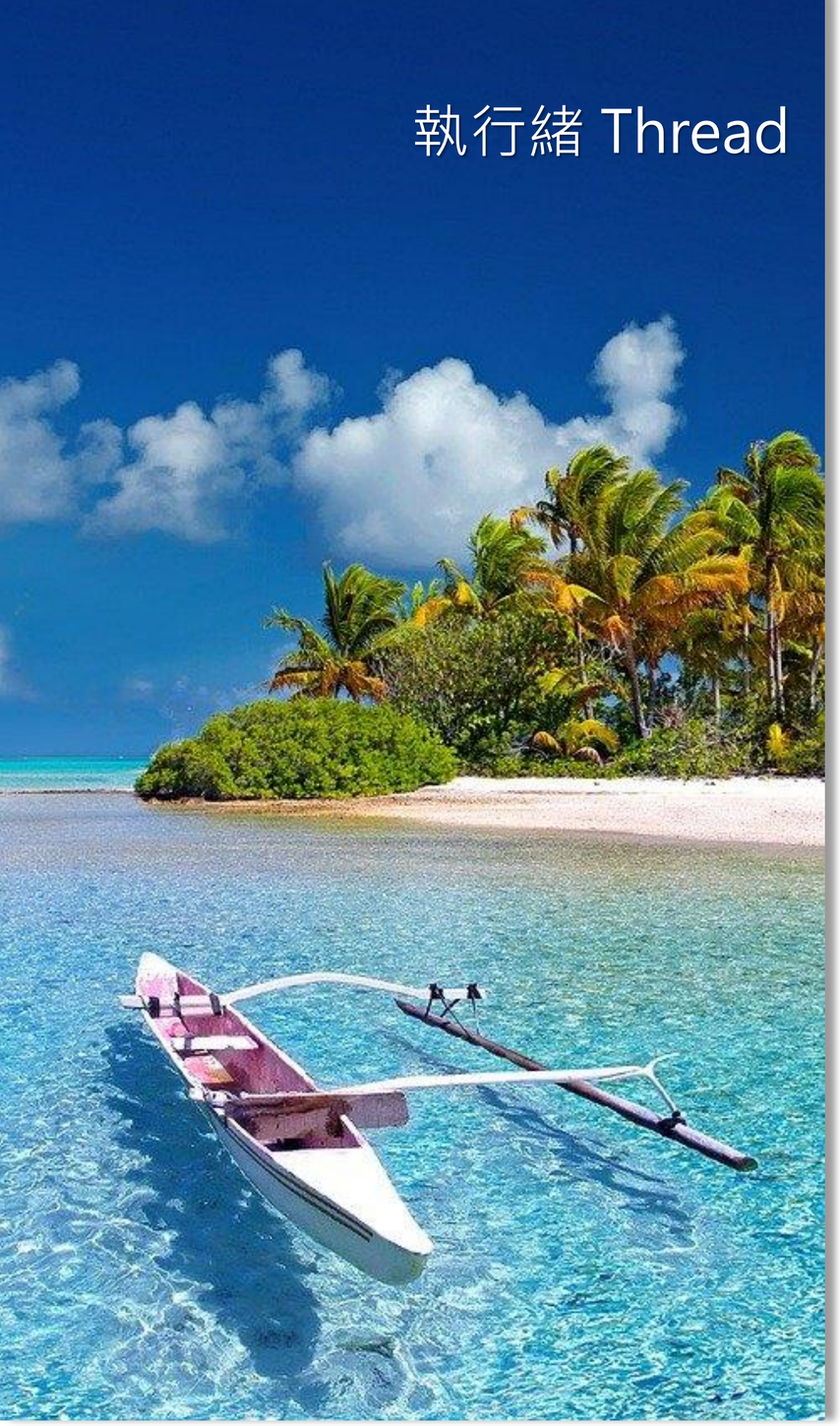
執行緒 Thread



一

- 程式(program)是一種靜態的代碼為了某特定工作而寫的一串文字,例如java程式,word
- 程序(processor)將靜態程式執行的過程,是一種動態的
- 執行緒(thread)在程序執行中可細化成多執行緒,每個執行緒擁有個自的資源(如記憶體,程式計數器),執行緒是最基本的調度和執行單位
- 執行緒的分類(一個java程序至少會有三個執行緒 main gc exception)

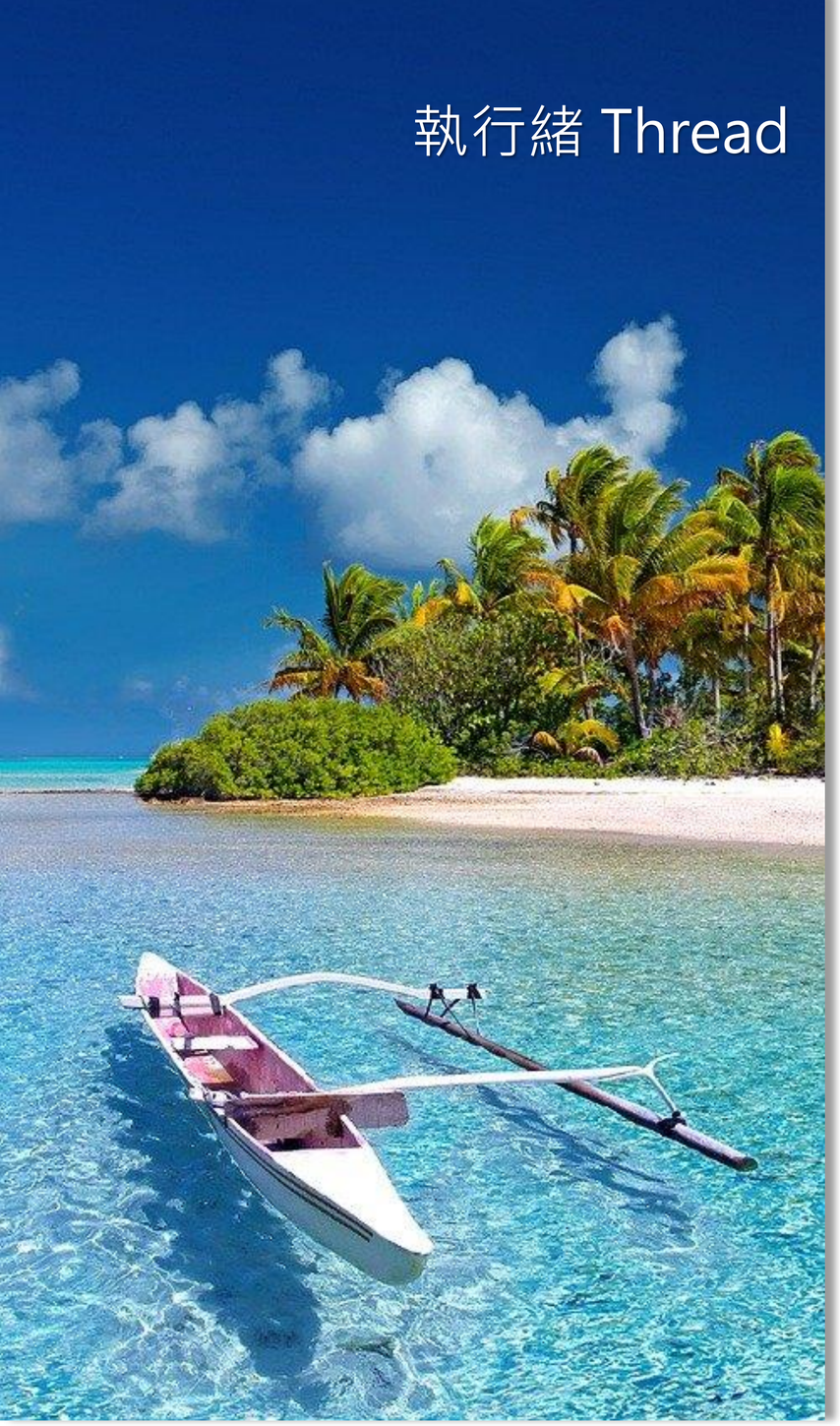
執行緒 Thread



一

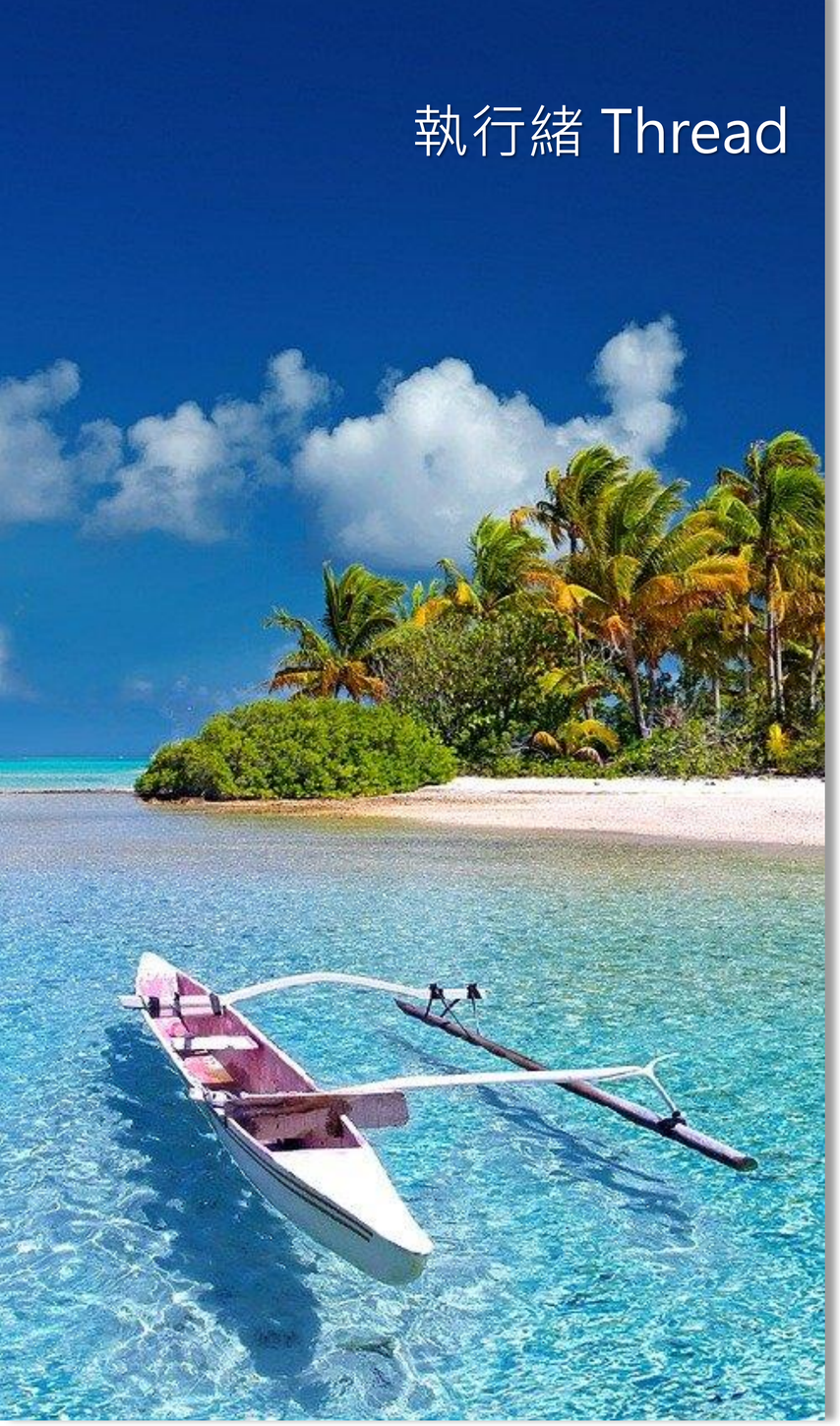
- 建立執行緒的四種方式
  - 繼承Thread類別
  - 實作Runnable的介面
- java.lang類別庫的Thread類別,定義眾多的函式,這些函式也定義了Runnable介面,而Thread類別實作了Runnable介面,我們只要繼承Thread類別,就可以設計多執行緒程式,步驟如下
  1. 將自訂類別繼承Thread類別
  2. 改寫run方法,而run內的程式就是多執行緒要執行的內容,當run執行完畢,代表該執行緒執行完畢
  3. 產生執行緒物件,可以利用建構子達成
  4. 執行執行緒物件的start()啟動該執行緒
- 不能直接呼叫run方法,因為還是單一main執行緒
- 若要有二個執行緒,需再new 一個自訂的Thread類別物件

執行緒 Thread



```
SubThread s=new SubThread();
        s.start();
         for(int m=0;m<=100;m++)
            System.out.println(Thread.currentThread.getName()+m);
class SubThread extends Thread{
    @Override
    public void run(){
       for(int m=0;m<=100;m++)
            System.out.println(Thread.currentThread().getName()+ m);
    }
}
```

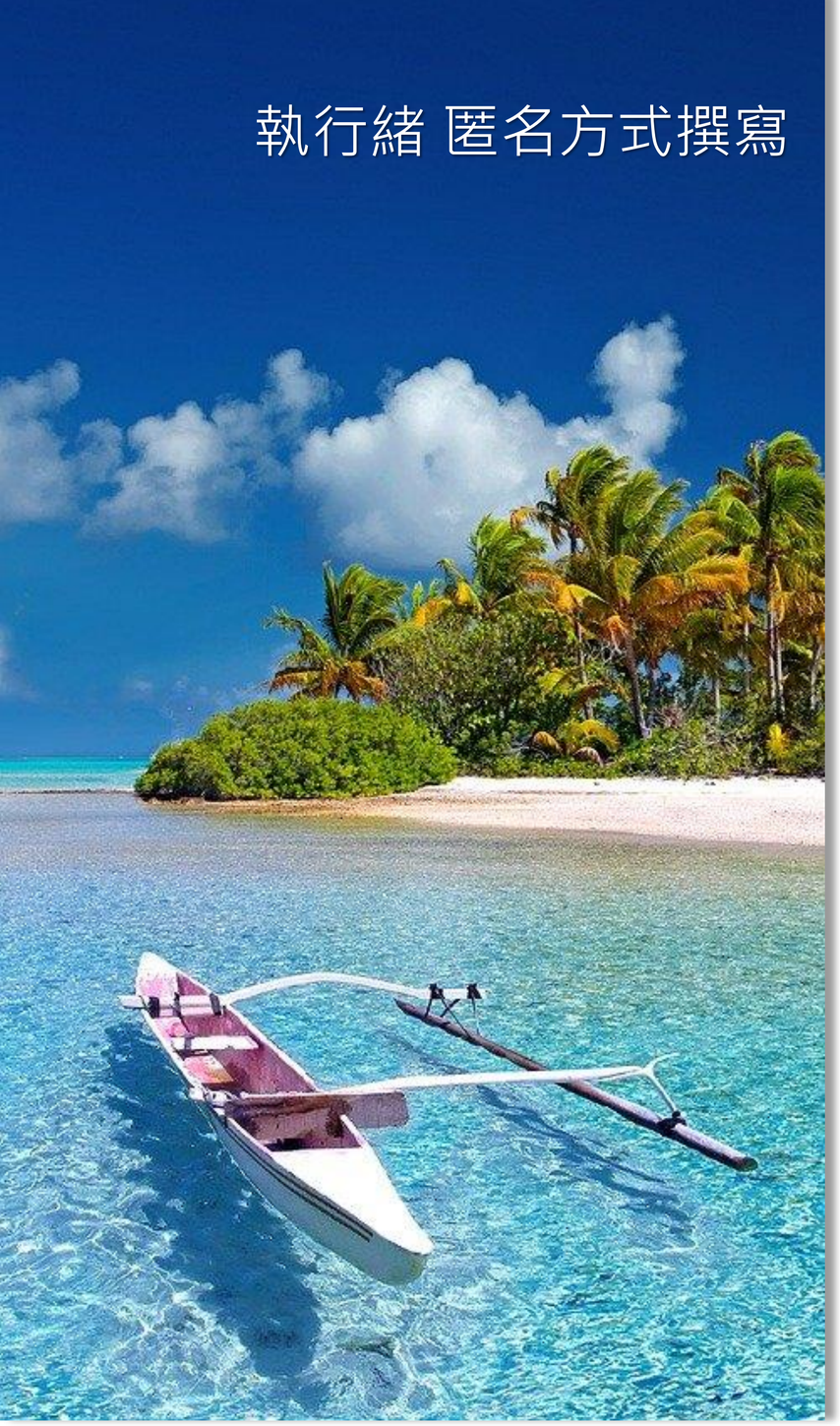
執行緒 Thread



若要二個功能不同的執行緒,只需創建二個子類,然後在二個子類覆寫run方法,並在run方法內撰寫不同功能的程式即可

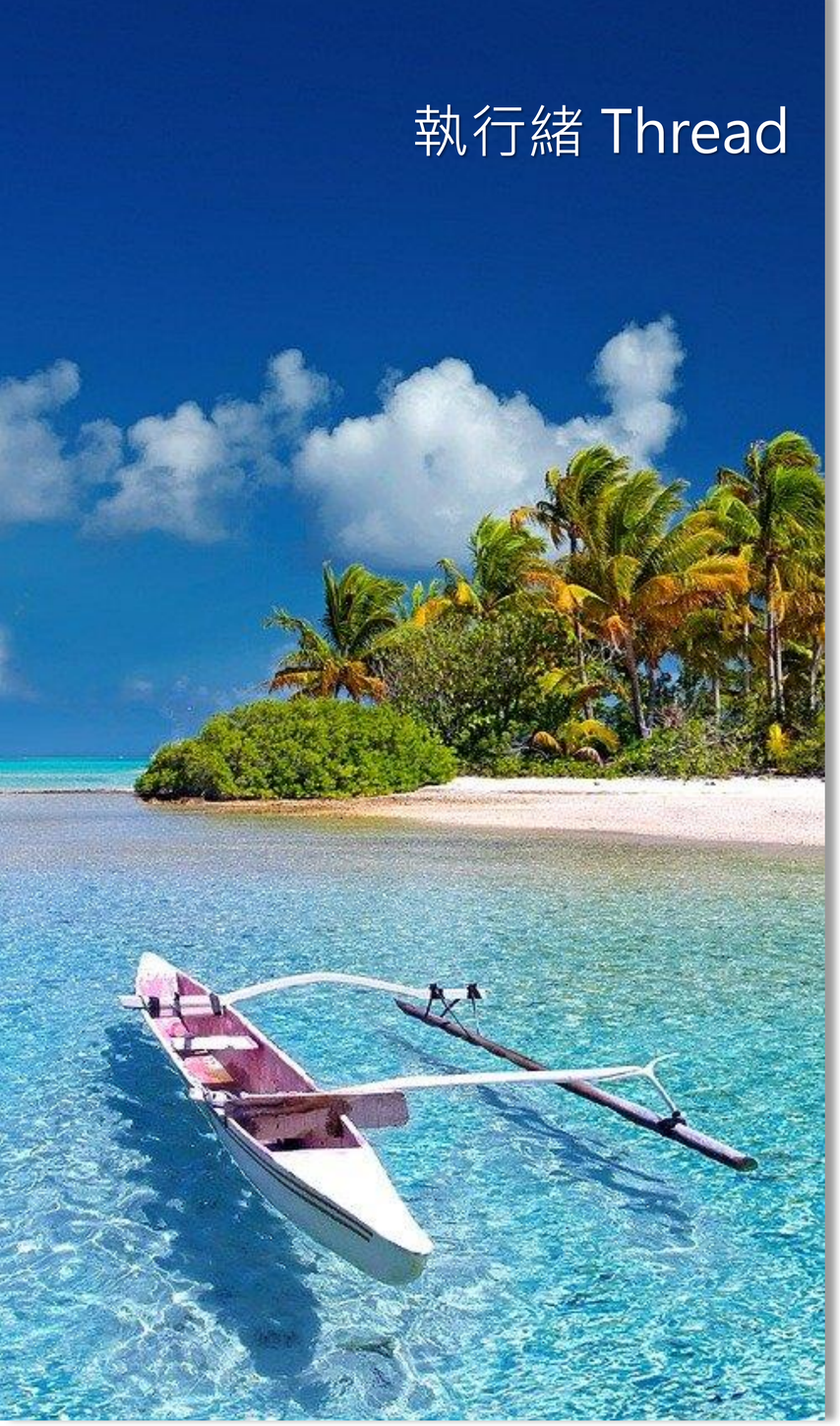
```
class subThread extends Thread{
    @Override
    public void run(){
      for(int m=0;m<=100;m++)
        if(m%2==0)

System.out.println(Thread.currentThread()
.getName()+":"+m);
    }
}
class subThread1 extends Thread{
    @Override
    public void run(){
      for(int m=0;m<=100;m++)
        if(m%2!=0)

System.out.println(Thread.currentThread()
.getName()+":"+m);
    }
}
public static void main(String[] args) {
subThread s=new subThread();
subThread1 s1=new subThread1();
s1.start();
s.start();
    }
```

執行緒 匿名方式撰寫



```
new Thread(){
        @Override
    public void run(){
      for(int m=0;m<=100;m++)
        if(m%2==0)
         System.out.println(Thread.cur
rentThread().getName()+":"+m);
    }
     }.start();
     new Thread(){
       @Override
    public void run(){
      for(int m=0;m<=100;m++)
        if(m%2!=0)
         System.out.println(Thread.cur
rentThread().getName()+":"+m);
    }
     }.start();
```

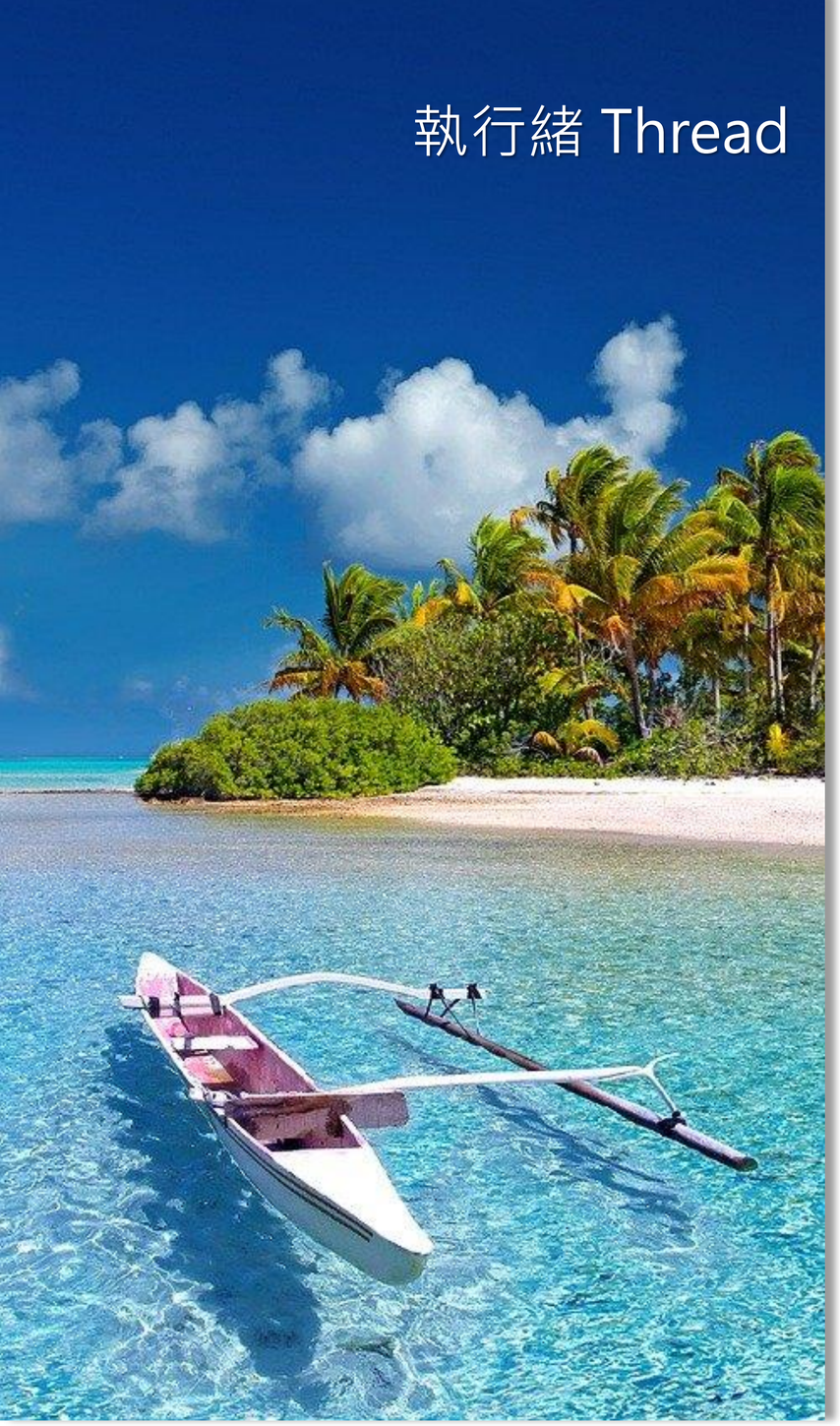
執行緒 Thread



一

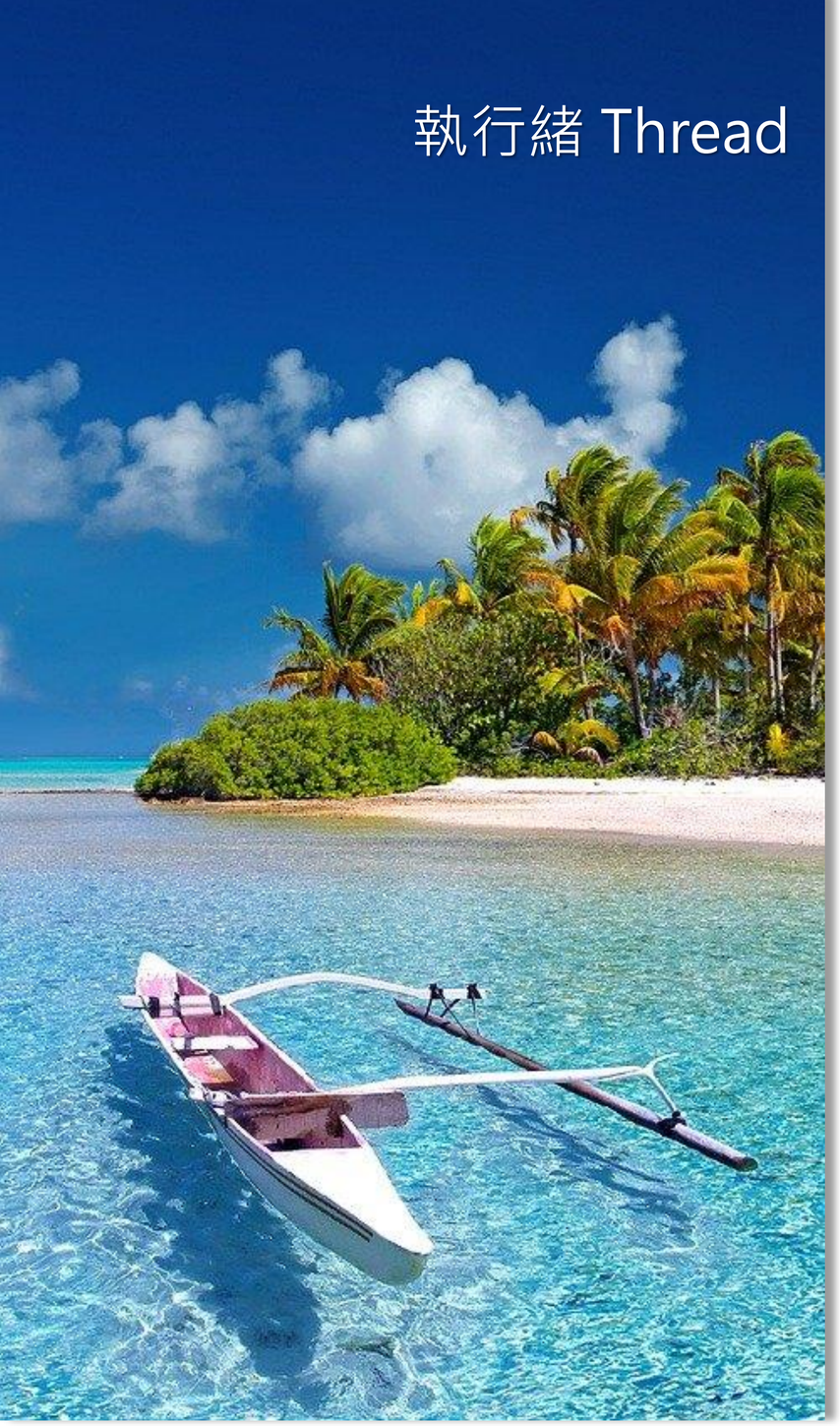
- Thread類別常用方法
  - start()啟動當前執行緒
  - run()欲執行的程式碼
  - currentThread:靜態方法,返回正在執行的執行緒代碼
  - getName():取得當前執行緒的名稱
  - setName():設定當前執行緒的名稱//如下範例
  - join()加入執行緒,即換手的意思,會等到加入的執行緒執行完後才會再執行原有的執行緒
  - sleep(1000)暫停,需try—catch 因父類無拋出例外

執行緒 Thread



```
Thread a=new Thread(){
        @Override
    public void run(){
      for(int m=0;m<=100;m++)
        if(m%2==0)
          System.out.println(Thread.currentThread().getName()+":"+m);
    }
      };
//設定主執行緒的名稱main()
      Thread.currentThread().setName("這是main主執行緒");
          System.out.println(Thread.currentThread().getName());
      a.setName("Hello");
      a.start();
//也可以在new執行緒時即設定執行緒名稱如下程式碼
      Thread b=new Thread( "第二個"){
          @Override
    public void run(){

          System.out.println(Thread.currentThread().getName()+":");
    }
      };
      b.start();
```

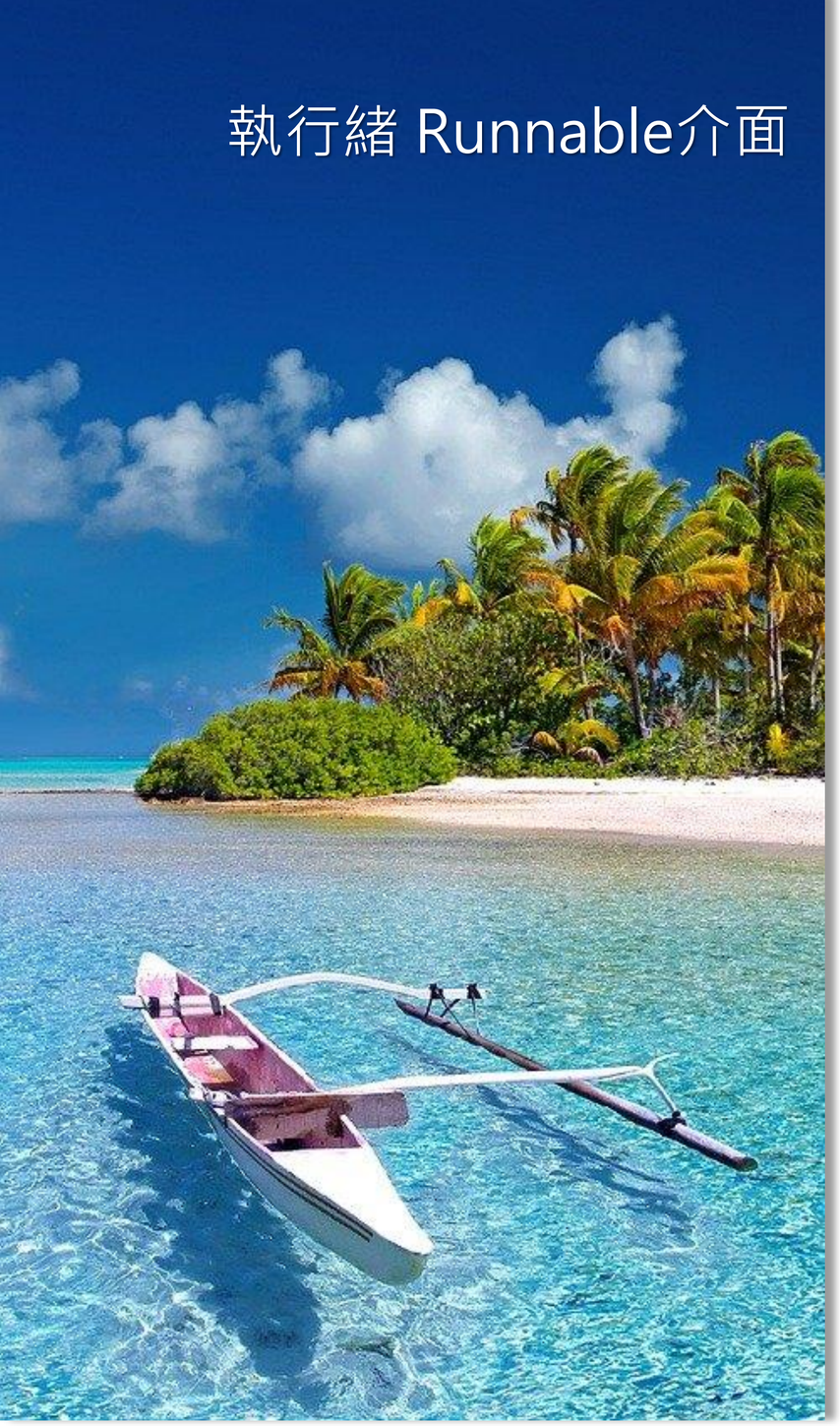
執行緒 Thread



一

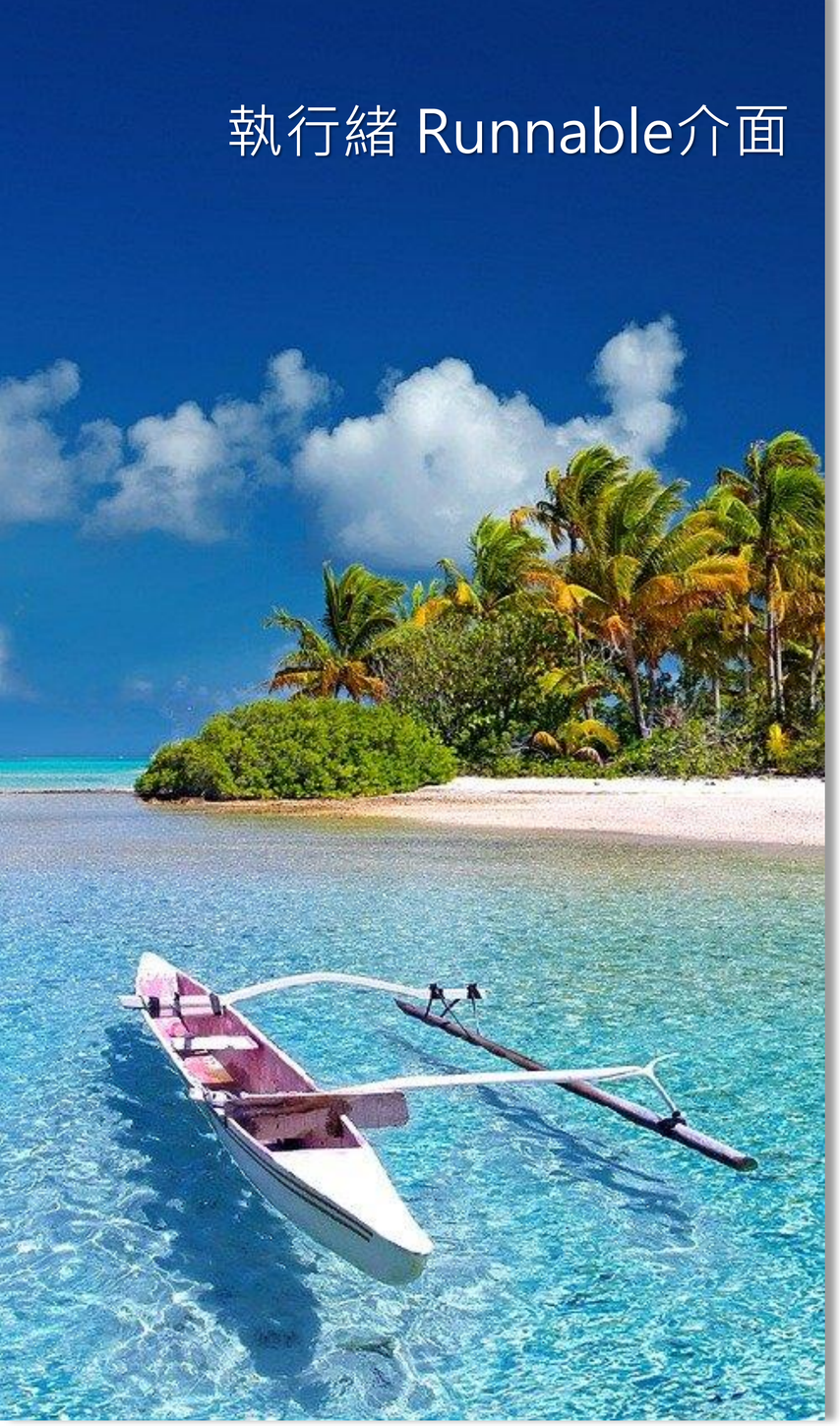
- class ThreadTest1 extends Thread{
- @Override
- public void run(){
- for(int m=0;m<=100;m++)
- {
- System.out.println(Thread.currentThread().getName()+":"+m);
- }
- }
- }
- 以下程式撰寫到主程式內
- Thread.currentThread().setName("主執行緒");
- ThreadTest1 t1=new ThreadTest1();
- t1.setName("測試一:");
- t1.start();
- for(int m=0;m<=100;m++)
- {
- System.out.println(Thread.currentThread().getName()+":"+m);
- if(m==20)
- t1.join();
- }

執行緒 Runnable介面



- 創建多執行緒方式二
  - 建立一個實作Runnable介面的類別
  - 實作Runnable介面的抽象方法run
  - 建立物件
  - 將此物件作為參數傳給Thread的建構子,以建立Thread的類別物件
  - 利用此物作去呼叫start的方法

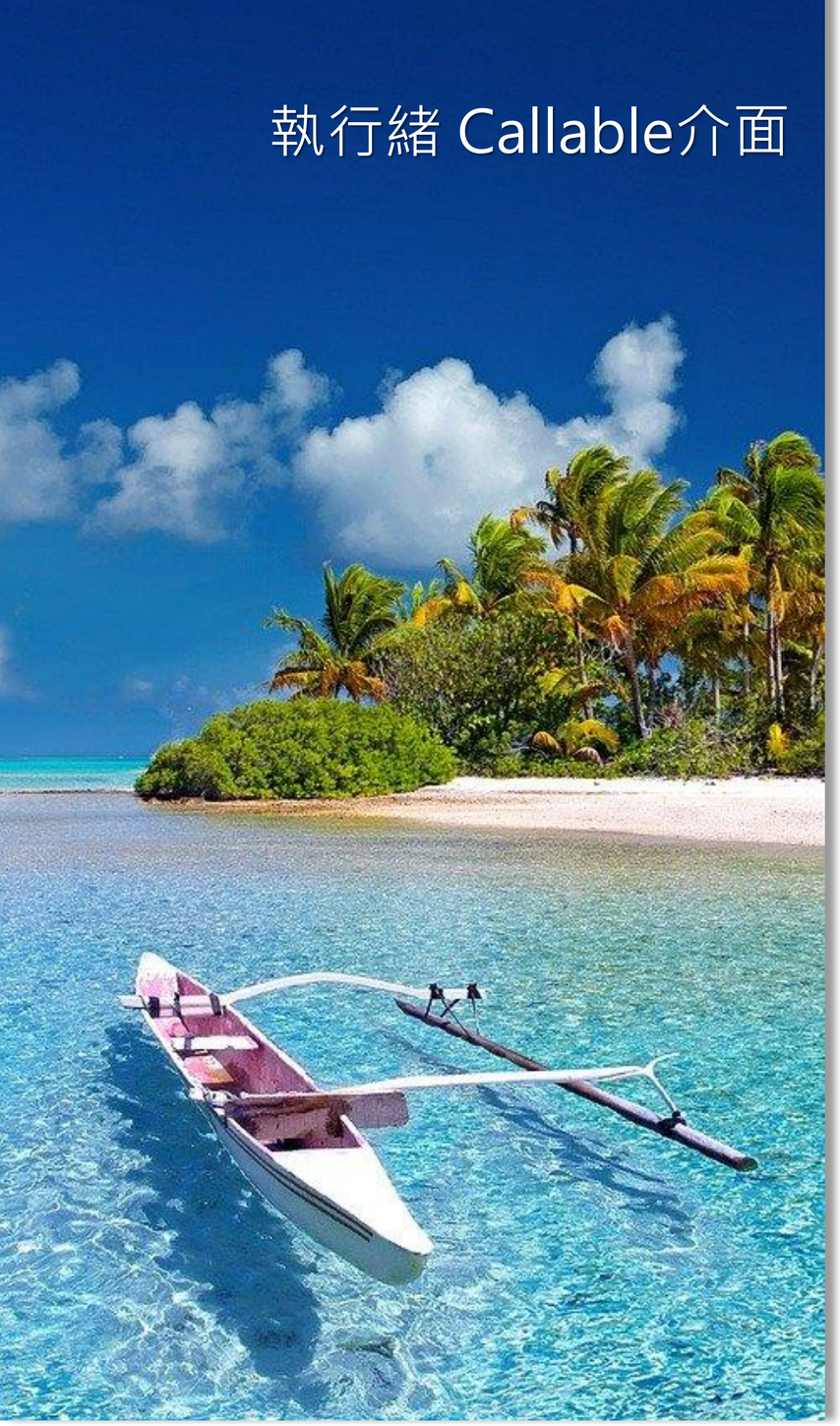
執行緒 Runnable介面



```
Runnable r=new Runnable(){
        @Override
        public void run() {
          for(int m=1;m<=10;m++)
          {

System.out.println(Thread.currentT
hread().getName()+"---m="+m);
          }
        }
    };
```

```
//以下程式撰寫在主方法內
    Thread t=new Thread(r);
    t.setName("Test");
    t.start();
```

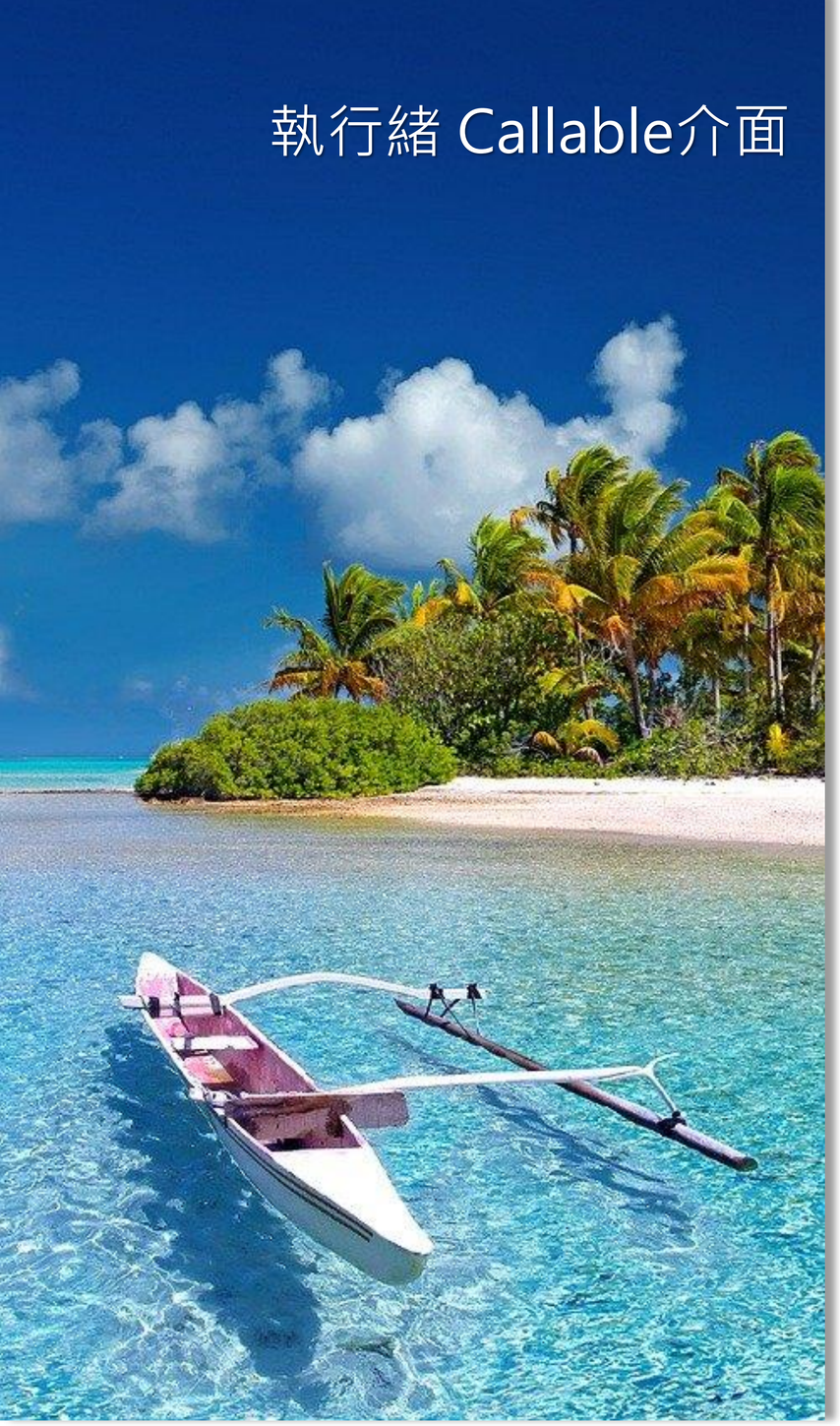
執行緒 Callable介面



一

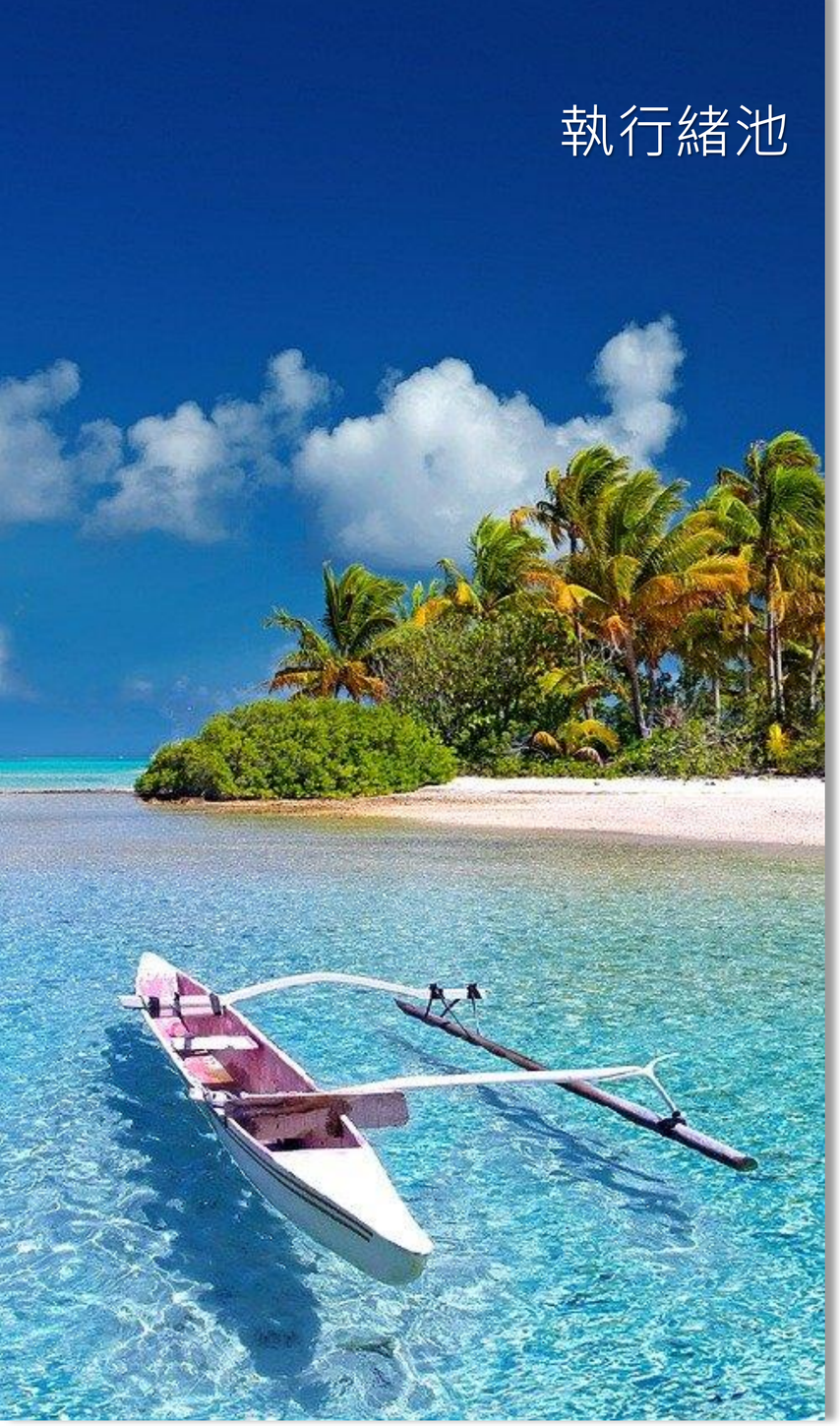
- 創建一個實現Callable的實現類
- 實作call方法,將此執行緒要做的工作撰寫於此方法中
- 創建Callable介面的實現類的物件
- 將此物件做為參數傳遞到FutureTask的建構子中以創建FutureTask類的物件
- FutherTask類別物件做為參數放到Thread類的建構子
- 啟動Thread類　start()

執行緒 Callable介面

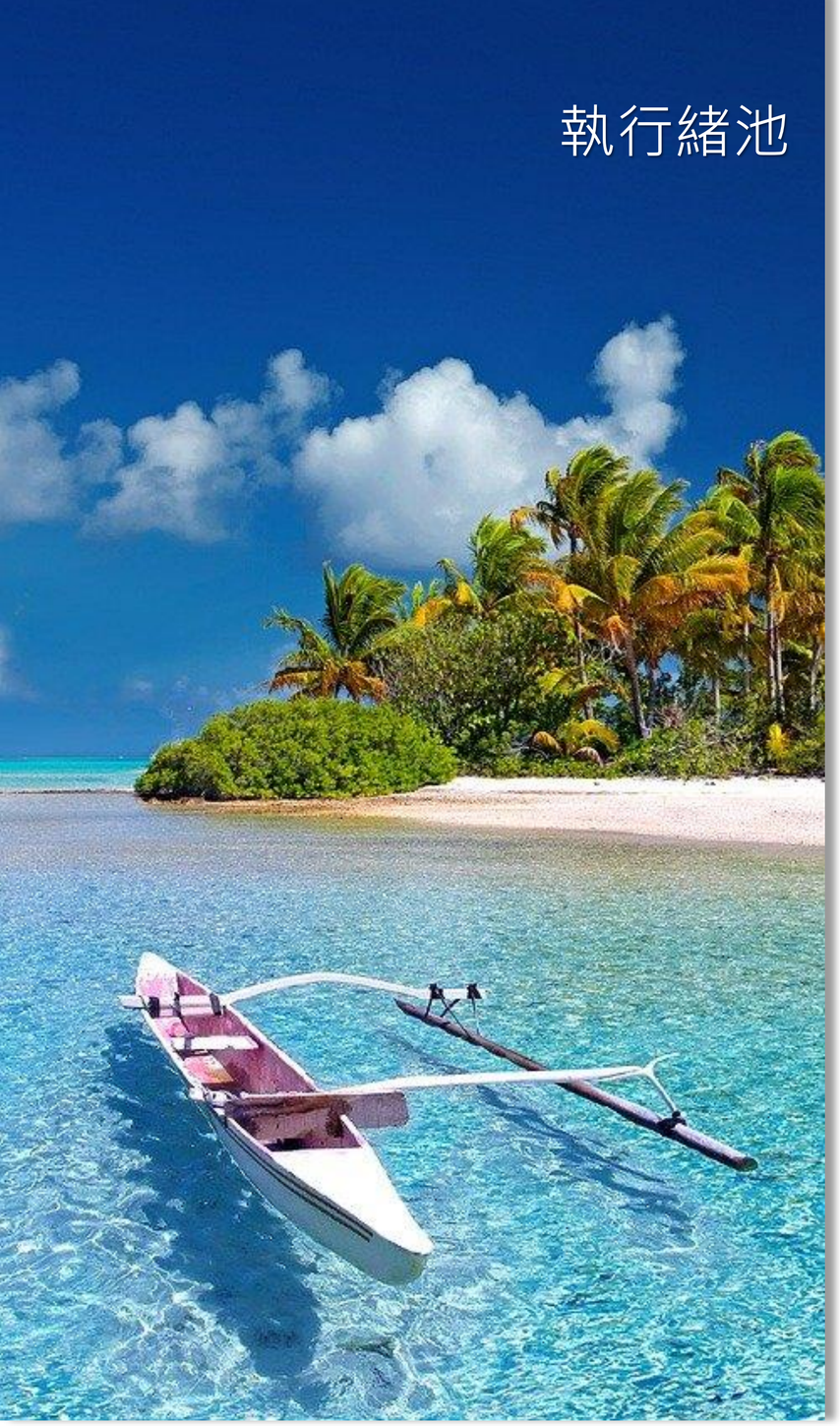


```
 public class ThreadTest1 {
public static void main(String[] args) throws InterruptedException, ExecutionException {
//3.創建Callable介面的實現類的物件
CallTest test=new CallTest();
//4.將此物件做為參數傳遞到FutureTask的建構子中以創建FutureTask類的物件
FutureTask task=new FutureTask(test);
//5.FutherTask類別物件做為參數放到Thread類的建構子
Thread t1=new Thread(task,"我的執行緒");
t1.start();
System.out.println(task.get());//獲取返回值

        }
}
//1.創建一個實現Callable的實現類
class CallTest implements Callable<Integer>{
        @Override
//2.實作call方法,將此執行緒要做的工作撰寫於此方法中
public Integer call() throws Exception {
int sum=0;
for (int i = 0; i <=100; i++) {System.out.println(Thread.currentThread().getName()+":"+i);
sum+=i;
}
return sum;
}
}
```

執行緒池



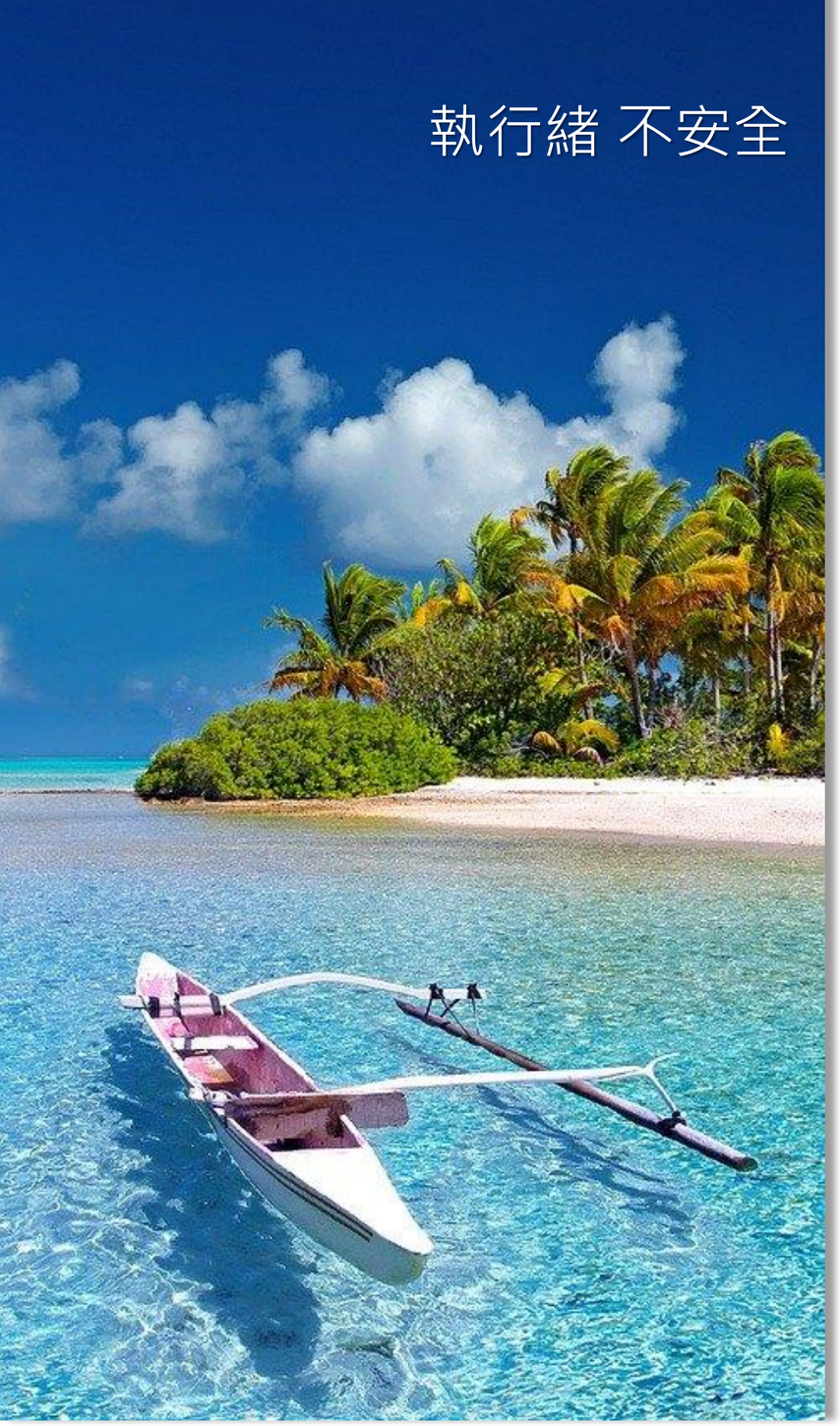
- **提前創建好多個執行緒,放入執行緒池中,使用時直接獲取,使用完後放回池中,可以避免頻繁創建銷毀,實現重覆利用.**
- ExecutorService 真正的執行緒池介面,常見子類 ThreadPoolExecutor
  - void execute(Runnable command) 沒有返回值 一般用來執行Runnable
  - <T>Future<T> submit(Callable<T> task)有返回值 一般用來執行Callable
  - void shutdown() 關閉連接池
- Executors 工具類 用於創建並返回不同類型的執行緒池
  - Excutors.newCacheThreadPool()
  - Excutors.newFixedThreadPool(n)
  - Excutors.newSingleThreadExecutor()
  - Executors.newScheduledThreadPool(n)



執行緒池

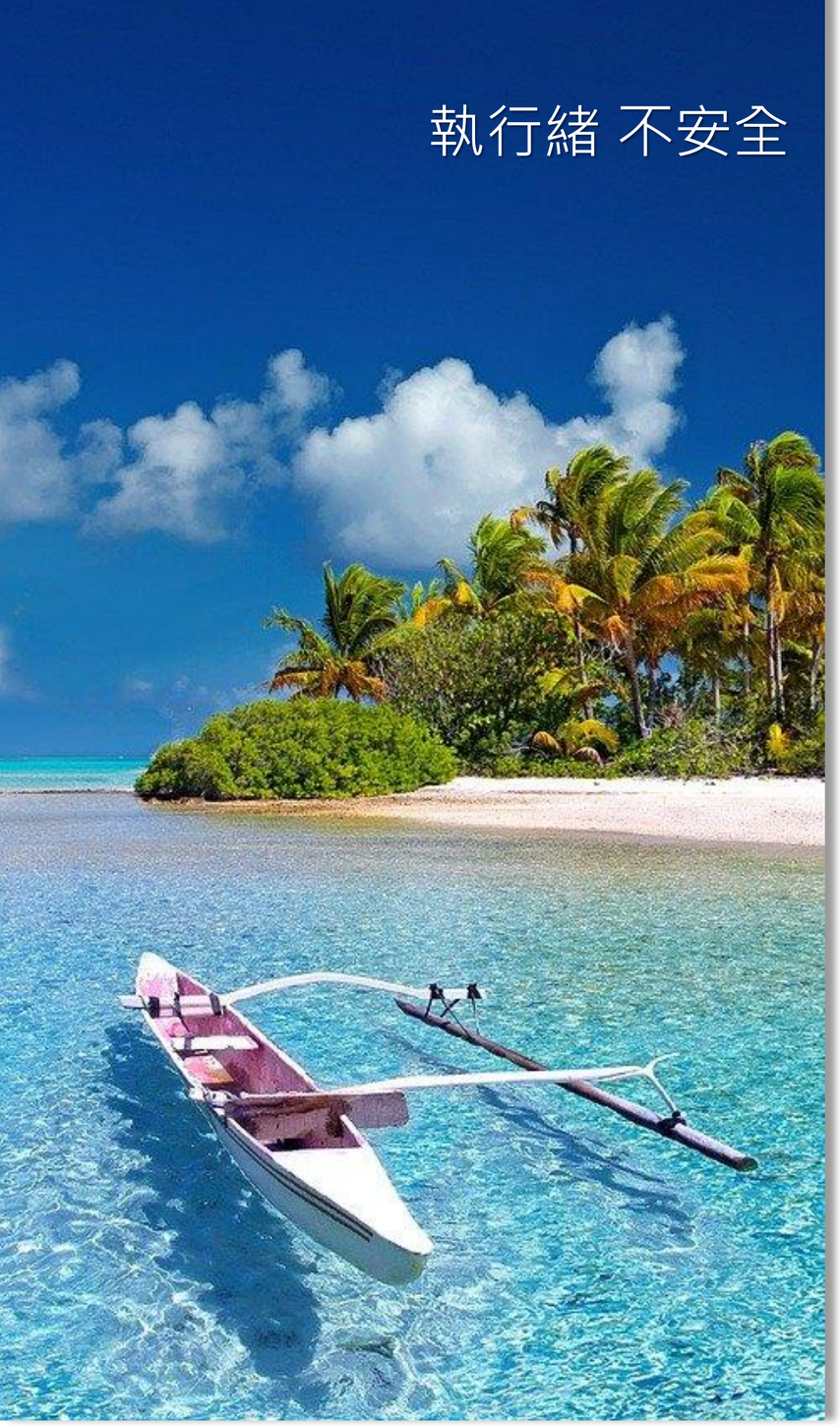
```
class ThreadPool implements Runnable{
@Override
public void run() {
for (int i = 0; i<=50; i++) {
System.out.println(Thread.currentThread().getName()+":"+i);
}

}
}
class ThreadPool2 implements Runnable{
        @Override
public void run() {
for (int i = 51; i<=100; i++) {
System.out.println(Thread.currentThread().getName()+":"+i);
}
}
}
public class ThreadTest2 {
public static void main(String[] args) throws InterruptedException, ExecutionException {
ExecutorService service = Executors.newFixedThreadPool(5);
service.execute(new ThreadPool());
service.execute(new ThreadPool2());
service.submit(new CallTest1());
service.shutdown();//執行完一定要關閉執行緒池,否則仍一直執行
        }
}
//1.創建一個實現Callable的實現類
class CallTest1 implements Callable<Integer>{
@Override
//2.實作call方法,將此執行緒要做的工作撰寫於此方法中
public Integer call() throws Exception {
int sum=0;
for (int i = 100; i <=150; i++) {System.out.println(Thread.currentThread().getName()+":"+i);
        sum+=i;
}
                return sum;
        }
}
```

執行緒 不安全



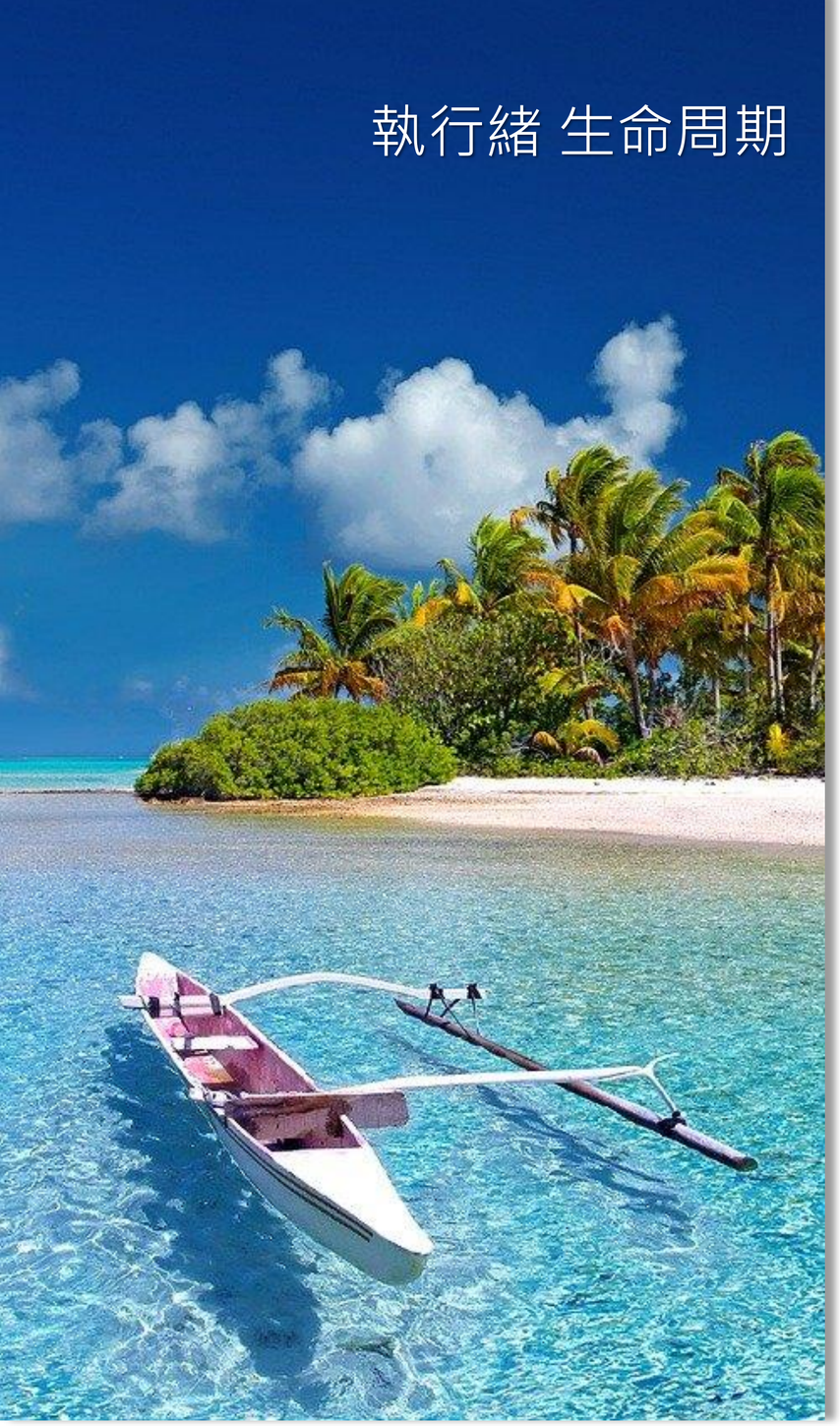
```
 public class ThreaSafe {
public static void main(String[] args) {
Sells w1=new Sells("窗口1");
Sells w2=new Sells("窗口2");
Sells w3=new Sells("窗口3");
w1.start();
w2.start();
w3.start();
}
}
class Sells extends Thread{
//private int ticket=100;此為每個窗口皆有100張票賣
private static int ticket=100;
public Sells(String msg) {
super(msg);
}
@Override
public void run() {
while(true){
if(ticket>0)
{
System.out.println(getName()+"票號:"+ticket);
ticket--;
}
else break;
}
}
}
```

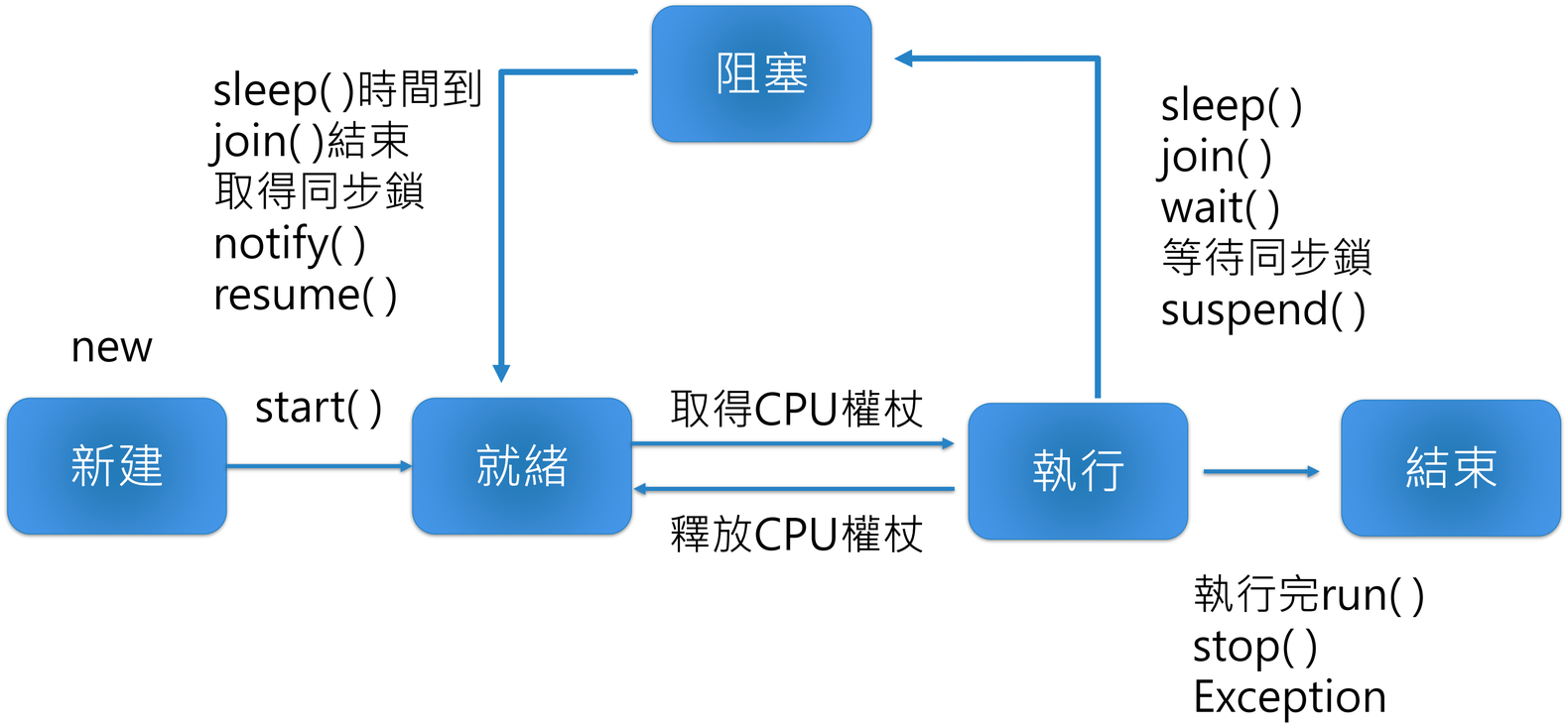
執行緒 不安全



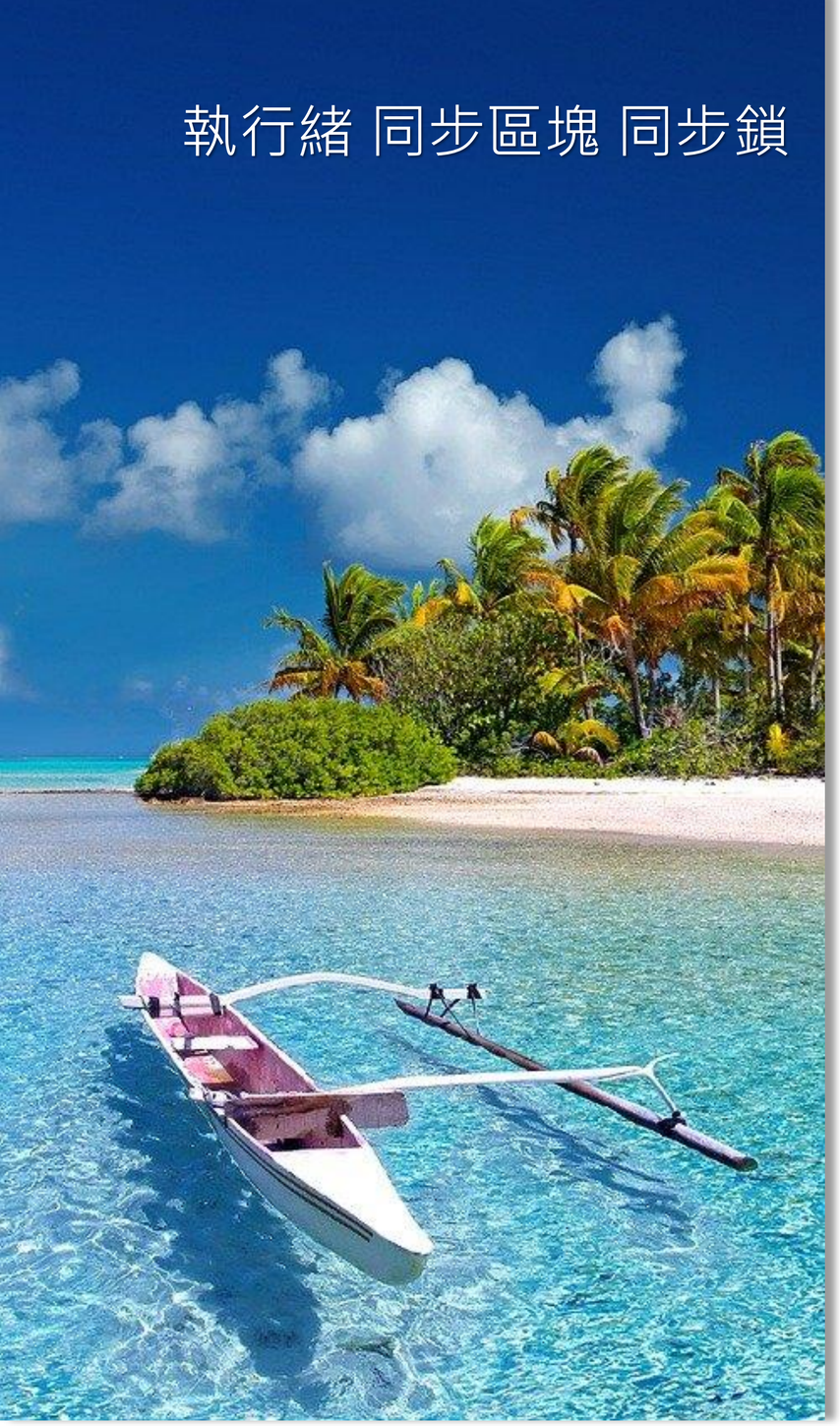
- Runnable r=new Runnable() {
- private int ticket=100;
- @Override
- public void run() {
- while(ticket>0) {
- System.out.println(Thread.currentThread().getName()+"-票號:"+ticket);
- ticket--;
- }
- }
- };
- Thread t1=new Thread(r,"窗口1");
- Thread t2=new Thread(r,"窗口2");
- Thread t3=new Thread(r,"窗口3");
- t1.start();
- t2.start();
- t3.start();

執行緒 生命周期



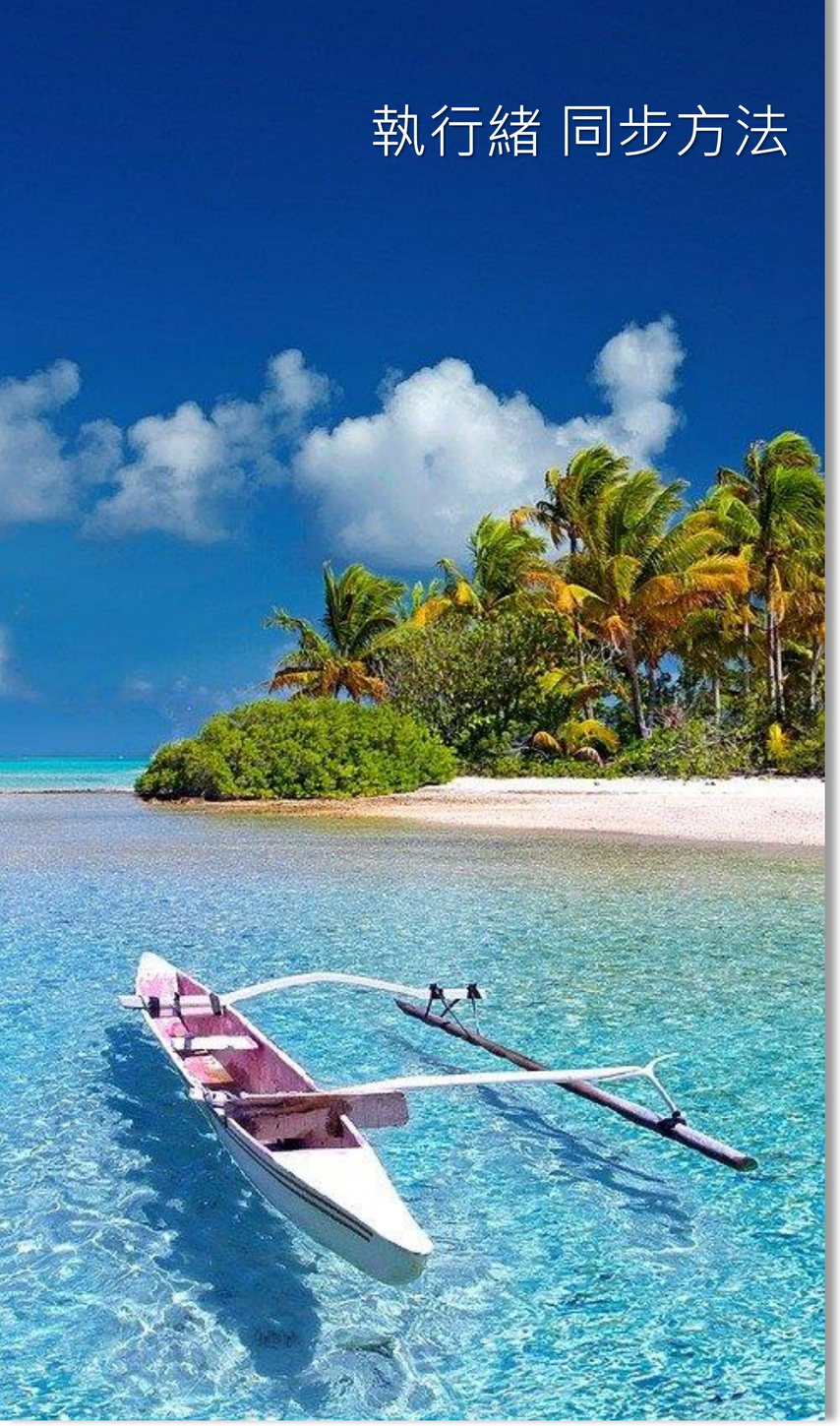

阻塞
sleep( )時間到
join( )結束
取得同步鎖
notify( )
resume( )
sleep( )
join( )
wait( )
等待同步鎖
suspend( )
new
start( )
新建
就緒
取得CPU權杖
釋放CPU權杖
執行
結束
執行完run( )
stop( )
Exception

執行緒 同步區塊 同步鎖



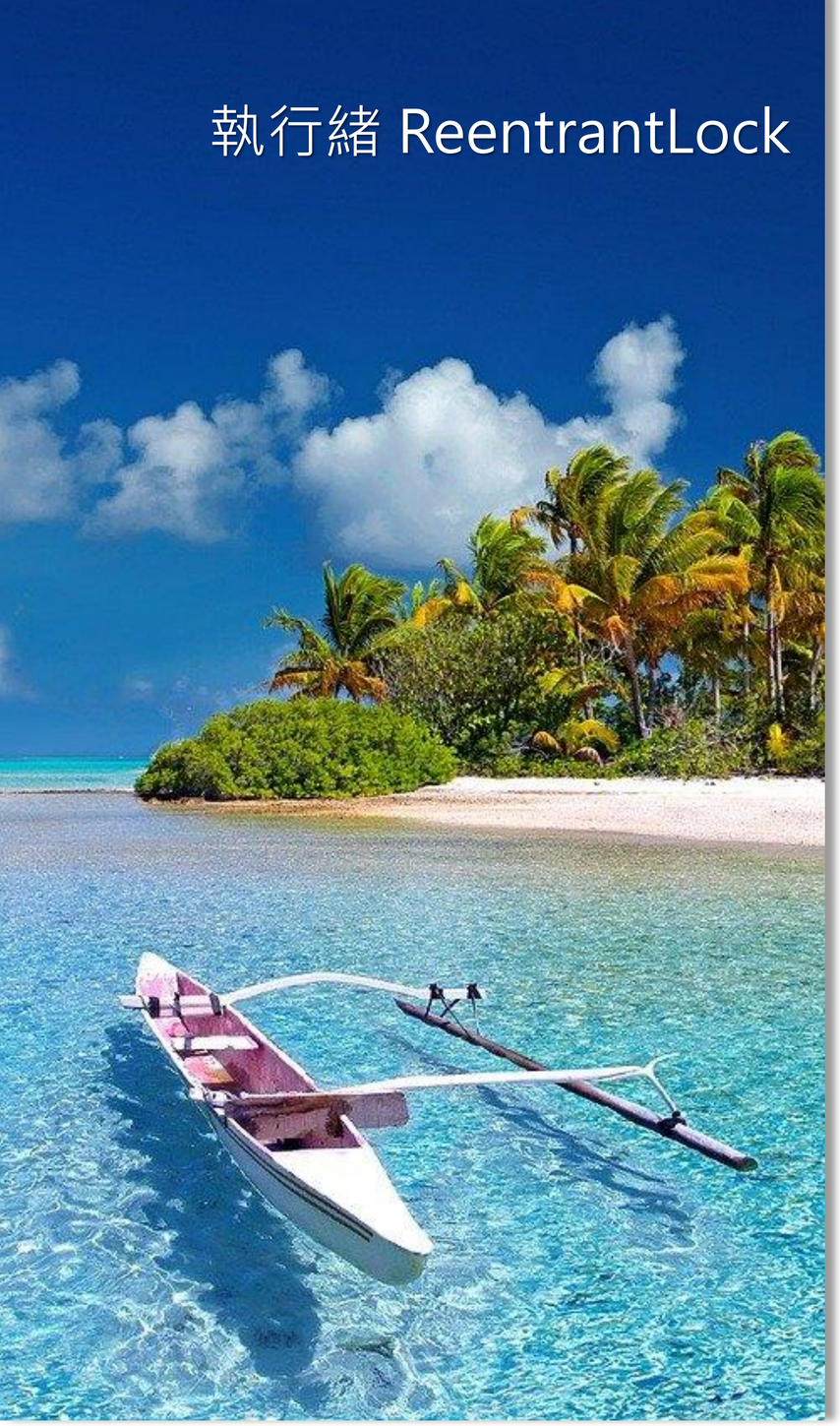
```
public class SellsTicket {
public static void main(String[] args) {
Sells s1=new Sells("窗口1");
Sells s2=new Sells("窗口2");
Sells s3=new Sells("窗口3");
s1.start();
s2.start();
s3.start();
}
}
class Sells extends Thread{
Sells(String name) {
super(name);
}
private static int ticket=100;//總票數只有100張共誰賣比較多
String lock="ok";
public void run() {
while(true)
{
synchronized(lock) {
if(ticket>0) {
System.out.println(getName()+"票數:"+ticket);
ticket--;
}
else
break;
}
}
}
}
```

執行緒 同步方法



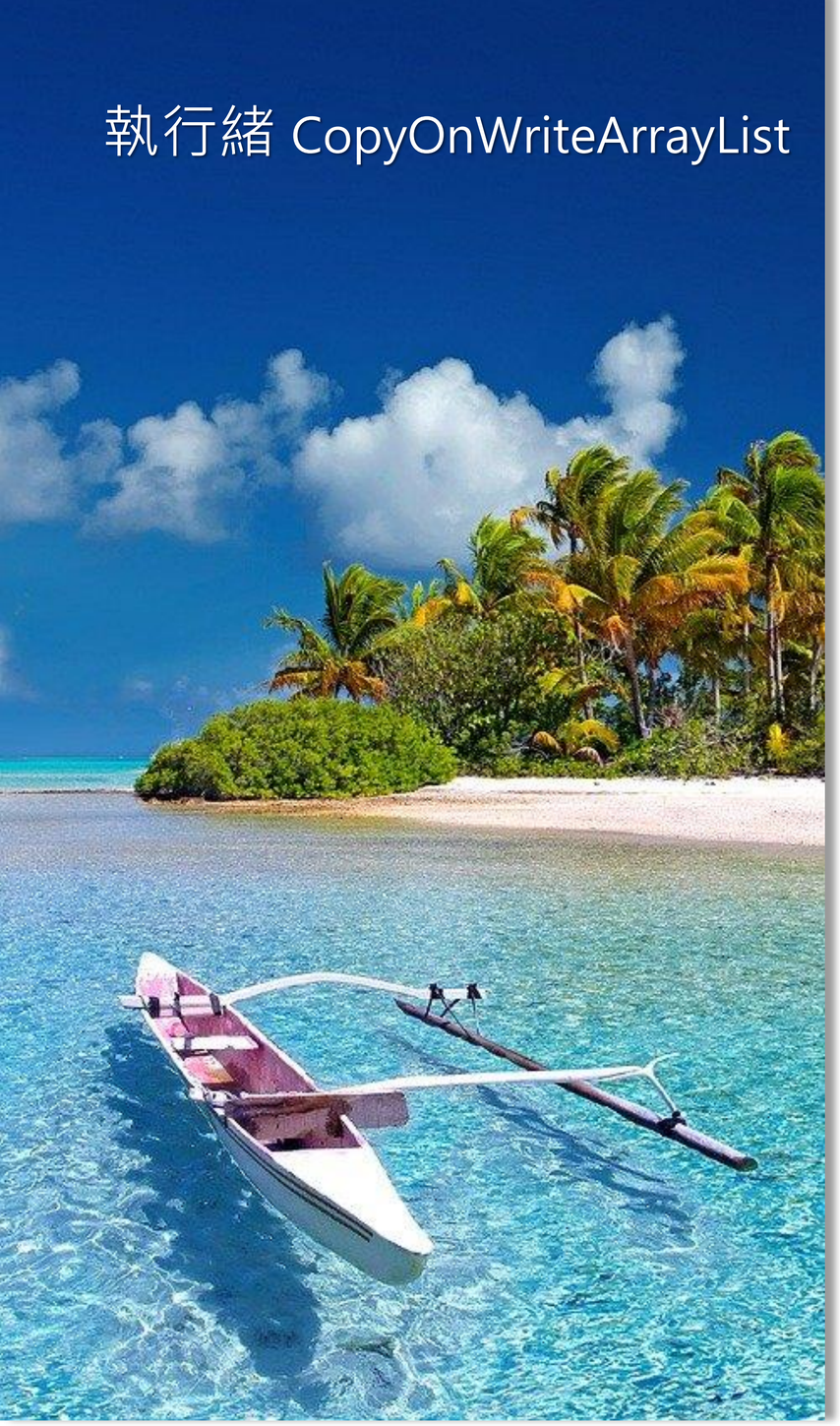
```
public class SellsTicket2 {

public static void main(String[] args) {
Sells2 s1=new Sells2("窗口1");
Sells2 s2=new Sells2("窗口2");
Sells2 s3=new Sells2("窗口3");
s1.start();
s2.start();
s3.start();

}

}
class Sells2 extends Thread{
Sells2(String name) {
super(name);
}

private static int ticket=100;//總票數只有100張共誰賣比較多

public void run() {
while(true)
{
showTicket();
}

}
private synchronized static void showTicket() {//不能使用物件方法,因為this會有三個物件,只能使用static,即為Sells2.class本身,也只有一個
if(ticket>0) {
System.out.println(Thread.currentThread().getName()+"票數:"+ticket);
ticket--;
}

}
}
```

執行緒 ReentrantLock



```
import java.util.concurrent.locks.ReentrantLock;
public class ThreadTest1 {
public static void main(String[] args) {
MyThread t1 = new MyThread("售票窗口一");
MyThread t2 = new MyThread("售票窗口二");
MyThread t3 = new MyThread("售票窗口三");
t1.start();
t2.start();
t3.start();
}
}
class MyThread extends Thread {
static ReentrantLock lock = new ReentrantLock();
MyThread(String msg) {
                super(msg);
}
static int ticket = 100;
@Override
public void run() {
while (true) {
try {
lock.lock();//上鎖
if (ticket > 0) {
try {

sleep(50);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println(currentThread().getName() + ":" + ticket);
ticket--;
}
} finally {lock.unlock();//解鎖
if (ticket == 0)
break;
}
}
}
}
```

執行緒 CopyOnWriteArrayList

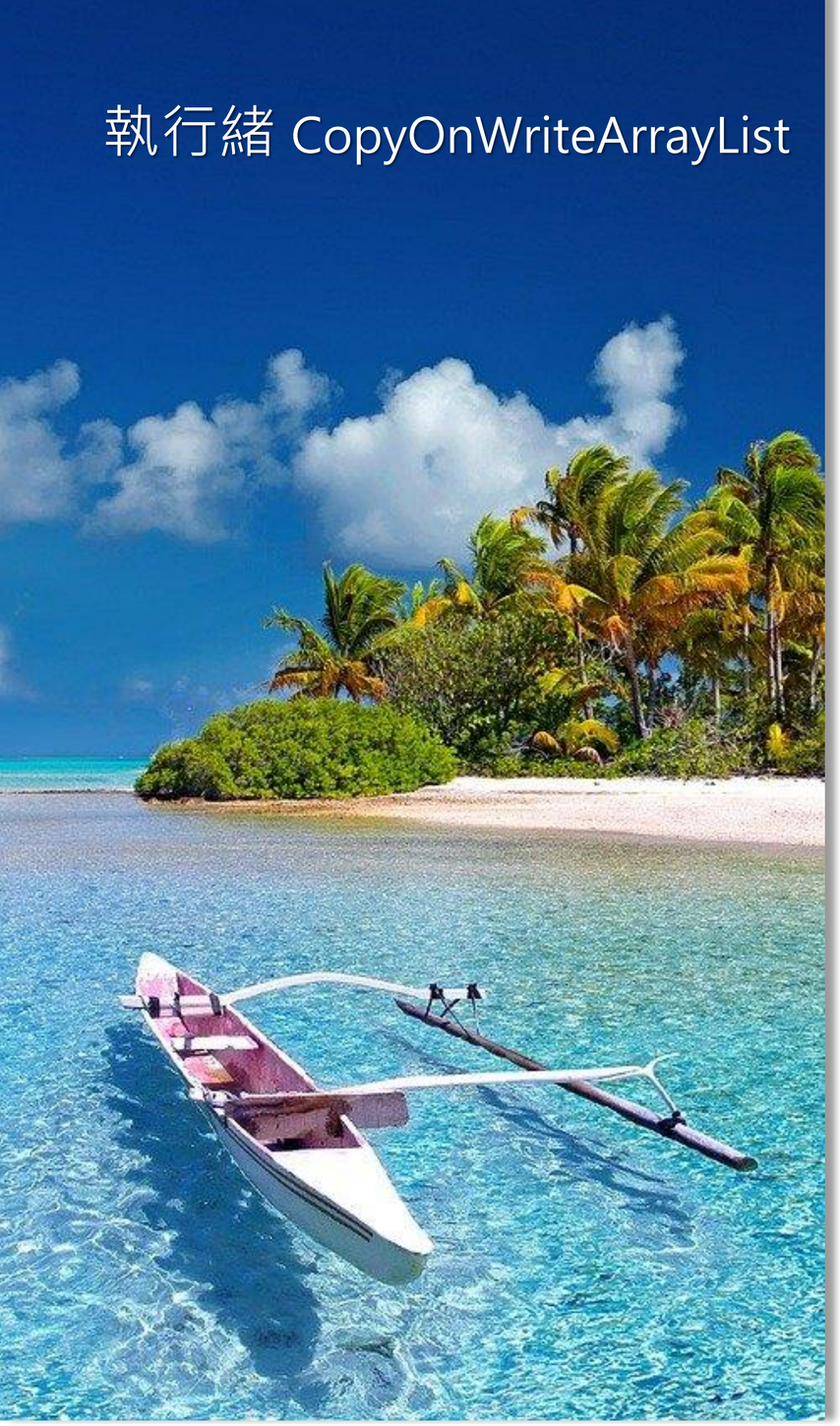


```
class ThreadExample implements Runnable {
                private List<String> list;
                public ThreadExample(List<String> list) {
                  this.list = list;
                }
                @Override
                public void run() {
list.set(2, "hello");
                  try {
                    for (int i = 0; i < 10; i++) {
                      list.add("b" + i);
                    }
                  } catch(Exception e) {
                    throw new RuntimeException(e);
                  }
                }
System.out.println(list);
                }
```

```
public class Main {
public static void main(String[] args) {
  try {
    List<String> list = new ArrayList<String>();
    list.add("a1");
    list.add("a2");
    list.add("a3");
    list.add("a4");
    list.add("a5");
    list.add("a6");
    Thread thread = new Thread(new ThreadExample(list));
    thread.start();
    for(String s: list) {
      System.out.println(s);
    }
  } catch(Exception e) {
    throw new RuntimeException(e);
  }
}
}
```

```
public class Main {
 public static void main(String[] args) {
 try {
    final CopyOnWriteArrayList<String> list = new
CopyOnWriteArrayList<String>();
    list.add("a1");
    list.add("a2");
    list.add("a3");
    list.add("a4");
    list.add("a5");
    list.add("a6");
    Thread thread = new Thread(new ThreadExample(list));
    thread.start();
    for(String s: list) {
      System.out.println(s);
    }
  } catch(Exception e) {
    throw new RuntimeException(e);
  }
System.out.println(list);
 }
}
```

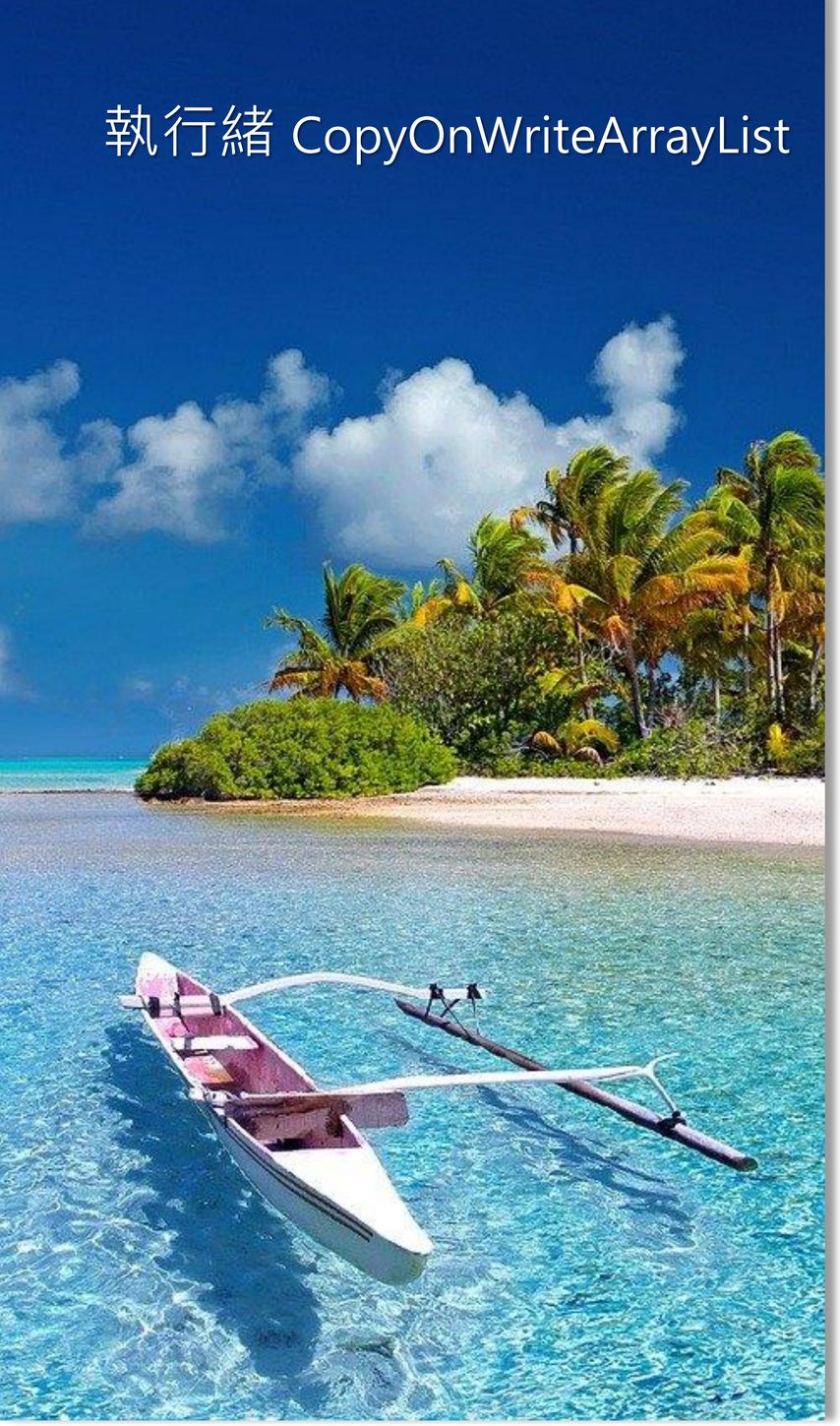
執行緒 CopyOnWriteArrayList



```
var c=new CopyOnWriteArrayList<>(List.of("1","2","3","4"));
Runnable r=()->{
    try {
        Thread.sleep(150);
    }catch(InterruptedException e) {
        System.out.println(e);
    }
    c.set(3, "four");
    System.out.print(c+" ");
};
Thread t=new Thread(r);
t.start();
for(var s:c) {
    System.out.print(s+ " ");
    Thread.sleep(100);
}
```

- ○ A. 1 2 [1, 2, 3, four] 3 four
- ○ B. 1 2 [1, 2, 3, 4] 3 four
- ○ C. 1 2 [1, 2, 3, four] 3 4
- ○ D. 1 2 [1, 2, 3, 4] 3 4

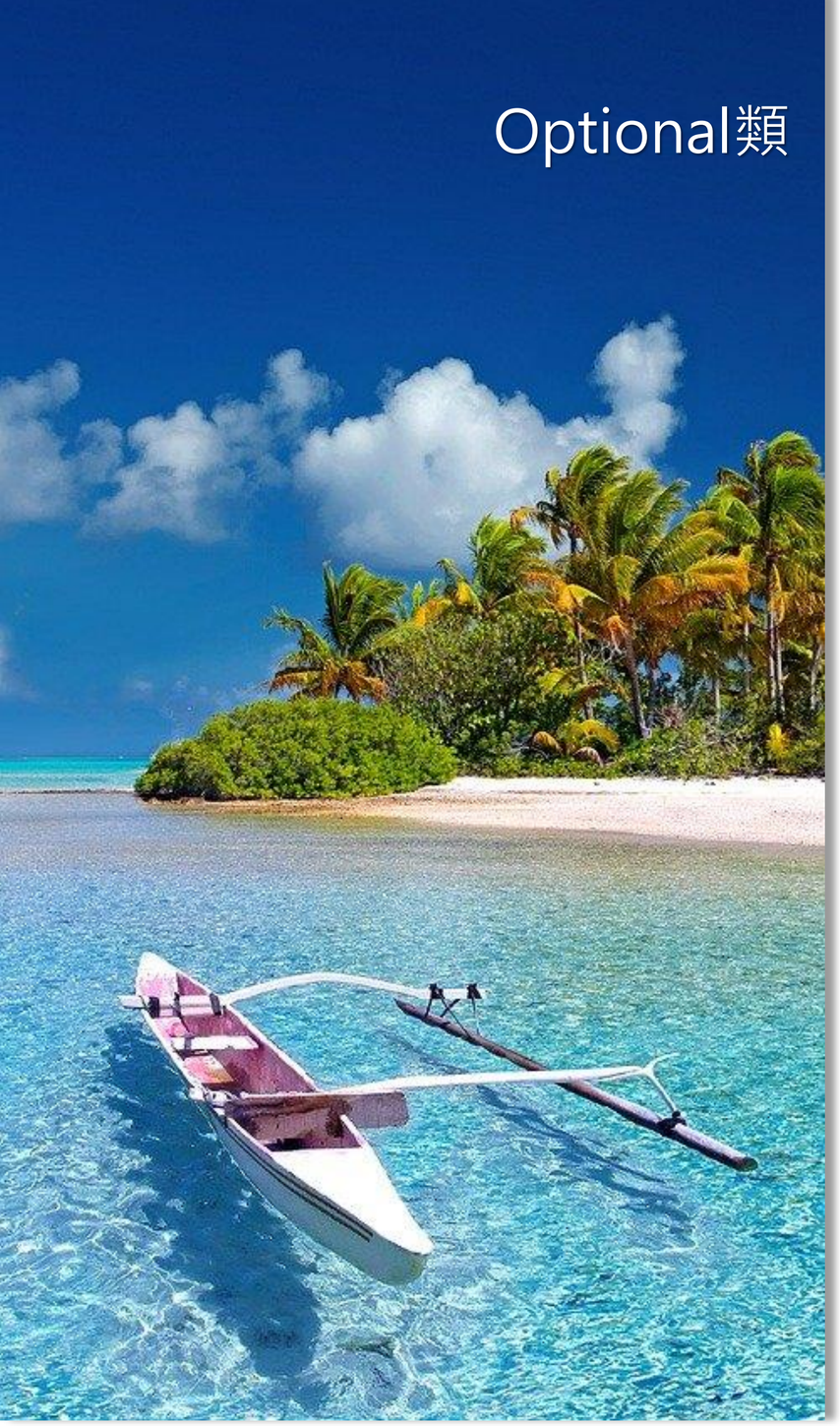
執行緒 CopyOnWriteArrayList



```
var c=new CopyOnWriteArrayList<>(List.of("1","2","3","4"));
        Runnable r=()->{
                try {
                        Thread.sleep(150);
                }
                catch(InterruptedException e) {
                        System.out.println(e);
                }
                c.set(3, "four");
c.add("2");
c.add("7");
                System.out.print(c+" ");
        };
        Thread t=new Thread(r);
        t.start();
        for(var s:c) {
                System.out.print(s+ " ");
                Thread.sleep(500);
        }
```

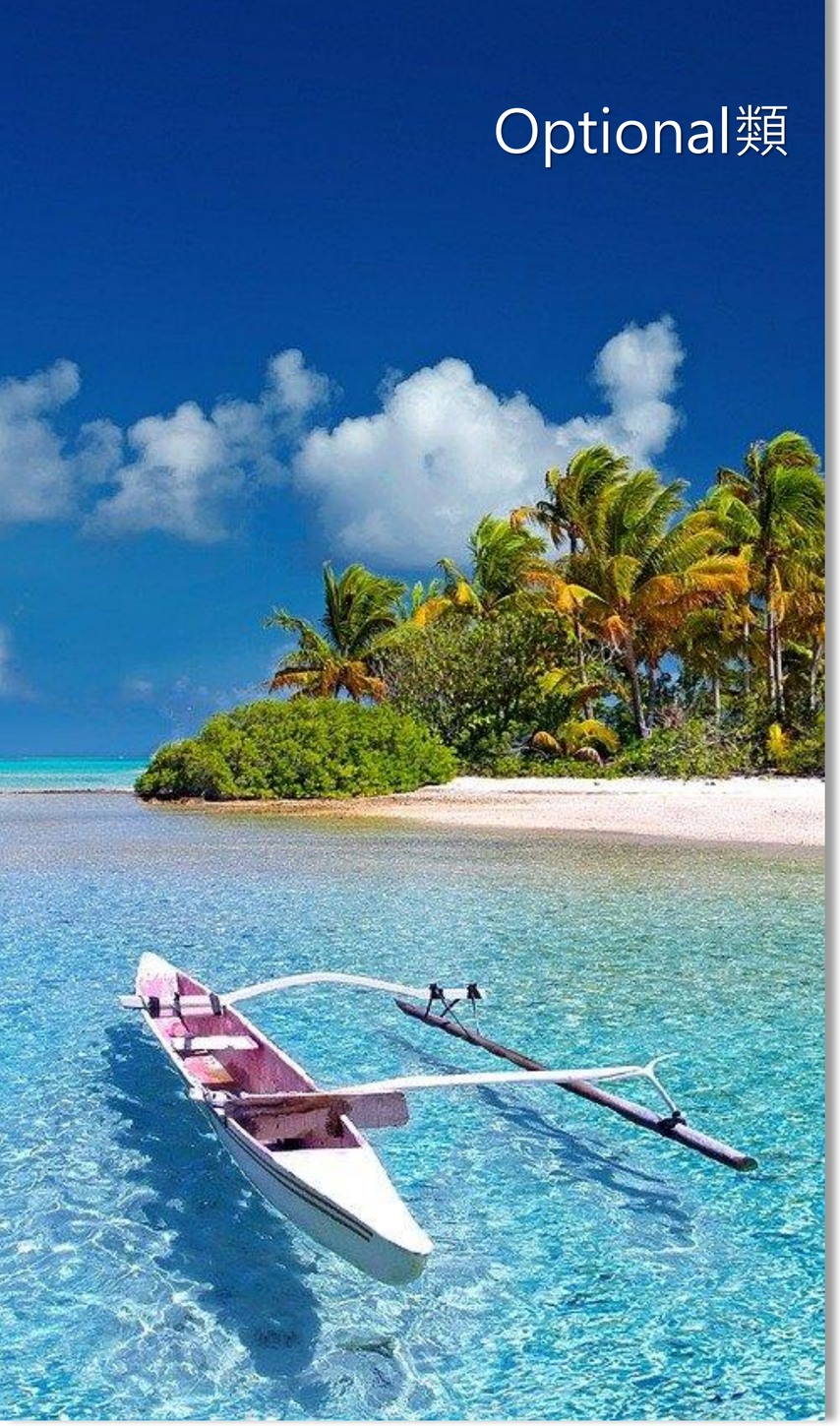
Optinal類

Optional類



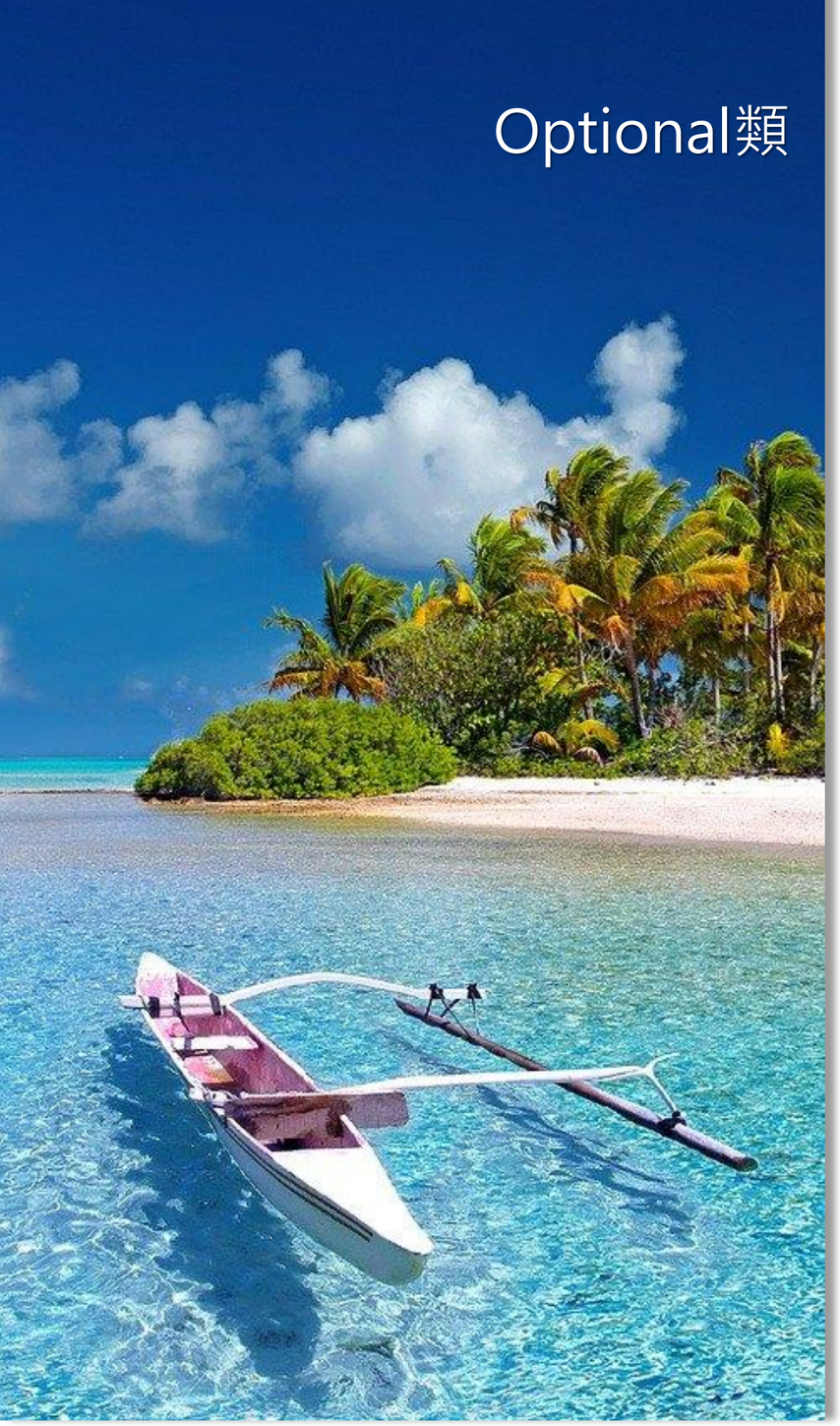
- Optional<T>是一個容器類,它可以保存類型T的值,代表這個值的存在,或者只是保存null值,表示這個值不存在,最主要用來解決空指標的問題
- Optional常用方法
  - Optional.of(T t) 建立一個Optional物件,t 必須非null
  - Optional.empty( ) 建立一個空的Optional物件Optional.empty
  - Optional.ofNullable(T t) t可以為null
  - Optional.orElse(another T t)若不為null回傳原物件,若為null則回傳another t
  - Optional.get( )回傳Optional所包原物件值,而不是Optional物件
    - 若get無內容會拋出NoSuchElementException例 Optional.*ofNullable(null).get()*

Optional類



```
public static void main(String[] args) {
//Cat c=new Cat("Amy");
//c=null;
//Optional<Cat> of = Optional.of(c);//of方法,不能為null
//Optional<Cat> ofNullable = Optional.ofNullable(c);
// Cat orElse = ofNullable.orElse(new Cat("Linda"));
// System.out.println(orElse);
//System.out.println(ofNullable);//若c為null不會發生異常,值為
Optional.empty
//測試getCatName
//Dog dog=new Dog();
//String catName = modifyGetCatName(dog);
//System.out.println(catName);
//String optionalGetCatName = optionalGetCatName(dog);
//System.out.println(optionalGetCatName);
                }
public static String getCatName(Dog dog) {
return dog.getCat().getName();

}
public static String modifyGetCatName(Dog dog) {
if(dog !=null) {
Cat cat=dog.getCat();
if(cat !=null)
{

                return cat.getName();
}
}
return null;

                }
public static String optionalGetCatName(Dog dog) {
Optional<Dog> optionalDog = Optional.ofNullable(dog);
Dog orElseDog = optionalDog.orElse(new Dog(new Cat("amy")));
//需再考慮Cat是否為null
Cat cat = dog.getCat();
Optional<Cat> cat1 = Optional.ofNullable(cat);
Cat cat2 = cat1.orElse(new Cat("Candy"));

return cat2.getName();
                }
}
class Dog{
                private Cat cat;
                public Dog(Cat cat) {
                this.cat = cat;
                }
                public Dog() {}
                public Cat getCat() {
                                return cat;
                }

                public void setCat(Cat cat) {
                                this.cat = cat;
                }

                @Override
                public String toString() {
                                return "Dog [cat=" + cat + "]";
                }

}
class Cat{
                private String name;

                public Cat() {
                                super();
                }
                public Cat(String name) {
                                this.name=name;
                }

                public String getName() {
                                return name;
                }

                public void setName(String name) {
                                this.name = name;
                }

                @Override
                public String toString() {
                                return "Cat [name=" + name + "]";
                }
}
```

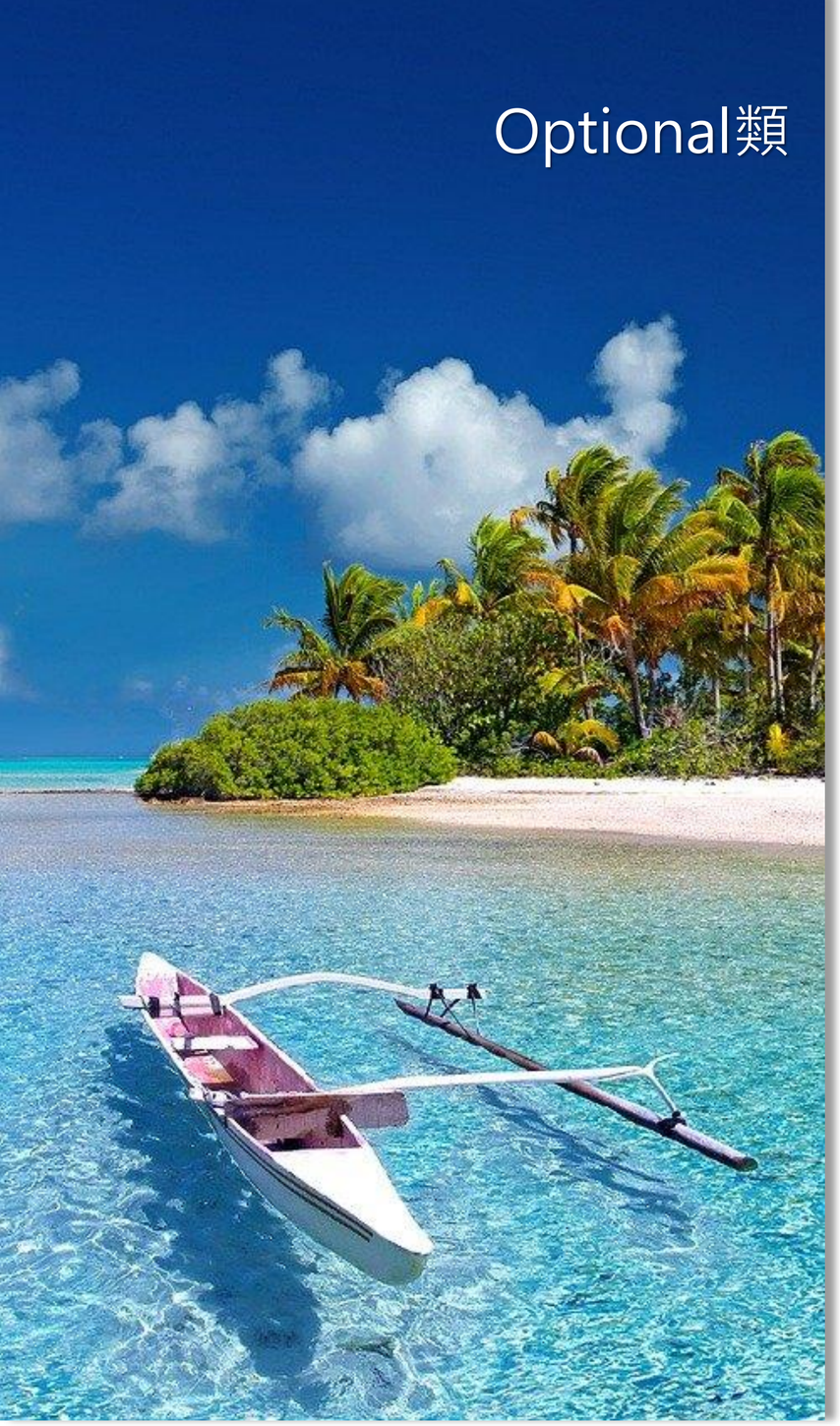
Optional類



```
public class Final22 {
    public static void main(String[] args) {
        System.out.println("Ans :"+convert("a").get());
    }
    private static Optional<Integer> convert (String a){
        try {
            return Optional.of(Integer.parseInt(a));
        }catch(Exception e) {
            return Optional.empty();
        }
    }
}
```

- A. Ans : a
- B. Ans :
- C. A java.util.NoSuchElementException is thrown at run time.
- D. The compilation fails

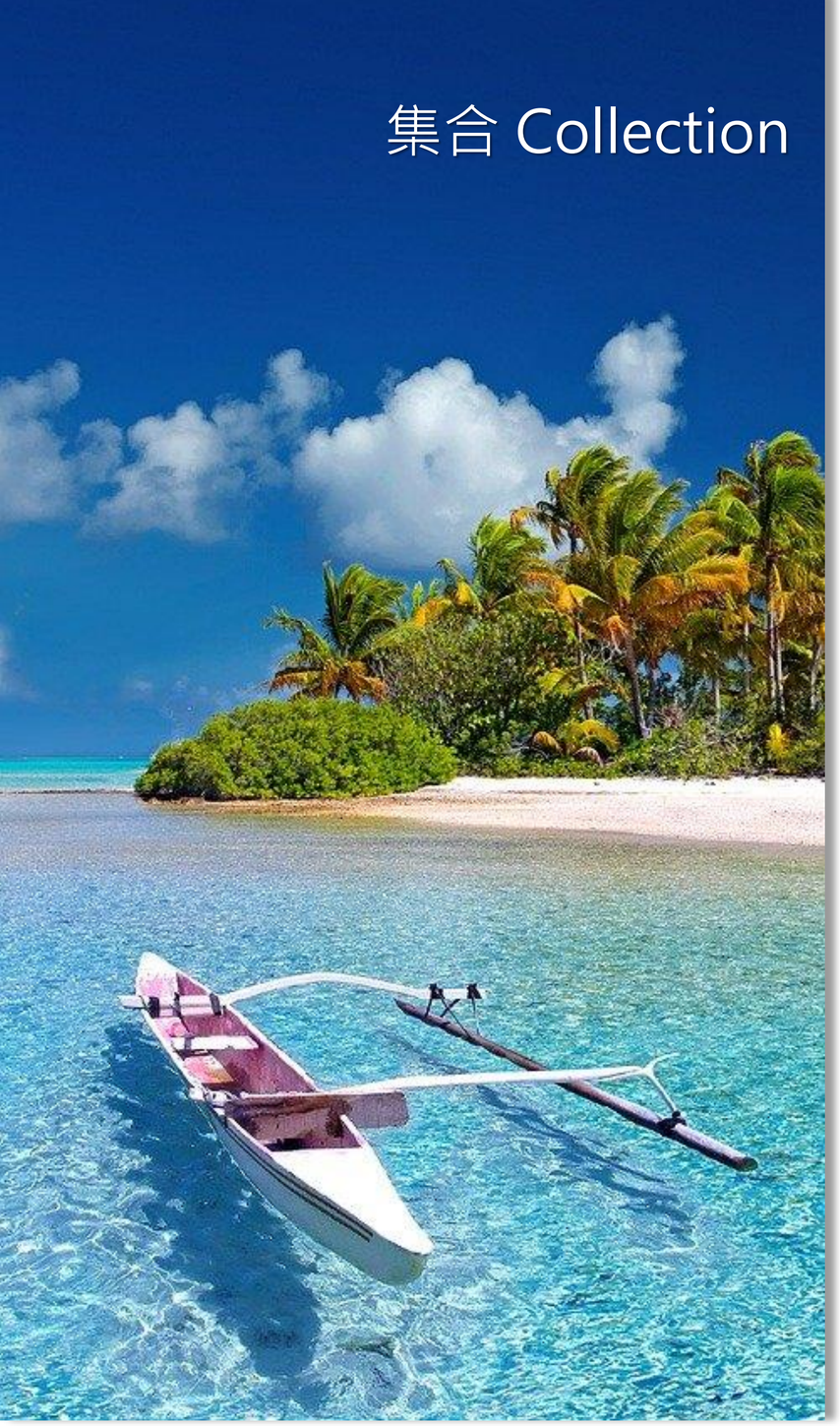
Optional類



```
public static void main(String[] args) {
                System.out.println("Ans :" + convert("a").get());
        }

        private static Optional<Integer> convert(String a) {
                try {
                        return Optional.ofNullable(Integer.parseInt(a));
        } catch (Exception e) {
                        return Optional.empty();
                }
        }
```

集合 COLLECTION

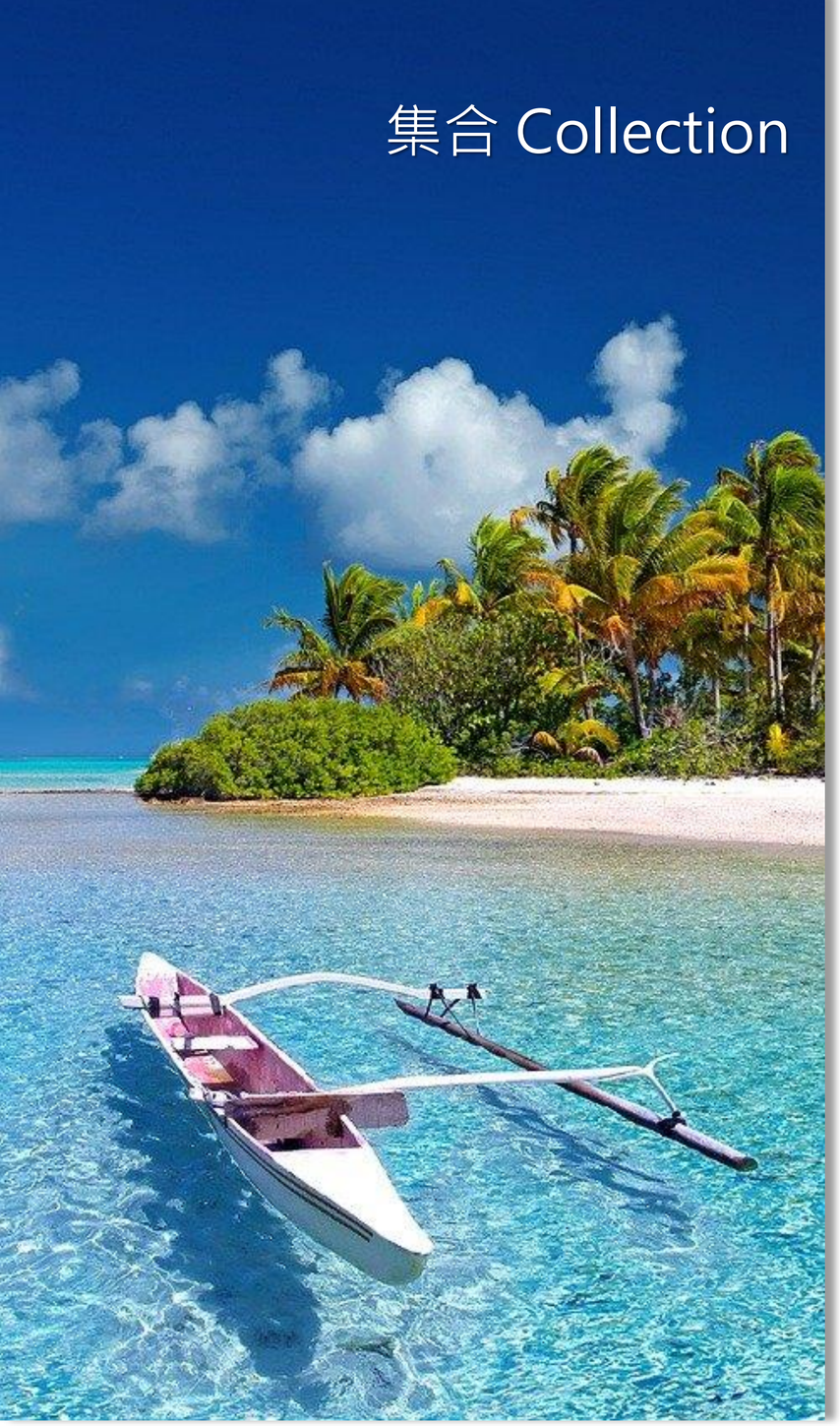
集合 Collection

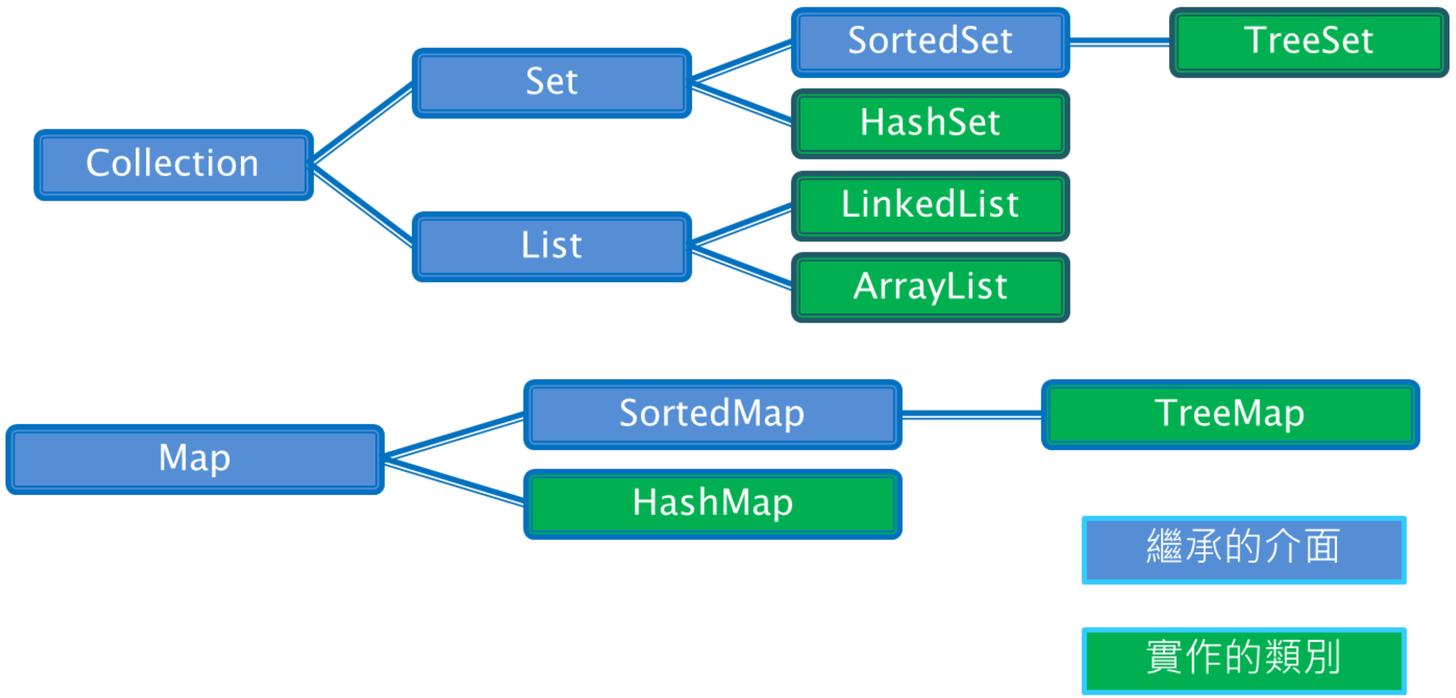


一

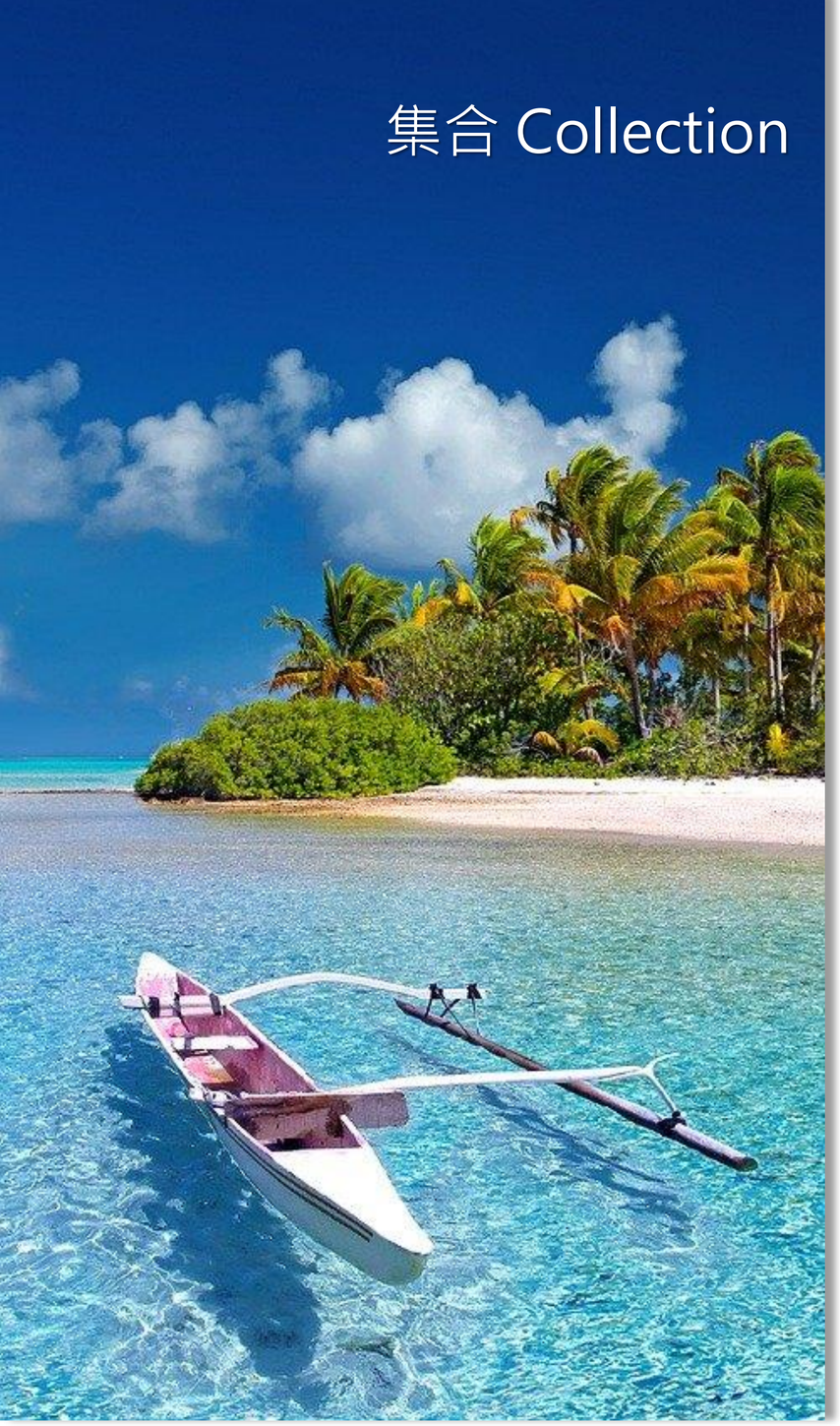
- 程式設計師在撰寫程式時，會在陣列中存放一大群數值資料或物件，但陣列的長度是預先給定且無法變更的，所以在執行時期，程式即無法變更陣列長度以動態產生所需的物件，針對這個問題，java util套件提供了各種類別(The collection API)讓程式開發者使用，如今程式設計師不再為動態資料存取而煩惱，只要會使用這些功能強大的類別，您可以將這些不同的集合想成不同的收納盒，並且可以根據不同的內容擺放物件而選擇不同的收納盒來收納。
- Java使用集合物件Collection來裝載及管理群組物件，概念上類似陣列，但相較陣列具備更多管理功能。
- 被置入集合的物件，其原本的資料型別將不存在，而是以物件object的型態存入集合，若是欲取出時，則需轉成原始的資料型別，否則會產生錯誤
- 集合特點
  - 以介面Collection為代表
  - 集合內的物件稱為elements 簡寫E
  - 集合內的物件不可以是基本型別，若有需要可使用基本型別的包覆類別
  - 大量使用泛型（Generic)
  - 都屬於java.util.*套件
  - Collection 介面繼承了Iterable 介面，因此所有集合物件都具備使用巡訪器的能力

集合 Collection



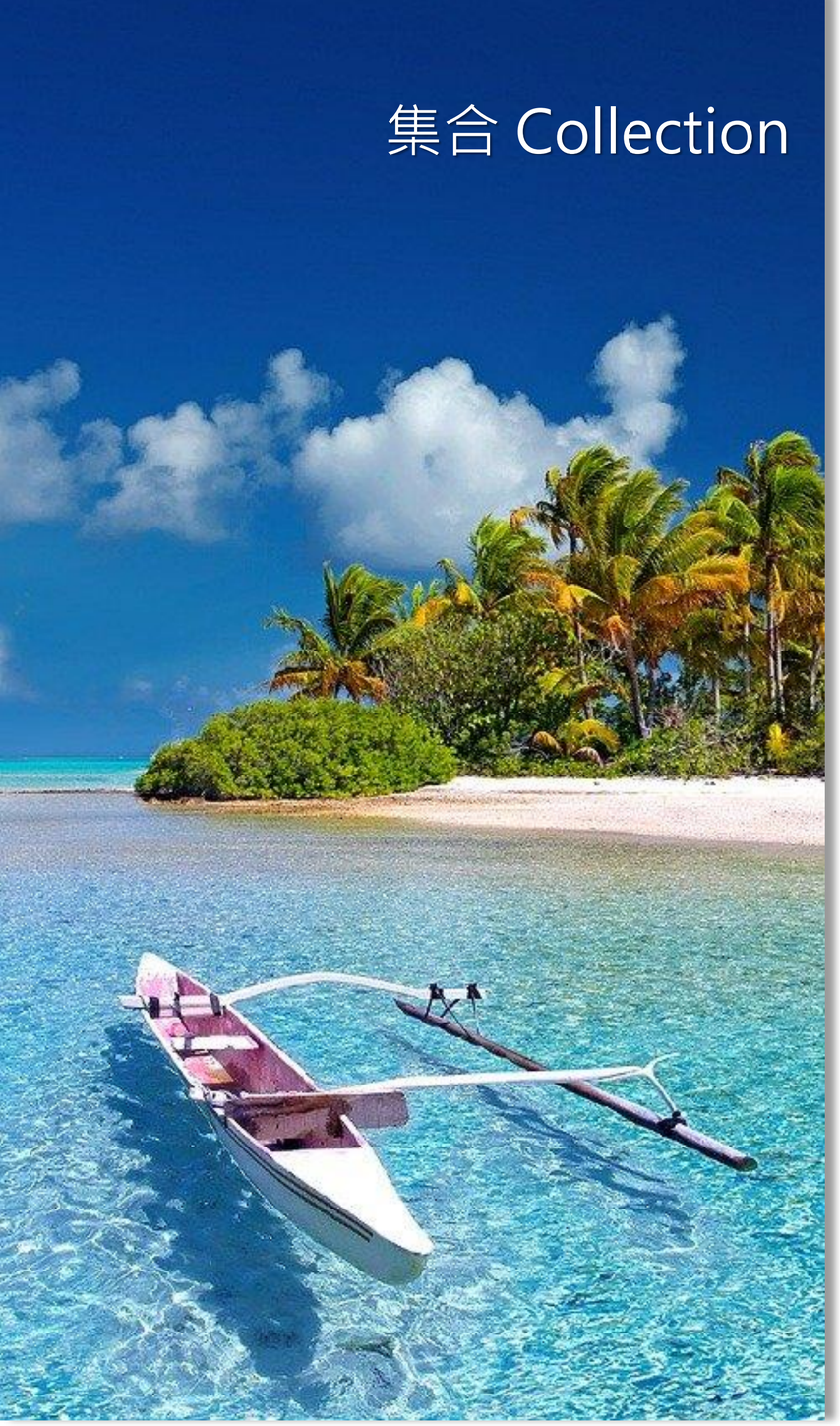


集合 Collection



- Collection 一列資料(物件)
  - List
    - 元素有序(索引),可重覆
  - Set
    - 元素無序,不可重覆
- Map 二列資料(物件),具有映射關係
  - key-value (y=f(x))

集合 Collection

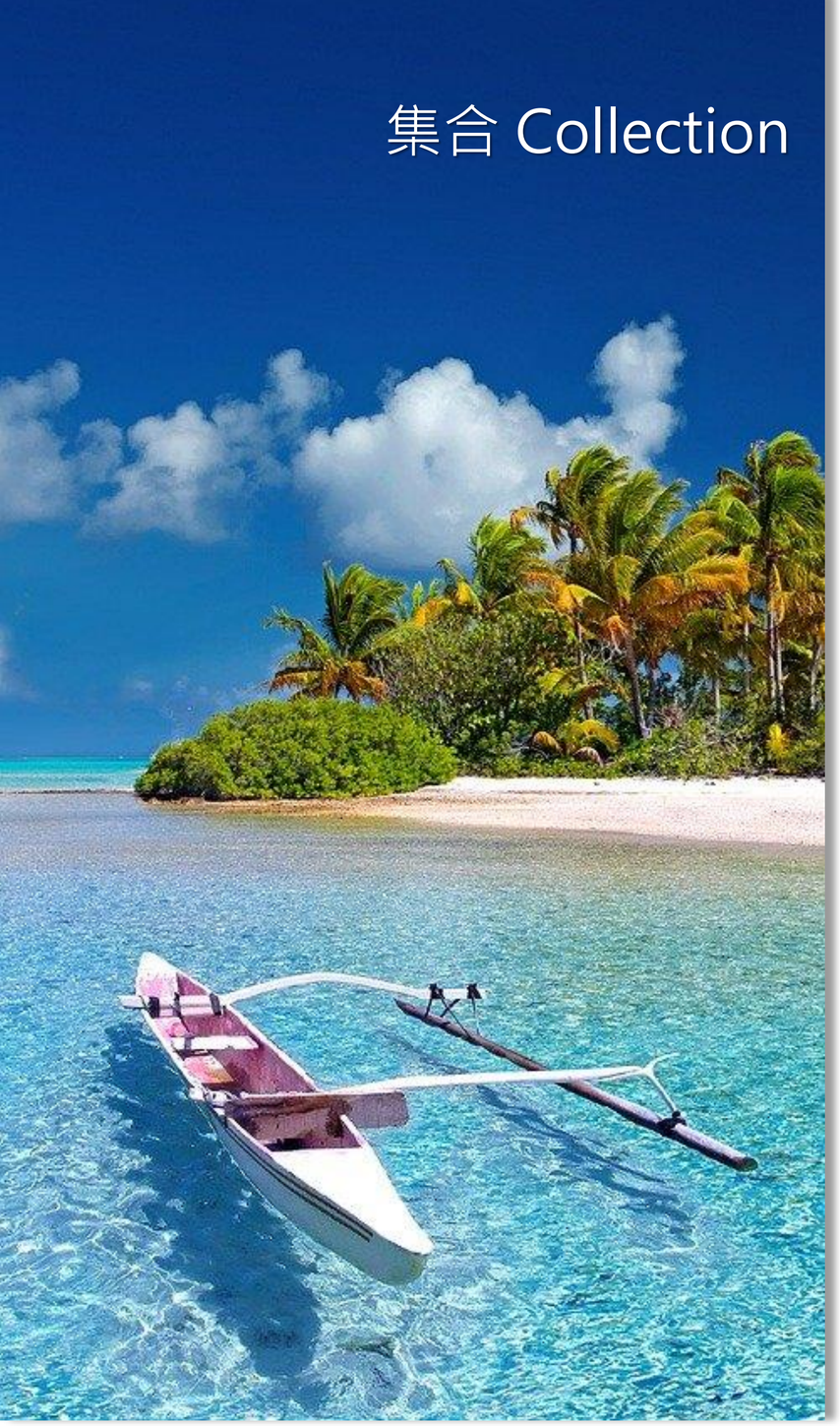


方法名稱	傳回值	說明
add(Object o)	boolean	將指定物件o加入集合中，成功傳回true，失敗 false
addAll(Collection c)	boolean	將集合物件c中所有元素加入集合中，true成功，失敗false
clear()	boolean	清除集合中所有元素
contains(Object o)	boolean	判斷o物件是否屬於該集合
containsAll(Collection c)	boolean	判斷b集合內的元素是否皆包含在A集合內,無關數量
equals(Object o)	boolean	判斷二集合是否相等,list需考慮順序是否相同,set則不用考慮
isEmpty()	boolean	判斷集合是合為空集合
remove(Object o)	boolean	移除集合中的元素,內容為物,不是索引值
removeAll(Collection c)	boolean	移除集合中對應到某集合的所有元素
retainAll(Collection c)	boolean	保留集合中交集於某集合的元素
size( )	int	回傳集合中的元素數目
iterator( )	iterator	使用iterator方法之走訪器
hashCode( )	int	

PS：remove方法若是new HashSet()是物件,因其是無序沒有索引值則為物件,例 remove(111),此111為 Integer而不是索引

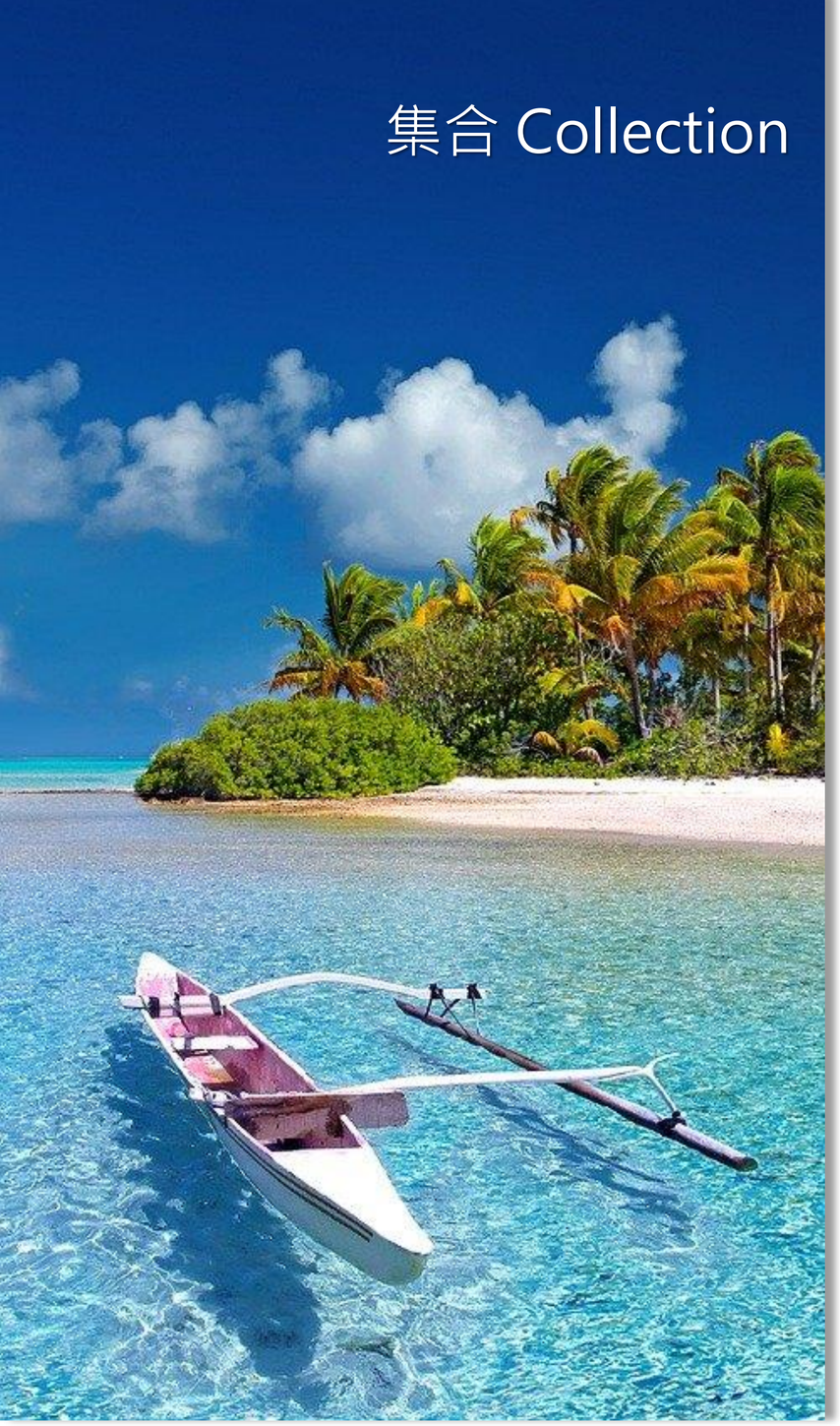
另外若是new ArrayList()則是索引值,因為List的介面的remove方法為 remove(index n) 及 remove(Object n)

集合 Collection



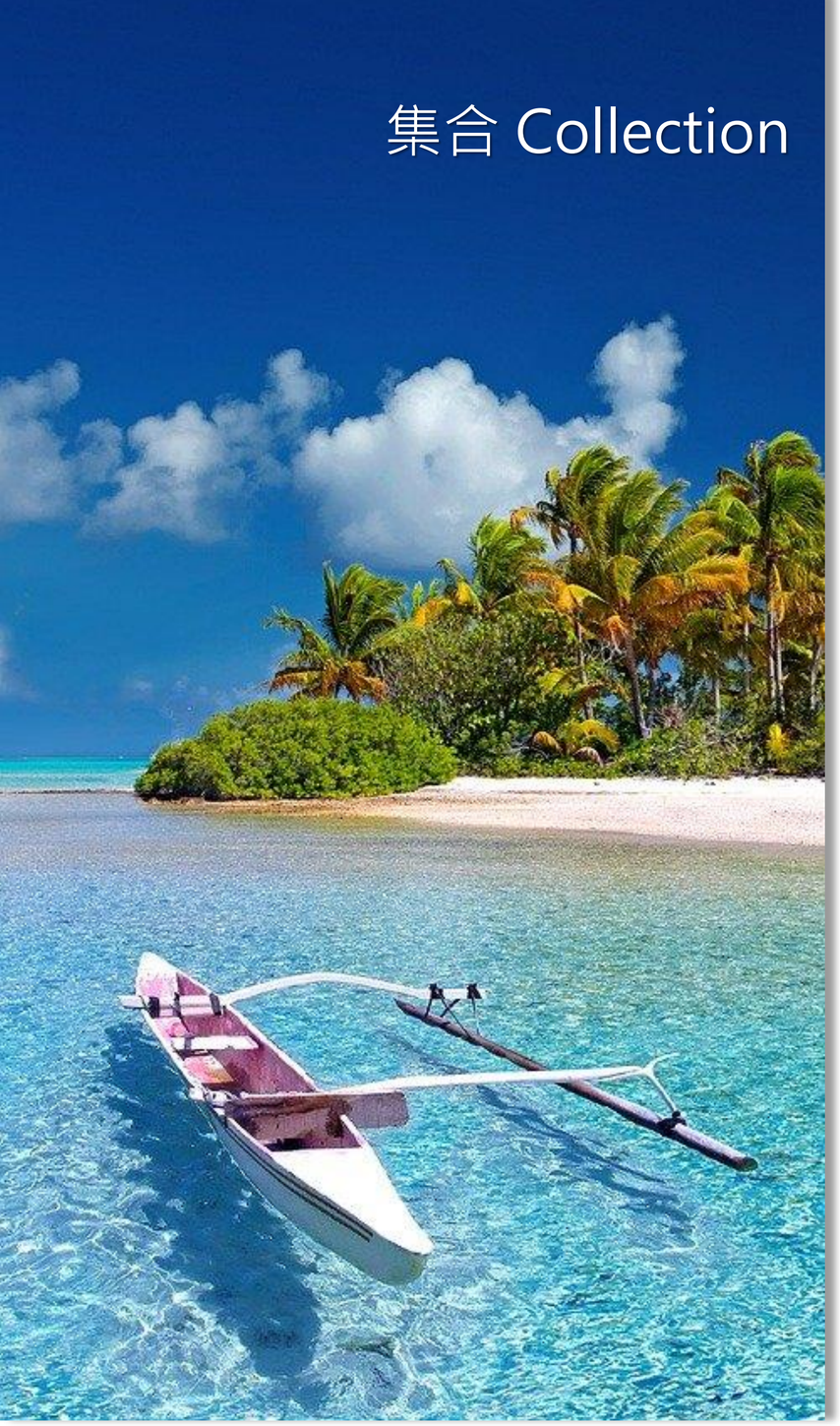
```
List ls=new ArrayList();
        ls.add(123);
        ls.add("java");
        ls.add(new Date());
        ls.add(123.456);
        System.out.println(ls);
        List ls2=new ArrayList();
        ls2.add(111);
        ls2.add("programe");
        ls2.add(222.456);
        ls.add(ls2);
        System.out.println(ls);
        System.out.println(ls.size());
        //addAll會將集合拆開再加入
        ls.addAll(ls2);
        System.out.println(ls.size());
        System.out.println(ls);
       System.out.println(ls.contains(123.456));
      System.out.println(ls.contains("123.456"));
        ls2.add(111);
        System.out.println(ls2);
        System.out.println(ls.containsAll(ls2));
        //移除以物件表示,而不是索引值
        System.out.println(ls2.remove(new Integer(111)));
        System.out.println(ls2);
        //移除二集合相同之元素
        ls.removeAll(ls2);
        System.out.println(ls);
        List ls3=new ArrayList();
        ls3.add(123);
        ls3.add("java");
        //保留二集合相同之元素
        ls.retainAll(ls3);
        System.out.println(ls);
        ls.clear();
        System.out.println(ls.isEmpty());
```

集合 Collection



```
//自訂類別需覆寫Object類的equals方法,否則無法比較
class Person {
    String name;
    int age;
    public Person() {    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Person{" + "姓名=" + name + ", 年齡=" + age + '}';
    }
}
//以下程式撰寫在主方法內
 List ls=new ArrayList();
        ls.add(123);
        ls.add(true);
        ls.add(new String("java"));
        ls.add(new Person("amy",25));

System.out.println(ls.contains("java"));

System.out.println(ls.contains(new Person("amy",25)));
```

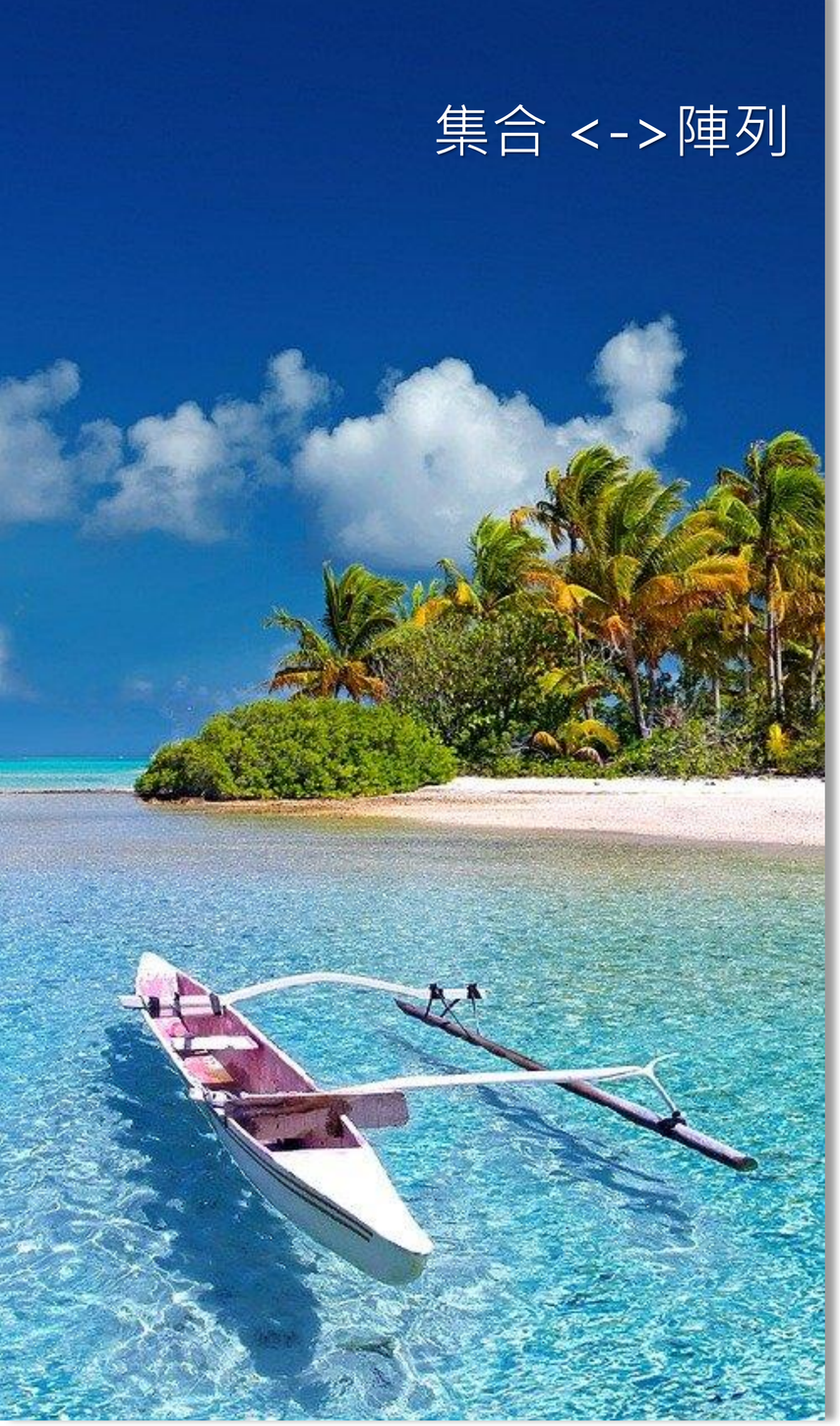
集合 Collection



```
Collection col2=new ArrayList();
    col2.add(123);
    col2.add(456);
    col2.add(789);
    Collection col3=new ArrayList();
    col3.add(456);
    col3.add(123);
    col3.add(789);
     System.out.println(col2.equals(col3));//false List為有序的
```

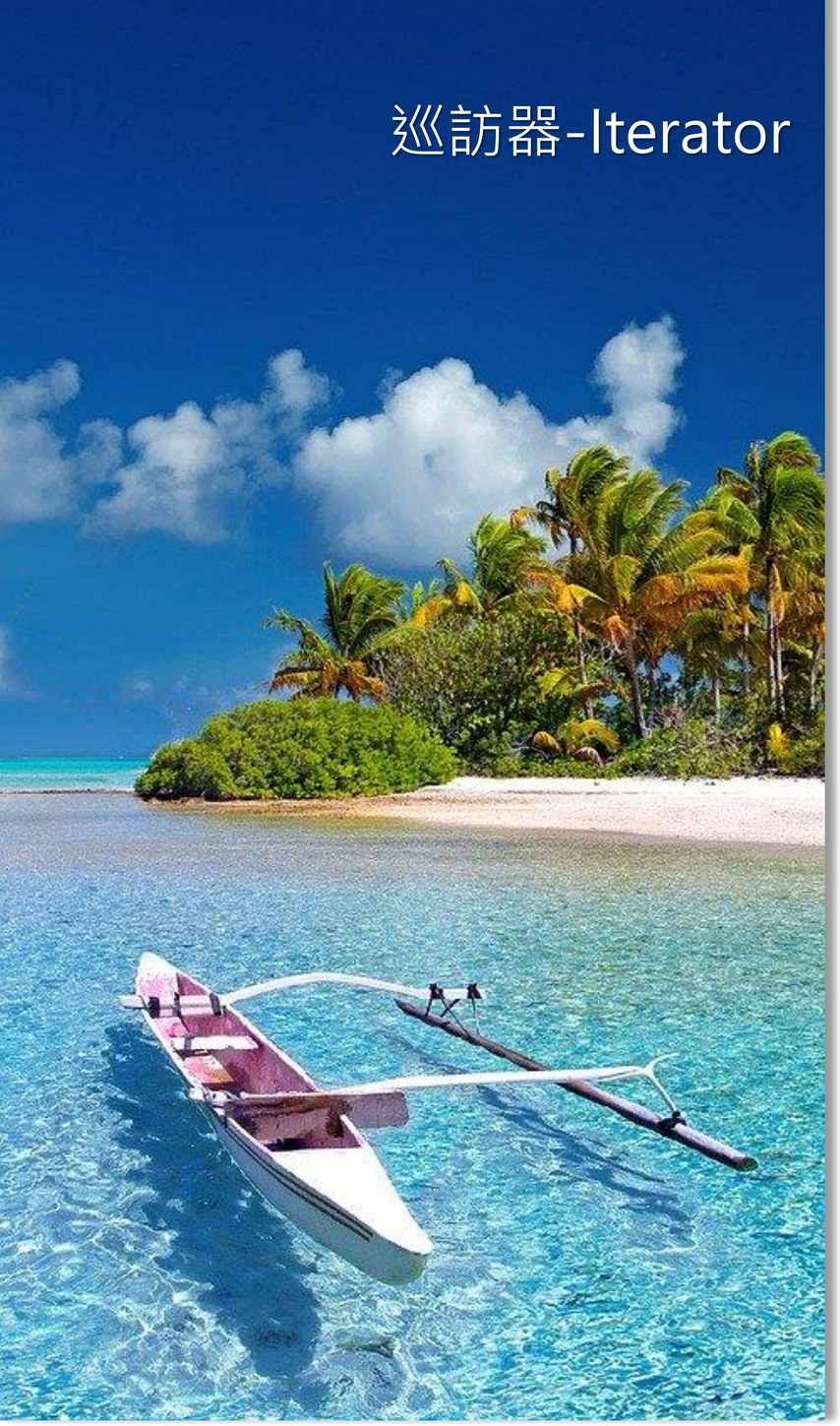
```
Collection col2=new HashSet();
    col2.add(123);
    col2.add(456);
    col2.add(789);
    Collection col3=new HashSet();
    col3.add(456);
    col3.add(123);
    col3.add(789);
     System.out.println(col2.equals(col3));//true Set為無序的
```

集合 <->陣列

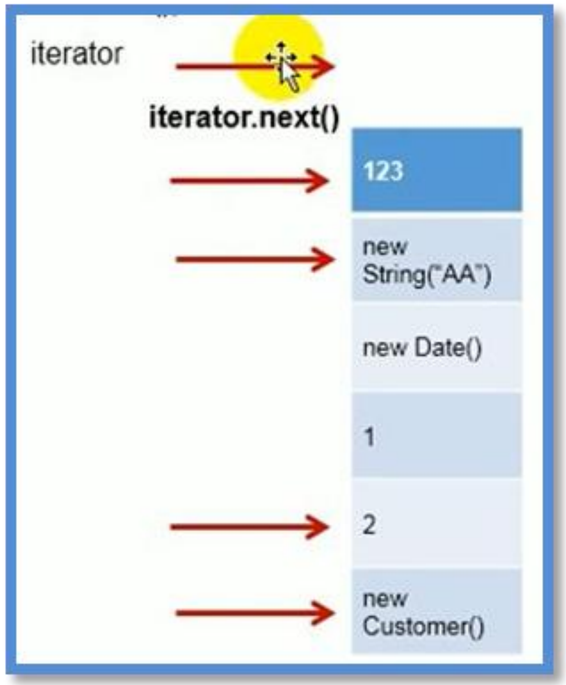


```
System.out.println("===集合轉陣列======");
        Collection col = new ArrayList();
        col.add(123);
        col.add(true);
        col.add(new String("java"));
        col.add(new Person("amy", 25));
        Object obj[] = col.toArray();
        for (Object x : obj) {
            System.out.println(x);
        }
        System.out.println("=====陣列轉集合======");
        Collection col2 = Arrays.asList(new String[]{"java", "oca", "ocp"});
        System.out.println(col2);
        Collection col3 = Arrays.asList(new int[]{11, 22, 33});//需注意
        System.out.println(col3);
        Collection col4 = Arrays.asList(new Integer[]{11, 22, 33});
        System.out.println(col4);
```

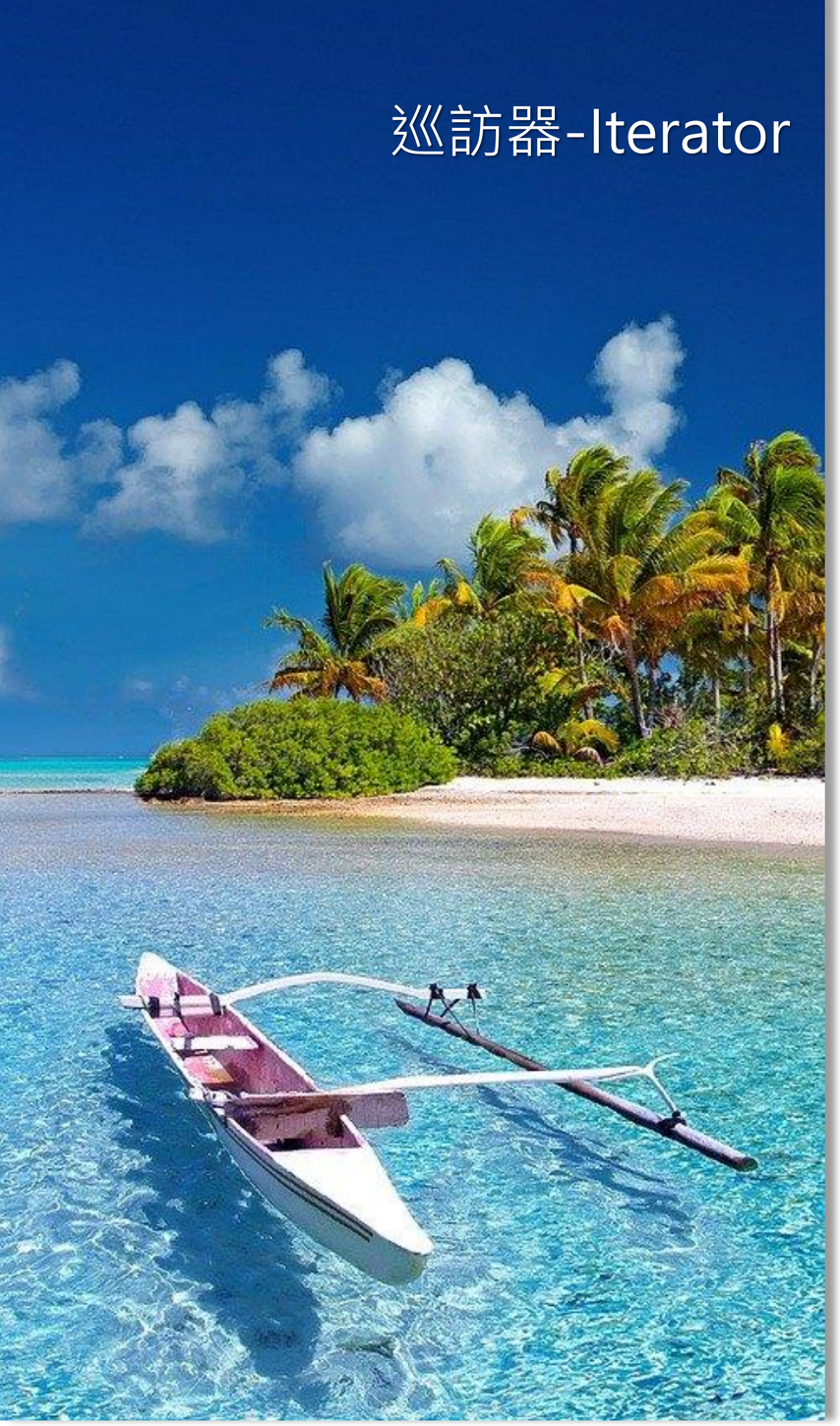
巡訪器-Iterator



```
//用途在於巡訪集合中所有元素
//利用Collection的iterator方法返回Iterator//介面的實例
Collection col = new ArrayList();
        col.add(123);
        col.add(true);
        col.add(new String("java"));
        col.add(new Person("amy", 25));
        Iterator iterator = col.iterator();
        System.out.println(iterator.next());
        System.out.println(iterator.next());
        System.out.println(iterator.next());
        System.out.println(iterator.next());
        //NoSuchElementsException
        System.out.println(iterator.next());
it=col.iterator();//需重新再指定一次
for(int m=0;m<col.size();m++)
            System.out.println(it.next());

        //        System.out.println("==================================");

  it=col.iterator();
        while(it.hasNext())
            System.out.println(it.next());
//二種使用疊代器錯誤的用法
//方法一,會產生 NoSuchElementException,且會跳著印 會印 2,4
Iterator it=col.iterator();
while(it.next() !=null)
 System.out.println(it.next());
//方法二,要先註解上方的程式碼才可測試
//會產生無限循環，只印第一個元素,因為是二個不同的it
while(col.iterator().hasNext())
            System.out.println(col.iterator().next());
```



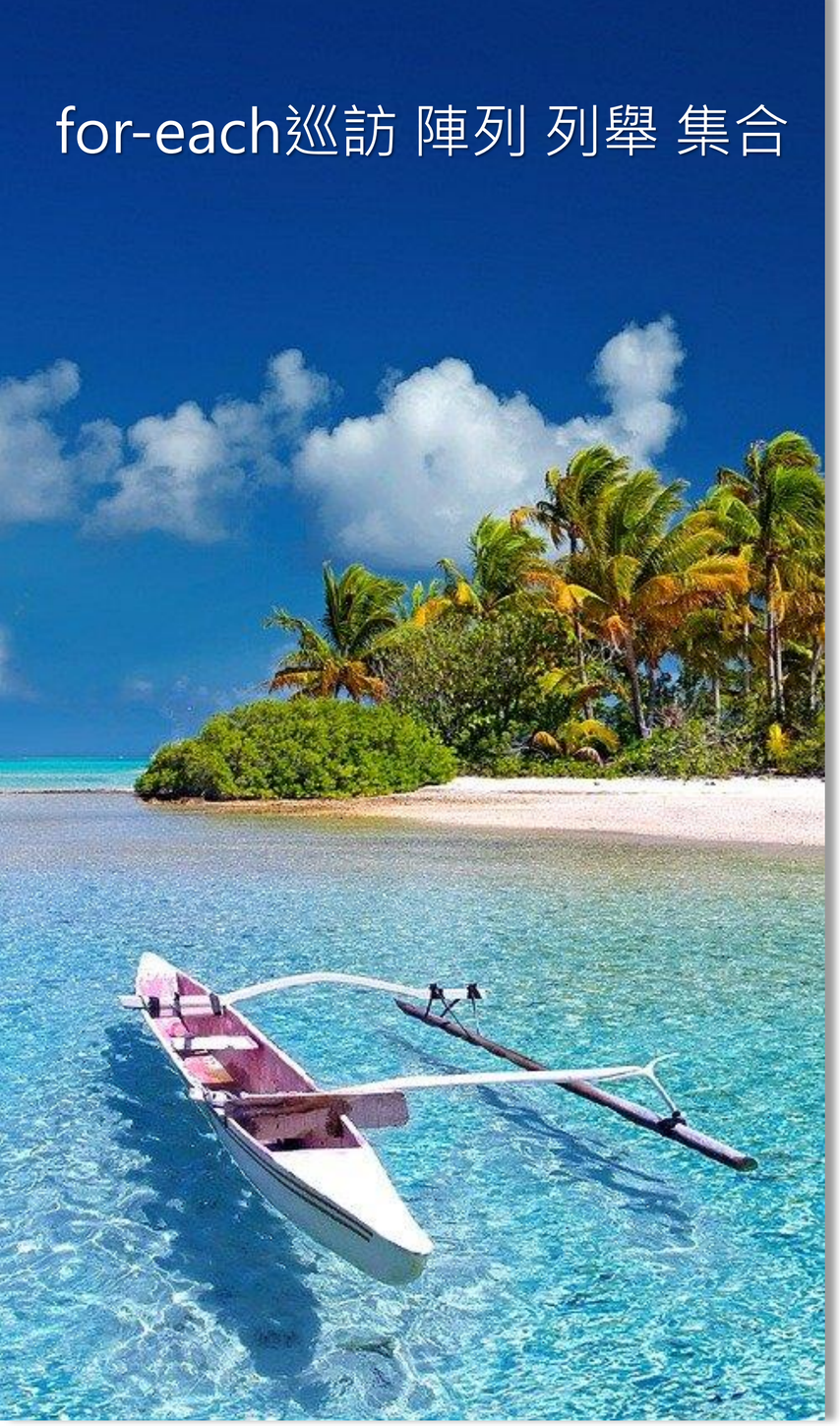
巡訪器-Iterator



```
ArrayList<Integer> a1=new ArrayList<>();
a1.add(1);
a1.add(2);
a1.add(3);
Iterator<Integer> itr=a1.iterator();
while(itr.hasNext()) {
    if(itr.next()==2) {
    a1.remove(2);
    System.out.println(itr.next());
    }
}
```

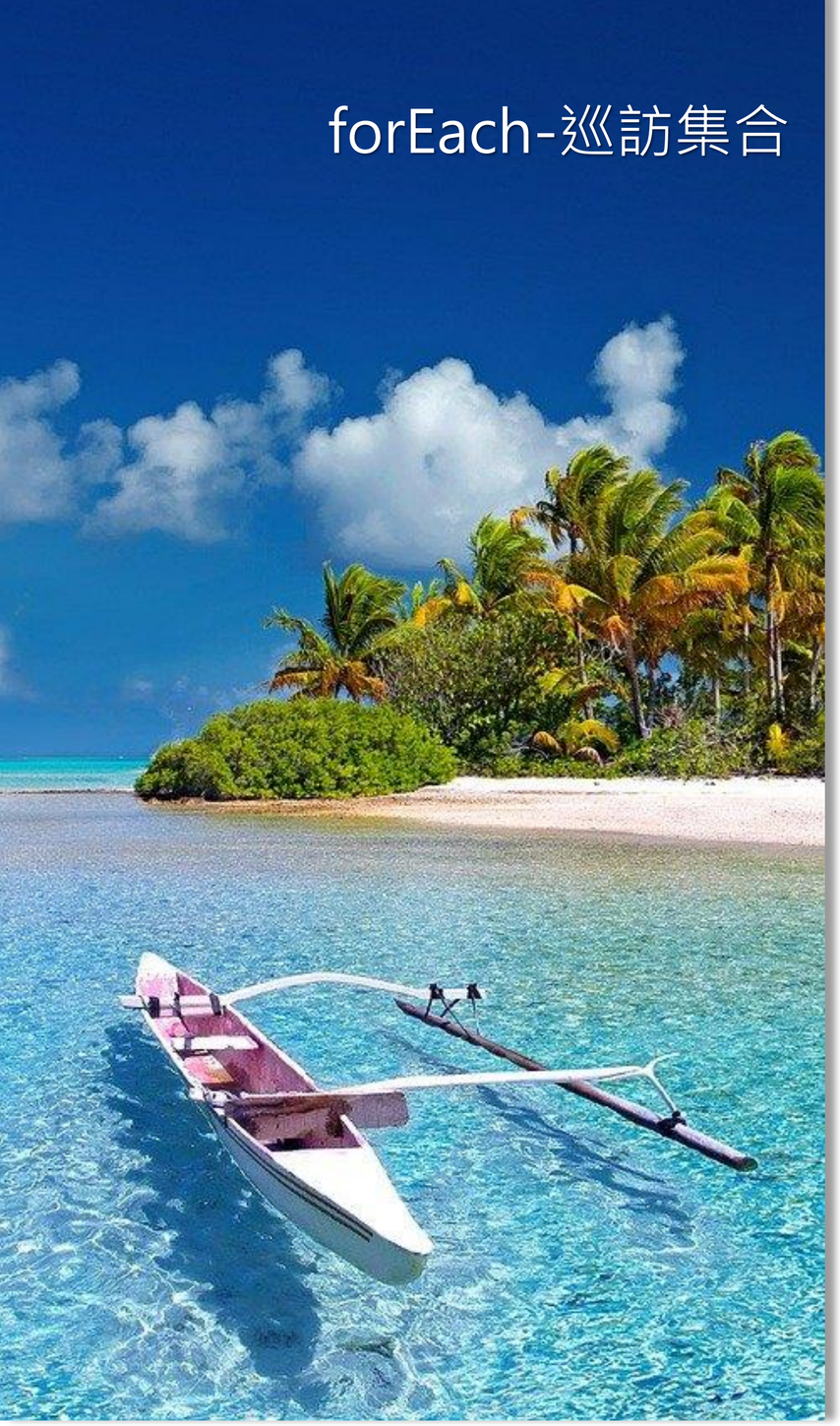
- A. 1 2 followed by an exception
- B. A ConcurrentModificationException is thrown at run time
- C. 1 2 3 4
- D. 1 2 3 followed by an exception

for-each巡訪 陣列 列舉 集合



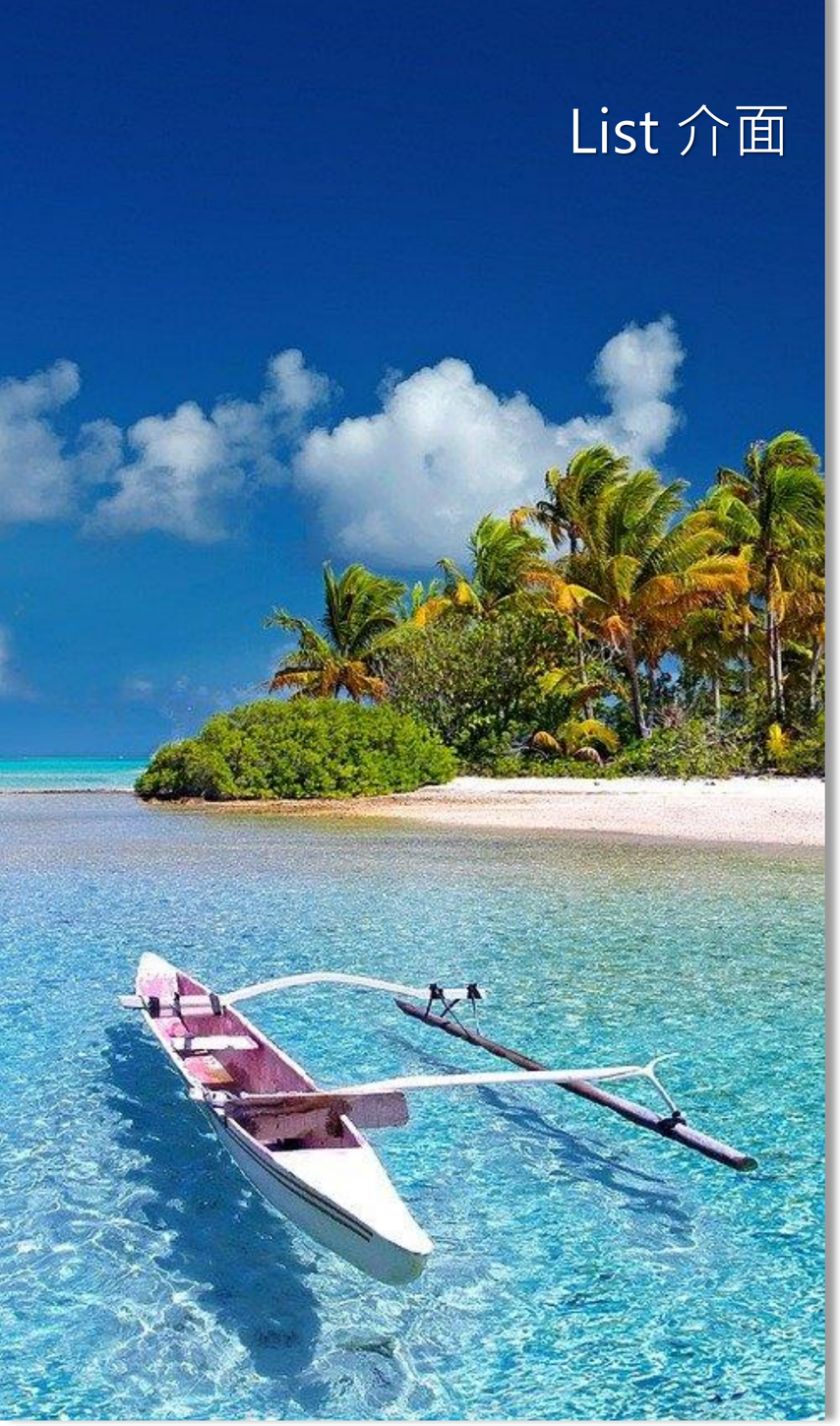
```
enum PowerState{
    ON,OFF,SUSPEND,NOTWORKING;
}
//以下程式貼至主方法內
        int arr[]={11,22,33,44,55};
        Collection col = new ArrayList();
        col.add(123);//自動裝箱
        col.add(new Date());
        col.add("abc");
        col.add(true);//自動裝箱
   //利用foreach巡訪列舉,陣列,集合
        System.out.println("==========
巡訪陣列==============");
        for(int x:arr)
            System.out.print(x+"\t");
        System.out.println();
       System.out.println("==========
巡訪列舉==============");
        for(PowerState x:PowerState.values())
            System.out.print(x+"\t");
         System.out.println();

System.out.println("==========巡訪
集合==============");
        for(Object x:col)
            System.out.print(x+"\t");
         System.out.println();
```

forEach-巡訪集合



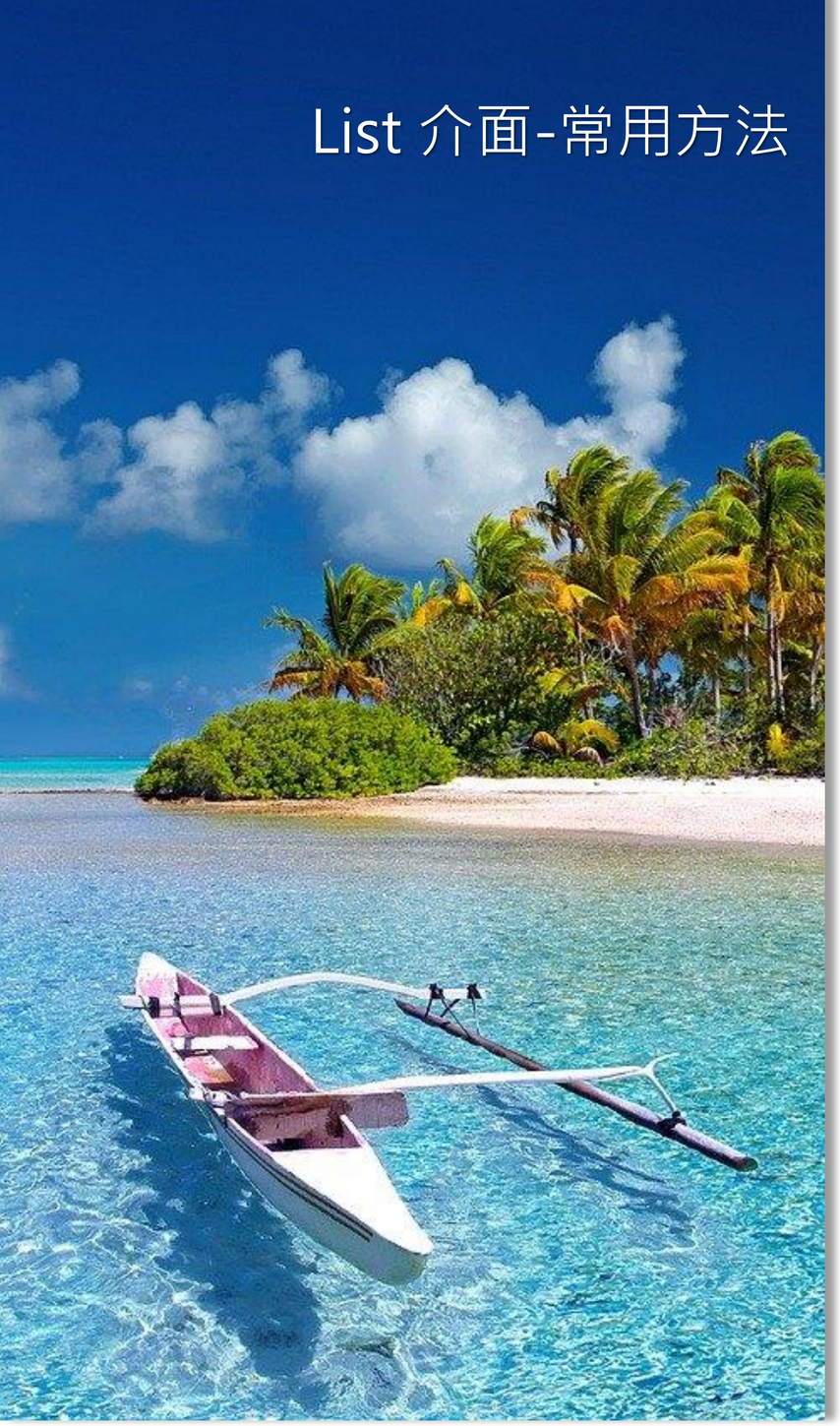
```
//父介面Iterable的預設方法
Collection list=new ArrayList();
      list.add(123);
      list.add(456);
      list.add(789);
      list.add(222);
      list.forEach(new Consumer(){
          @Override
          public void accept(Object t) {
              System.out.println(t);
          }
      });
//Lambda表達式
list.forEach((t)-> System.out.println(t));
//方法參照
 list.forEach(System.out::println);
```

List 介面



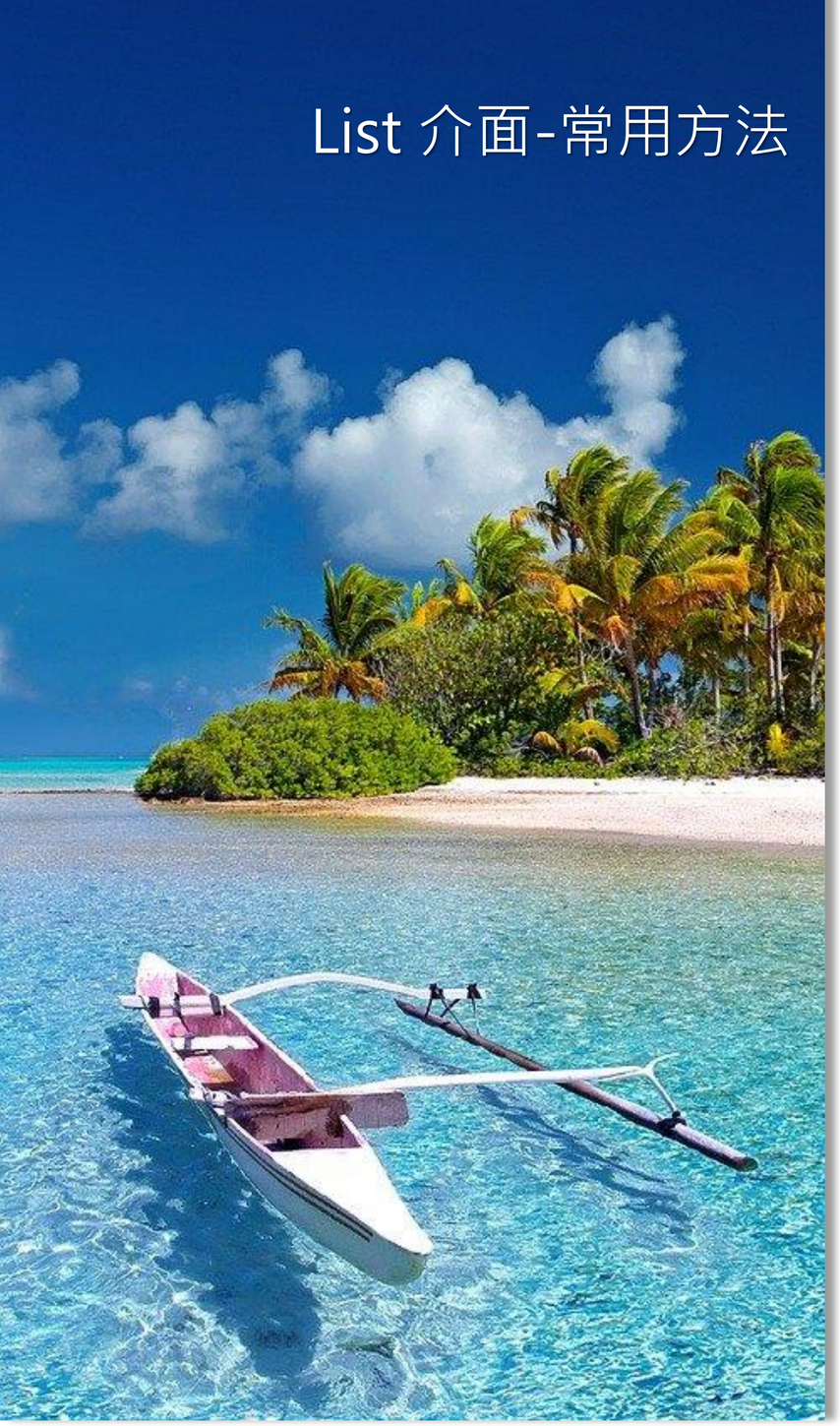
- 有別於java陣列中儲存資料的局限性,我們通常使用List來替代陣列
- List介面集中的元素是有序的,且可重覆的,集合中的每個元素都有其對應的順序索引
- List容器中的元素都對應一個整數型的序號,記載其在容器中的位置,可以據序號存取容器中的元素
- List介面的實現類常用的有 ArrayList ,LinkedList 和 Vector

List 介面-常用方法



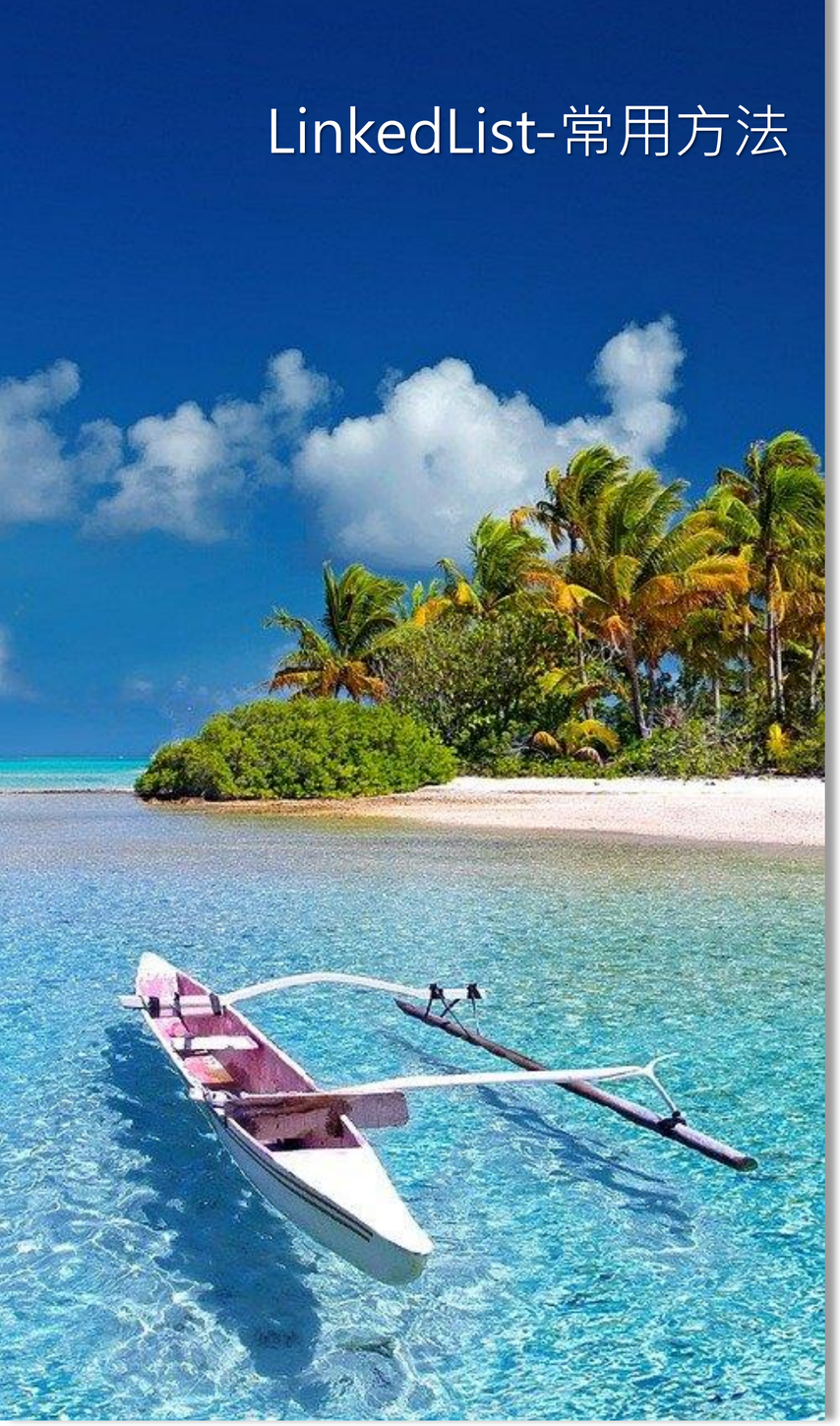
- add(index,object) 將object插入到特定位置
- get(index) 取得某索引位置的元素
- indexOf(object) 求出元素位置
- lastIndexOf(object) 求出元素最後一次出現的位置
- remove(index) 刪除某索引編號的位置元素或元素
- set(index,object) 更改元素內容
- subList(begin,end) 取得begin 到end-1的元素

List 介面-常用方法



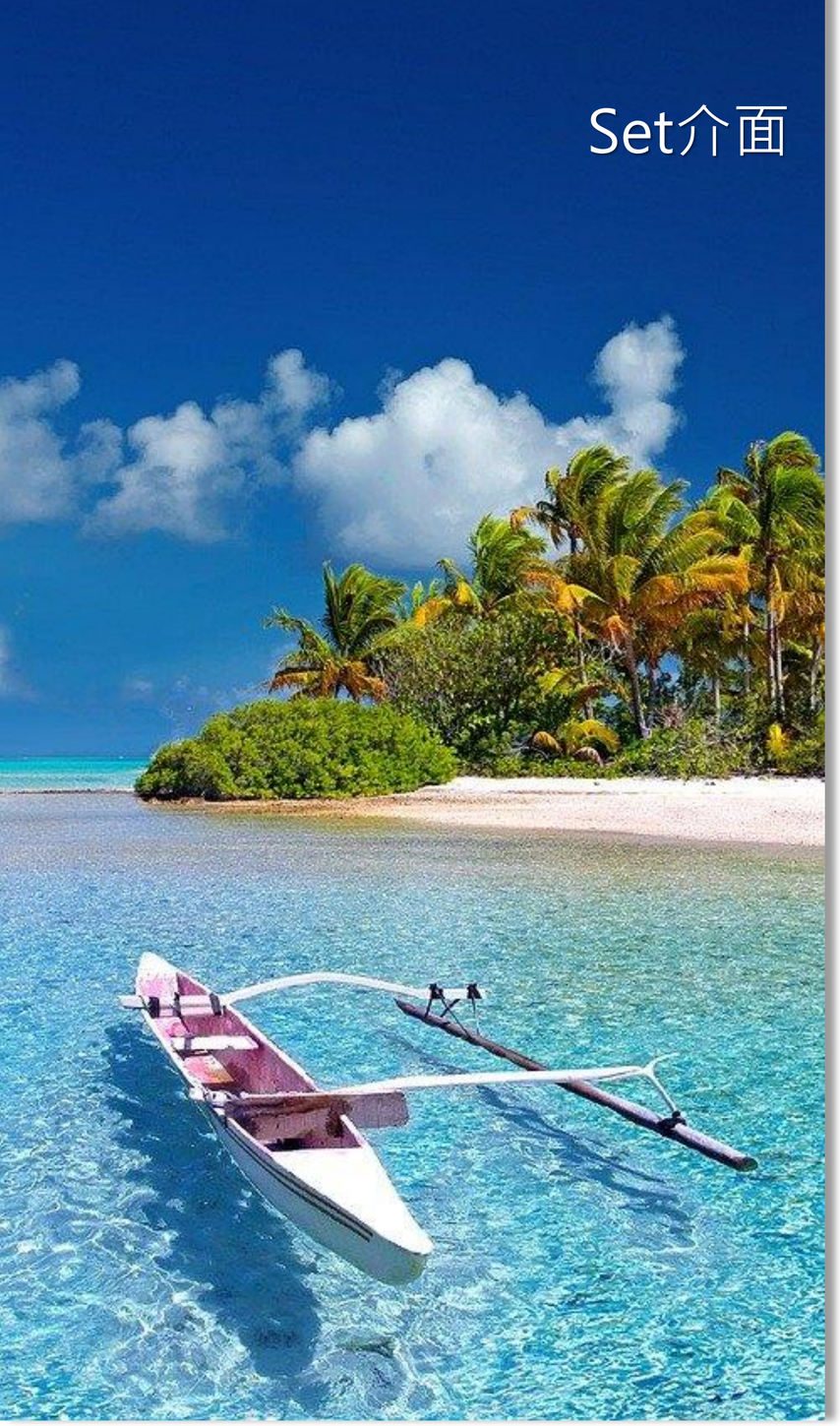
- List list = new ArrayList();
- list.add(111);
- list.add(222);
- list.add(333);
- list.add("456");
- list.add(new Date());
- System.out.println(list);
- list.add(1, 1515);//插入在索引編號為1的位置,即222往下移
- System.out.println(list);
- //取得某索引值的內容 get(index)
- System.out.println(list.get(1));//1515
- //查詢某元素位於集合中第幾個位置,有找到回傳索引值,沒找到回傳 -1
- System.out.println(list.indexOf(1515));//1
- System.out.println(list.indexOf(1818));//-1
- list.add(1515);//加到最後,但移除從前面
- System.out.println(list.lastIndexOf(1515));//6
- //list.remove(1515);//元素名稱,此行會發生異常,索引超出範圍,要以物件方式刪除new Integer("1515");
- list.remove(6);//索引編號
- System.out.println(list);
- //set(index,元素)更改元素內容
- list.set(3, "hello");//將索引編號為3的內容改為hello
- System.out.println(list);
- List subList=list.subList(2, 5);//取得begin 到end-1的元素
- System.out.println(subList);//[222, hello, 456]

LinkedList-常用方法



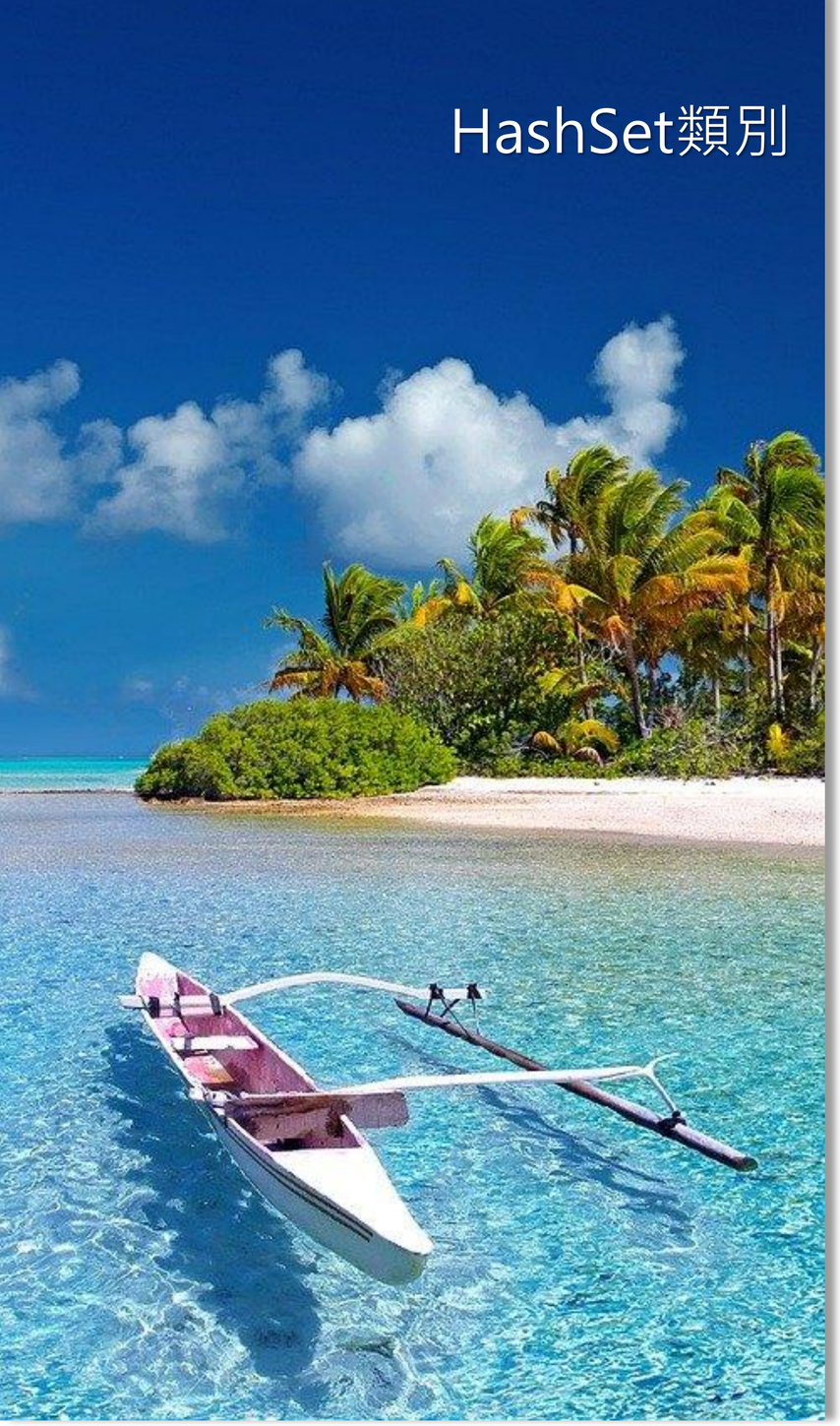
```
LinkedList<Integer> li=new LinkedList<Integer>();
    for(int i=1;i<=5;i++)
        li.add(i);
    li.addFirst(100);
    li.addLast(900);
    li.addFirst(50);
    li.add(3,88);
    li.add(900);
    System.out.println(li);
    System.out.println("取得第三個位置的元素："+li.get(2));
    System.out.println("傳回5在集合中第一次出現的位置為："+li.indexOf(5));
    System.out.println("刪除最後一個元素："+li.removeLast()+".....");
    System.out.println(li);
    System.out.println("第一個元素為："+li.getFirst());
    System.out.println("最後一個元素為："+li.getLast());
```

Set介面



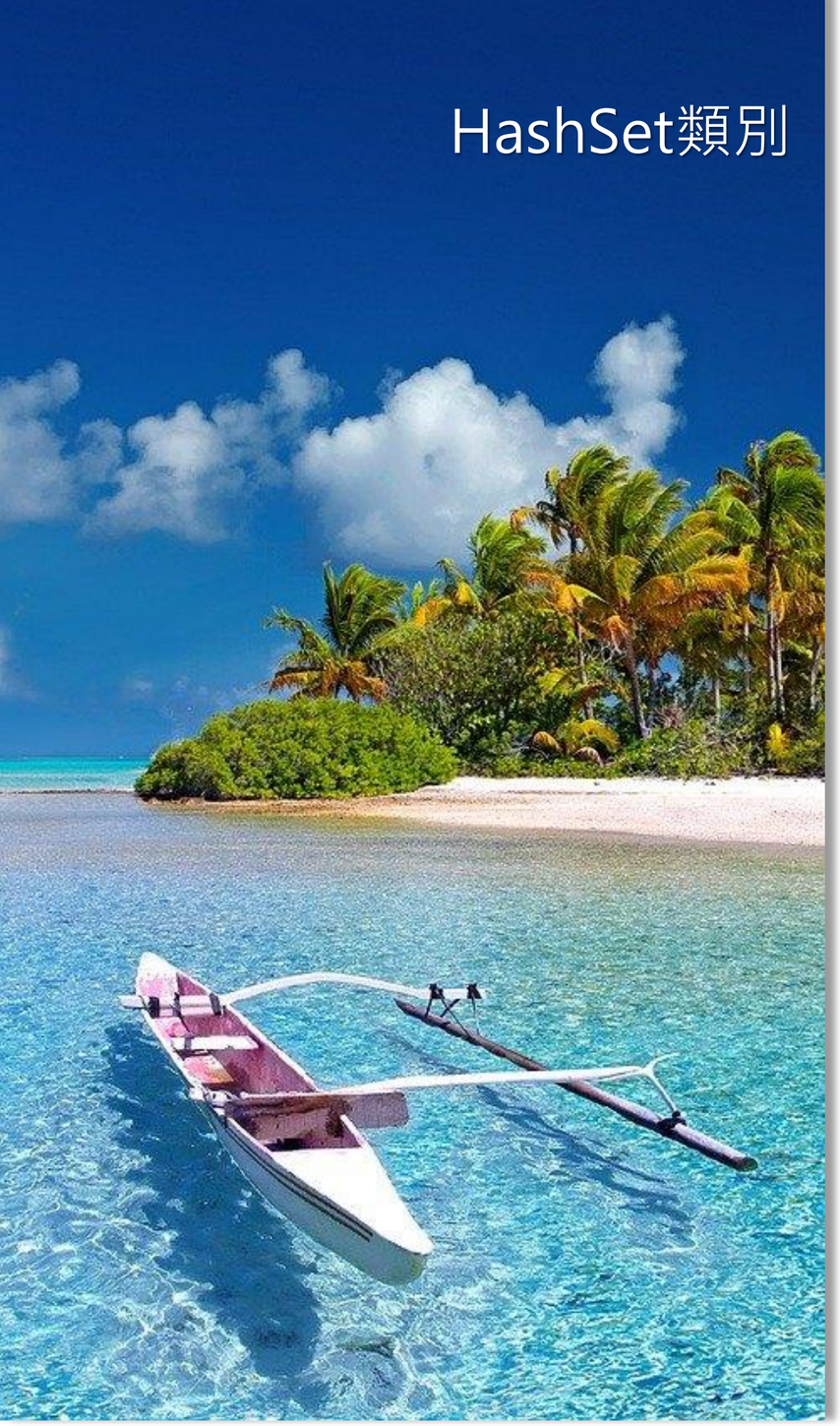
- 是沒有順序,且元素不能重覆
- 沒有順序,不等於隨機性,元素儲存時不是按照索引值增加,而是根據元素的hashcode而定
- 確保增加元素時利用equals方法判斷是否有重覆,若有則無法增加
- 沒有set(index,object)方法,因為其為無序

HashSet類別



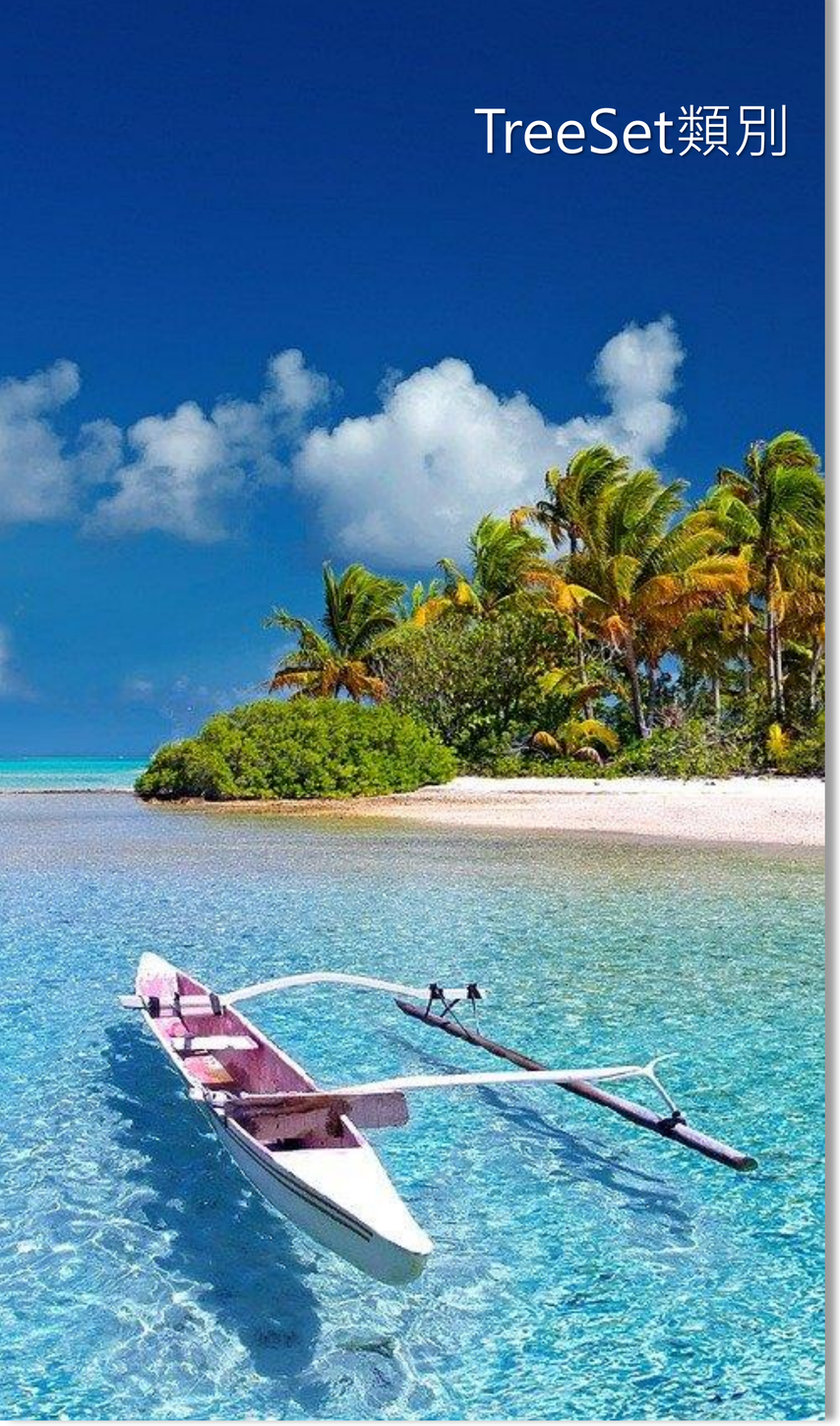
- 實作Set介面的類別
- 利用雜湊表演算法改進執行效率
- 儲存元素時，元素排列的順序和原先加入時的順序可能不同
- HashSet物件裡的元素都是唯一值，若重覆加入，會自動刪除

HashSet類別



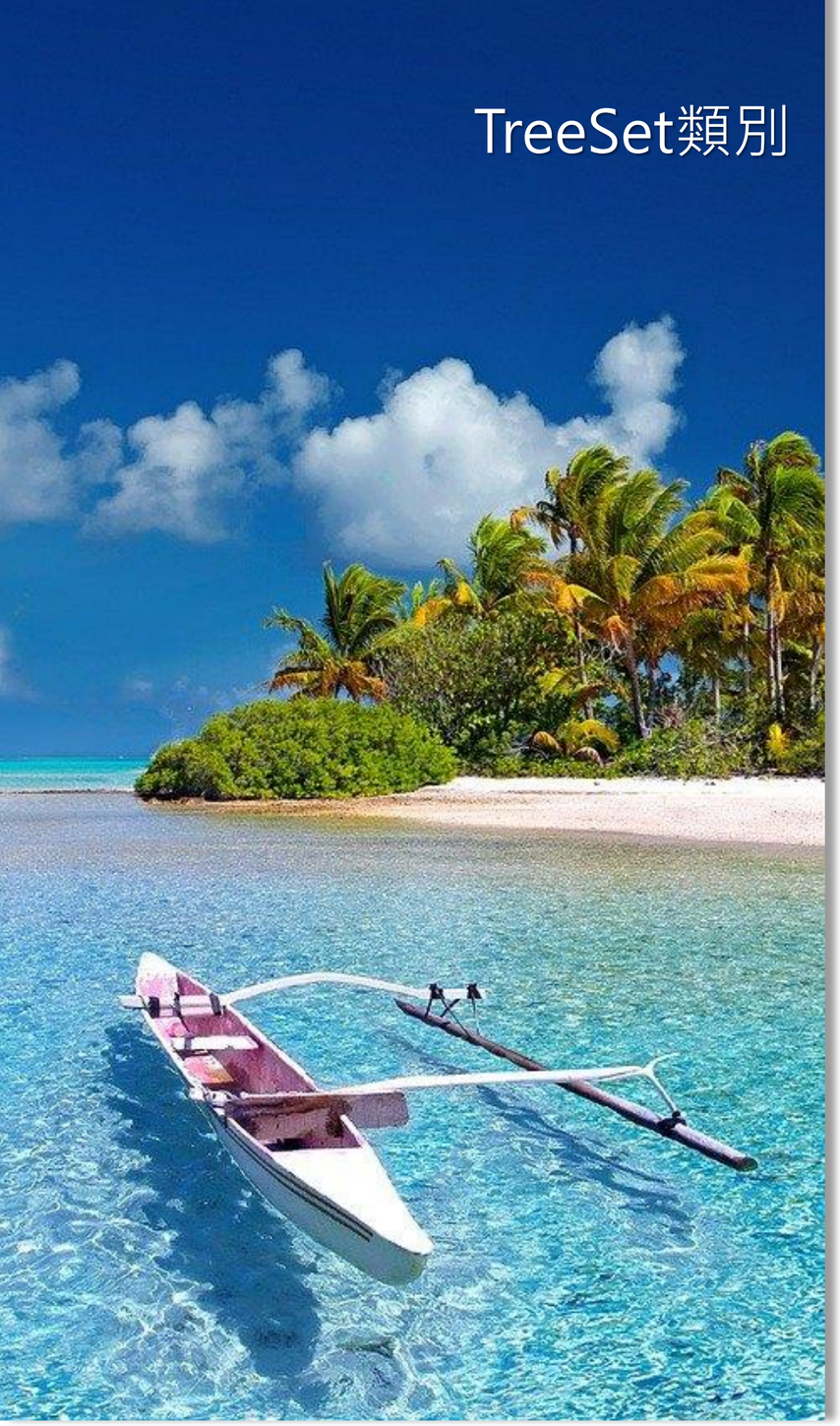
- HashSet<String> hs=new HashSet<String>();
- String str1="Dog";
- String str2="Cat";
- System.out.println("HashSet hs 是否為空值："+(hs.isEmpty()?"是":"否"));
- hs.add("Monkey");
- hs.add("Tiger");
- hs.add(str1);
- hs.add(str2);
- System.out.printf("HashSet hs 的元素共有: %d 個%n",hs.size());
- System.out.println("HashSet hs 是否為空值："+(hs.isEmpty()?"是":"否"));
- System.out.println("HashSet hs 的內容為："+hs);
- //利用Iterator的集合走訪器印出集合內容
- Iterator it=hs.iterator();
- String s=null;
- while(it.hasNext())
- {
- System.out.print(it.next()+"\t");
- }
- System.out.println();
- hs.remove(str2);
- System.out.printf("清除str 2後HashSet hs 的元素共有: %d 個%n",hs.size());
- System.out.println("HashSet hs 的內容為："+hs);
- System.out.println("HashSet hs 是否有包含str2："+(hs.contains(str2)?"是":"否"));
- System.out.println("HashSet hs 是否有包含Dog："+(hs.contains("Dog")?"是":"否"));
- System.out.println("HashSet hs 是否有包含str1："+(hs.contains(str1)?"是":"否"));
- hs.remove("Dog");
- System.out.println("HashSet hs 的內容為："+hs);

TreeSet類別



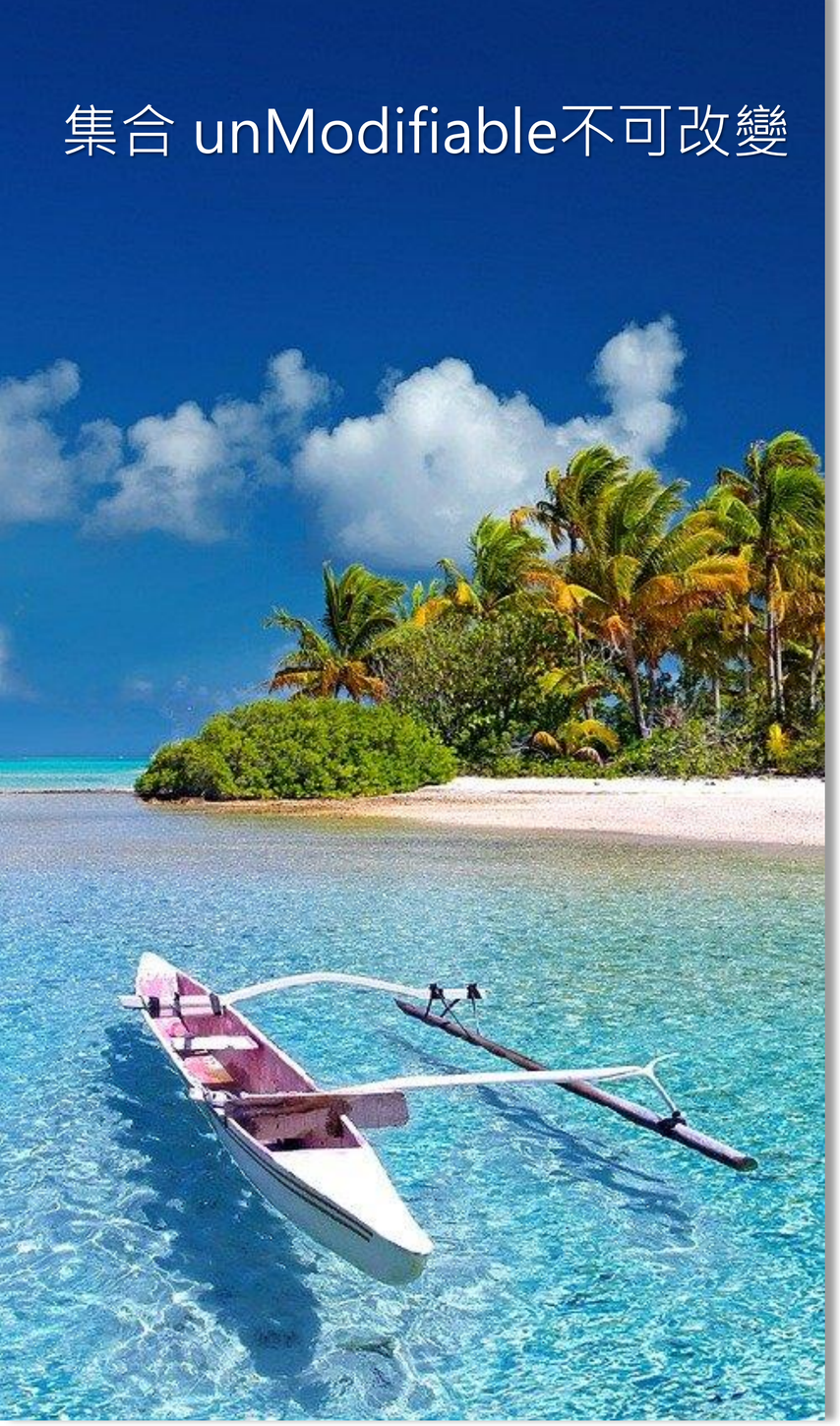
- 資料會自動由小到大排序
- 為一種排序集合（sorted collection）
- 資料不可重覆
- 常用方法
  - first( )　顯示集合中第一個值
  - last( )　顯示集合中最後一個值
  - subSet(start,end)　顯示集合中從多少到多少間的值
  - tailSet(n)　顯示大於等於n的所有值
  - headSet(n)　顯示小於n的所有值

TreeSet類別



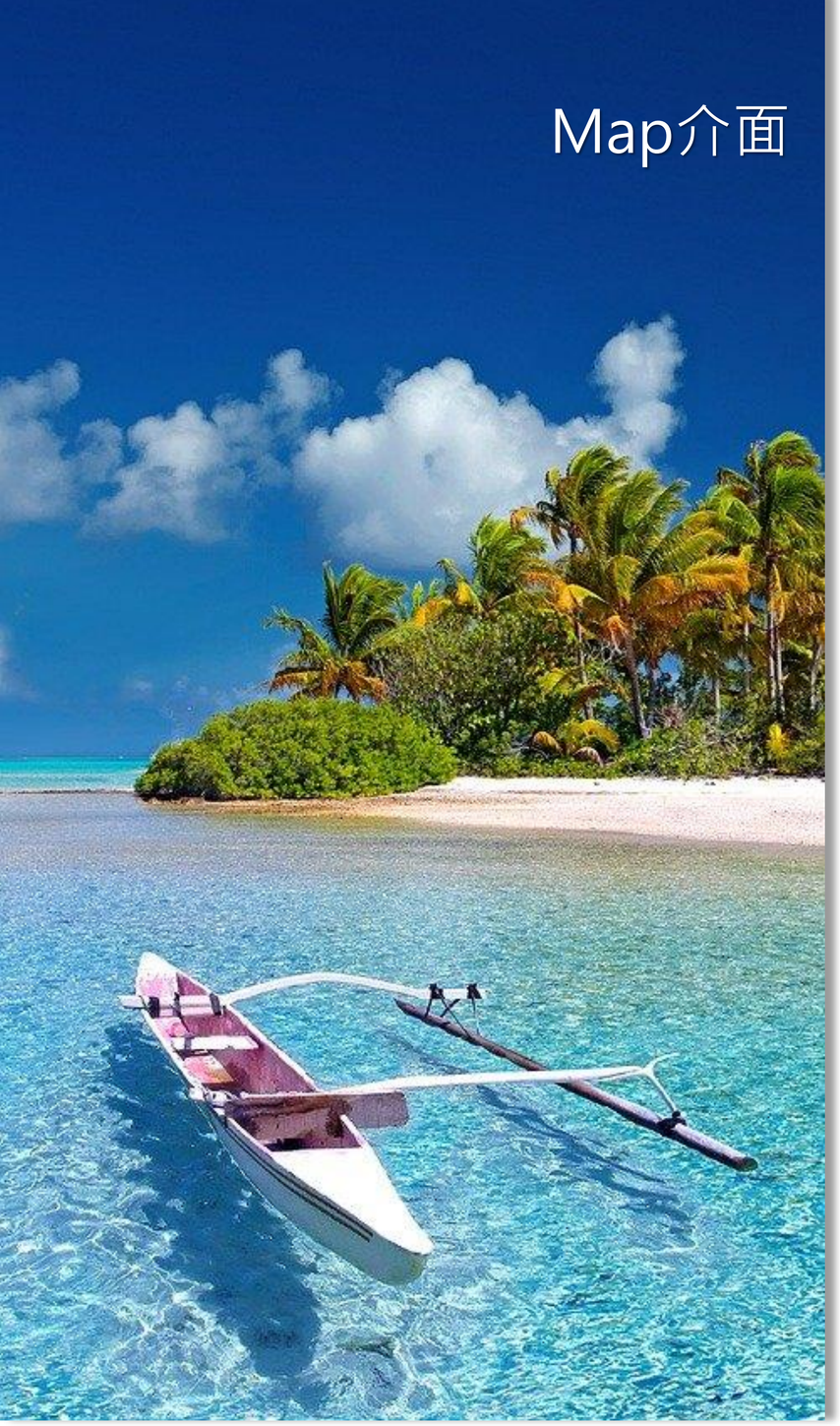
```
TreeSet<Integer> ts = new TreeSet<Integer>();
        for (int i = 1; i <= 10; i++) {
            int r = (int) (Math.random() * 100) + 1;
            System.out.print(r + "\t");
            ts.add(r);
        }
        System.out.println();
        System.out.println("ts內元素的個數：" + ts.size());
        System.out.println("ts集合內容為：" + ts);
        Iterator it = ts.iterator();
        while (it.hasNext()) {
            System.out.print(it.next() + "\t");
        }
        System.out.println();
        System.out.println("第一個元素為：" + ts.first());
        System.out.println("最後一個元素為：" + ts.last());
        System.out.println("介於20到60之間的集合為：" + ts.subSet(20, 60));//不包含60
        System.out.println("大於等於20的集合為：" + ts.tailSet(20));
        System.out.printf("大於等於20的集合共有 %d 個：%n", ts.tailSet(20).size());
        System.out.println("小於60的集合為：" + ts.headSet(60)); //不包含60

        System.out.printf("小於60的集合共有 %d 個：%n", ts.headSet(60).size());
```

集合 unModifiable不可改變



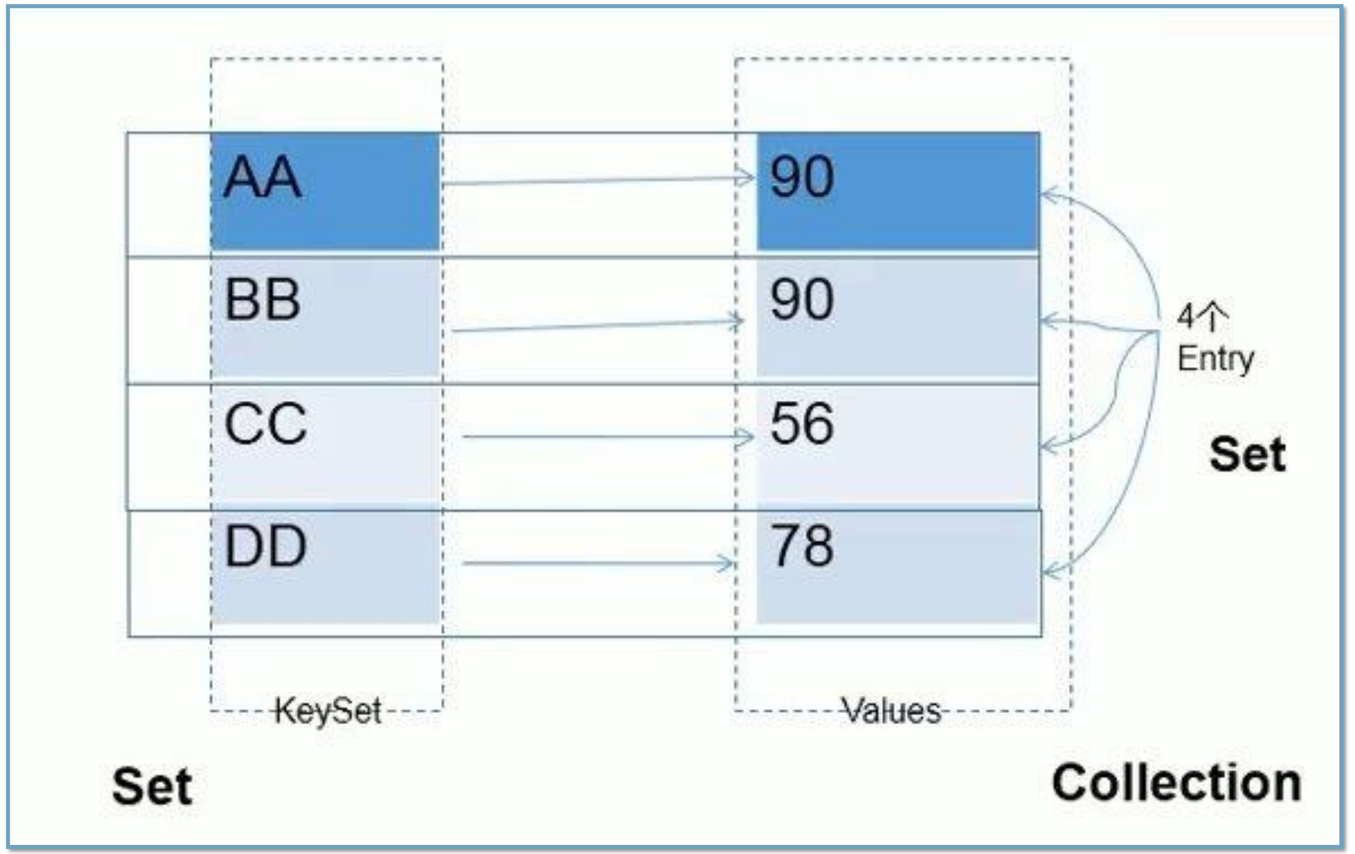
- 集合中產生集合後 只能讀不能修改、寫入的四種方法
  - Collections.unmodifiableList( ) 還有 Set,Collection
    - ArrayList<String> al=new ArrayList();
    - al.add("mike");
    - al.add("amy");
    - al.add("linda");
    - List<String> unModify = Collections.unmodifiableList(al);
    - unModify.set(2, "ok");//add及remove皆不行
  - Set.of( ) 或是 List.of( )方法
    - List<String> of = List.of("mike","amy","jacky");
    - of.add("Linda");
  - Arrays.asList( )方法
    - List<String> of = Arrays.asList("mike","amy","jacky");
    - of.add("Linda");
  - Set.CopyOf( ) 或是 List.CopyOf( )方法
    - 若是來源是new ArrayList( )則copy的目的與來源不是同一個物件,一個是可新增的,一個不可新增
    - 反之來源是 unmodifiable 則二個皆為不可新增的
      - ArrayList ls1 = new ArrayList();
      - ls1.add(1);
      - ls1.add(2);
      - ls1.add(3);
      - List copy1 = List.copyOf(ls1);
      - System.out.println(ls1==copy1);//false
      - ls1.add(4);
      - //copy1.add(5); 拋出 UnsupportedOperationException
      -
      - //複製不可改變的集合
      - List<Integer> ls2 = List.of(1,2,3,4,5);
      - List<Integer> copy2 = List.copyOf(ls2);
      - System.out.println(ls2==copy2);//true
      - //ls2.add(7); 拋出 UnsupportedOperationException
      - //copy2.add(8); 拋出 UnsupportedOperationException

Map介面



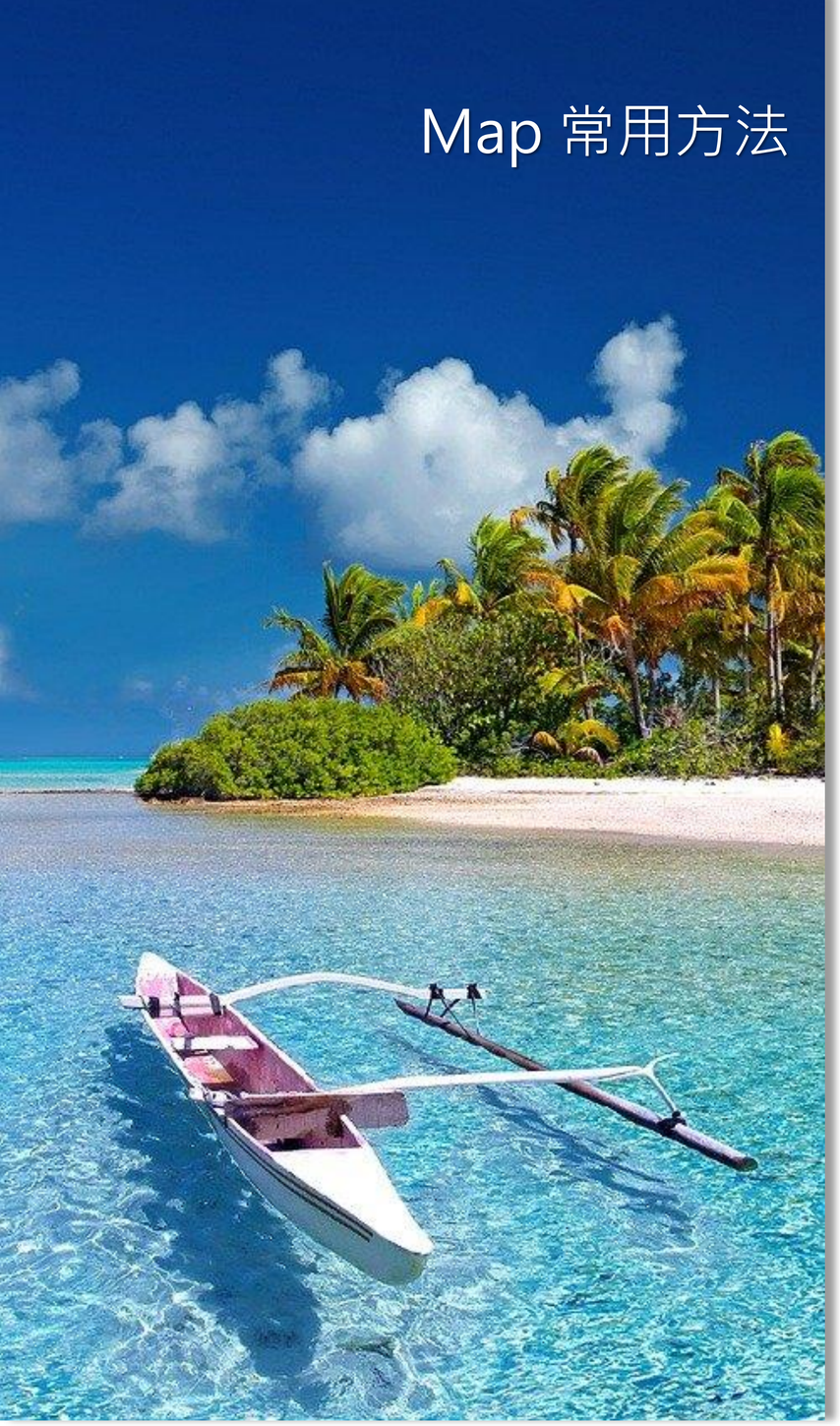
- 以關鍵值(key)儲存 ，函數 y=f(x)
- 關鍵值對應到的資料，即為對應值(value)
- 關鍵值不能重複，每一個鍵值最多只能對應一個元素
- 雖關鍵值不能重複，但卻可以儲存重複的對應值

Map介面



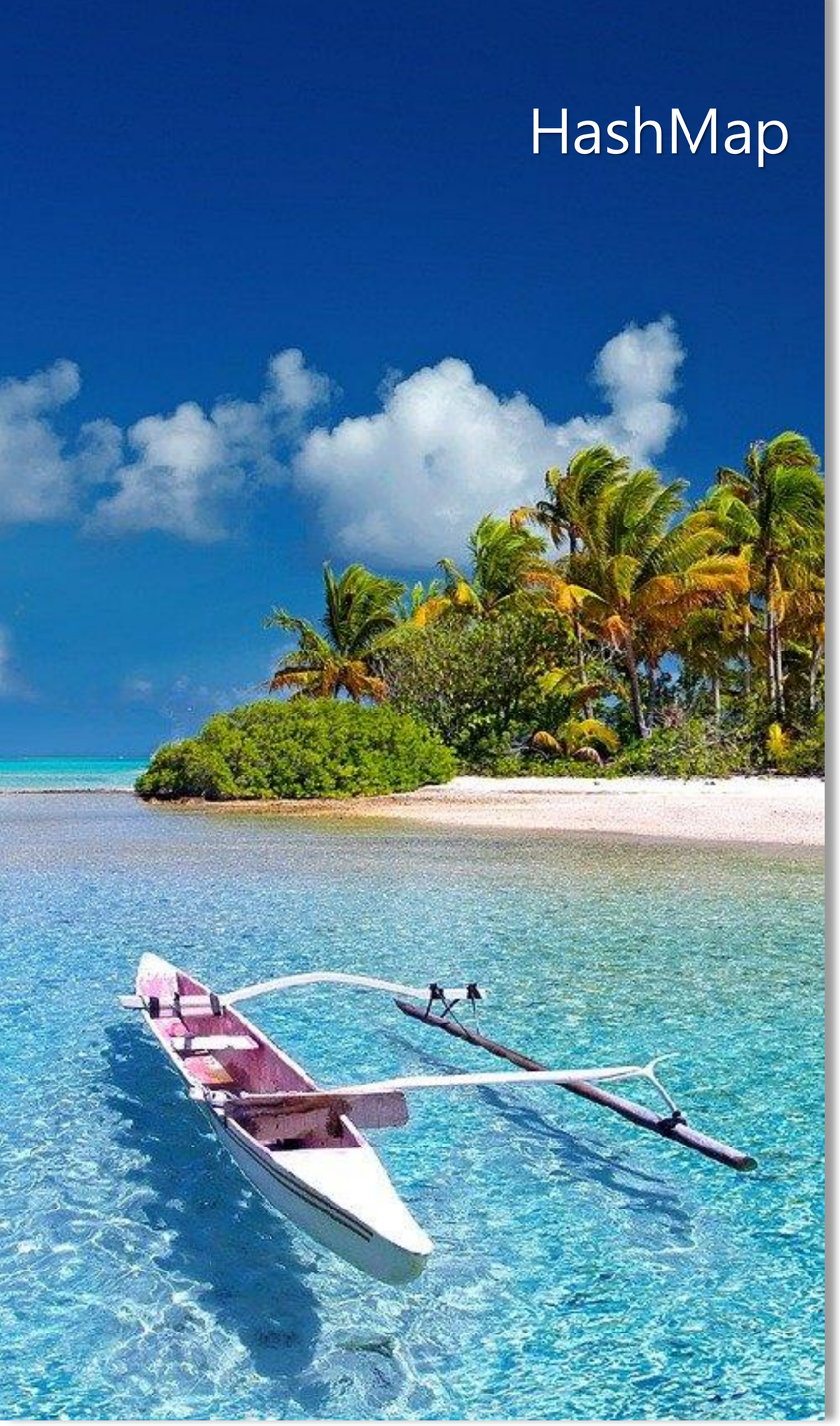
- KeySet 無序 不重覆
  - Values 重覆 無序
  - Entry 無序 不重覆

Map 常用方法



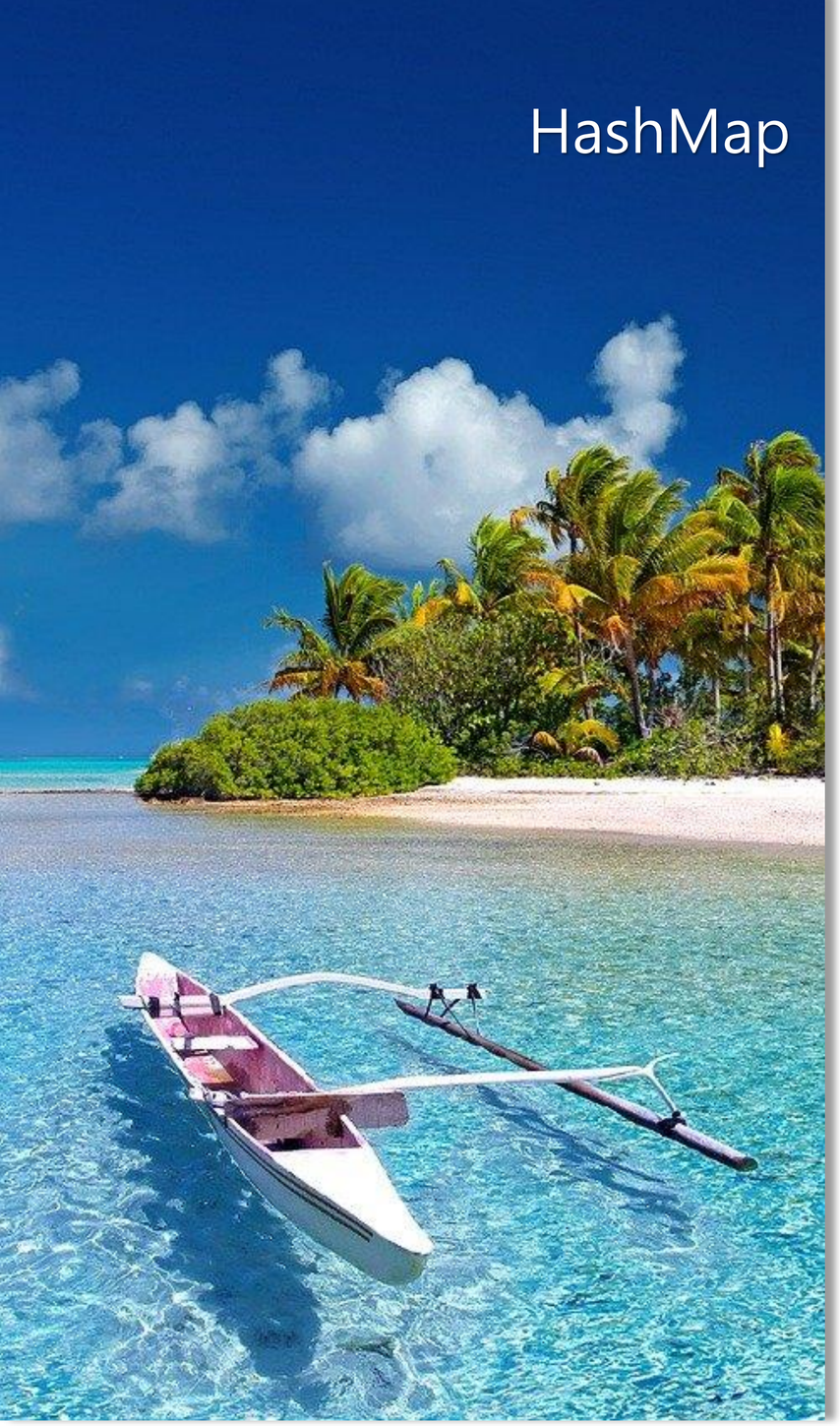
方法名稱	傳回值	說明
clear( )	void	清除Map所有資料
containsKey(Object key)	boolean	若指定的key物件存在於Map中，則會傳回true
containsValue(Object value)	boolean	若指定的value物件存在於Map中，則會傳回true
entrySet( )	set	將map以set型態傳回,其內容值為Entry類別(Map的內部類別)
getObject key)	Object	利用key指向Map中指定的元素並傳回
put(Object key,Object value)	Object	將所指定的key與value放入到Map中
putAll(Map m)	void	將所指定的m(Map物件）整份複製到Map中
remove(Object key)	Object	移除Map中key指向的元素
values( )	Collection	將Map中所包含的values以Collection型態傳回

HashMap



- HashMap map=new HashMap();
- map.put("A","SCJP");
- map.put("B",100);
- map.put(200,2000);
- map.put(new Object(),"SCBCD");
- map.put(null,null);
- map.put("B","HELLO");//此為修改
- System.out.println(map);
- System.out.println("Key A="+map.get("A"));
- System.out.println("Key 200="+map.get(200));
- System.out.println("Key C="+map.get("C"));//若沒有KEY值,則VALUE為NULL

- //利用foreach巡訪map
- for(Object x:map.keySet())
- System.out.println(x+"="+map.get(x));

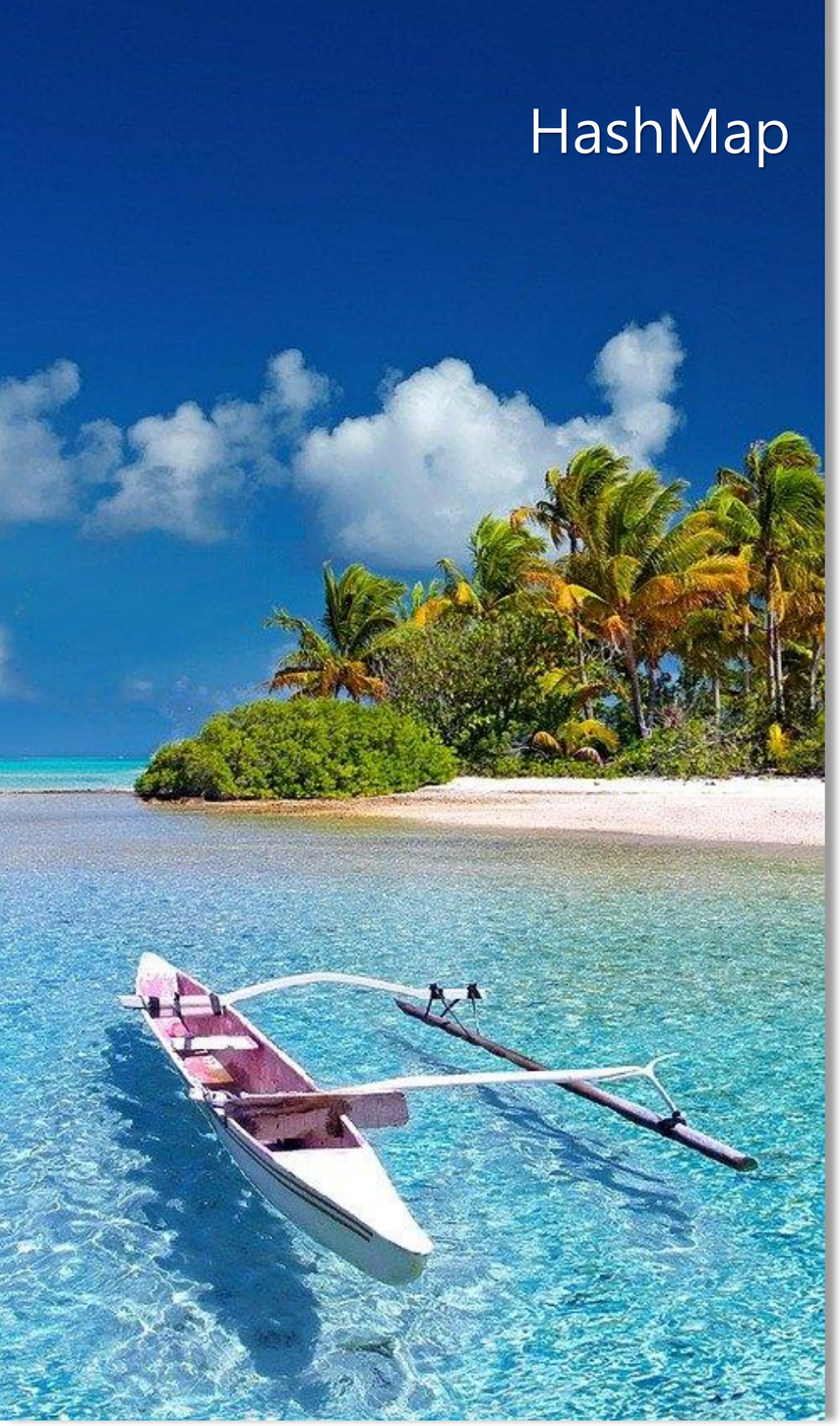


HashMap

使用Iterable forEach方法 其調用　BiConsumer 介面

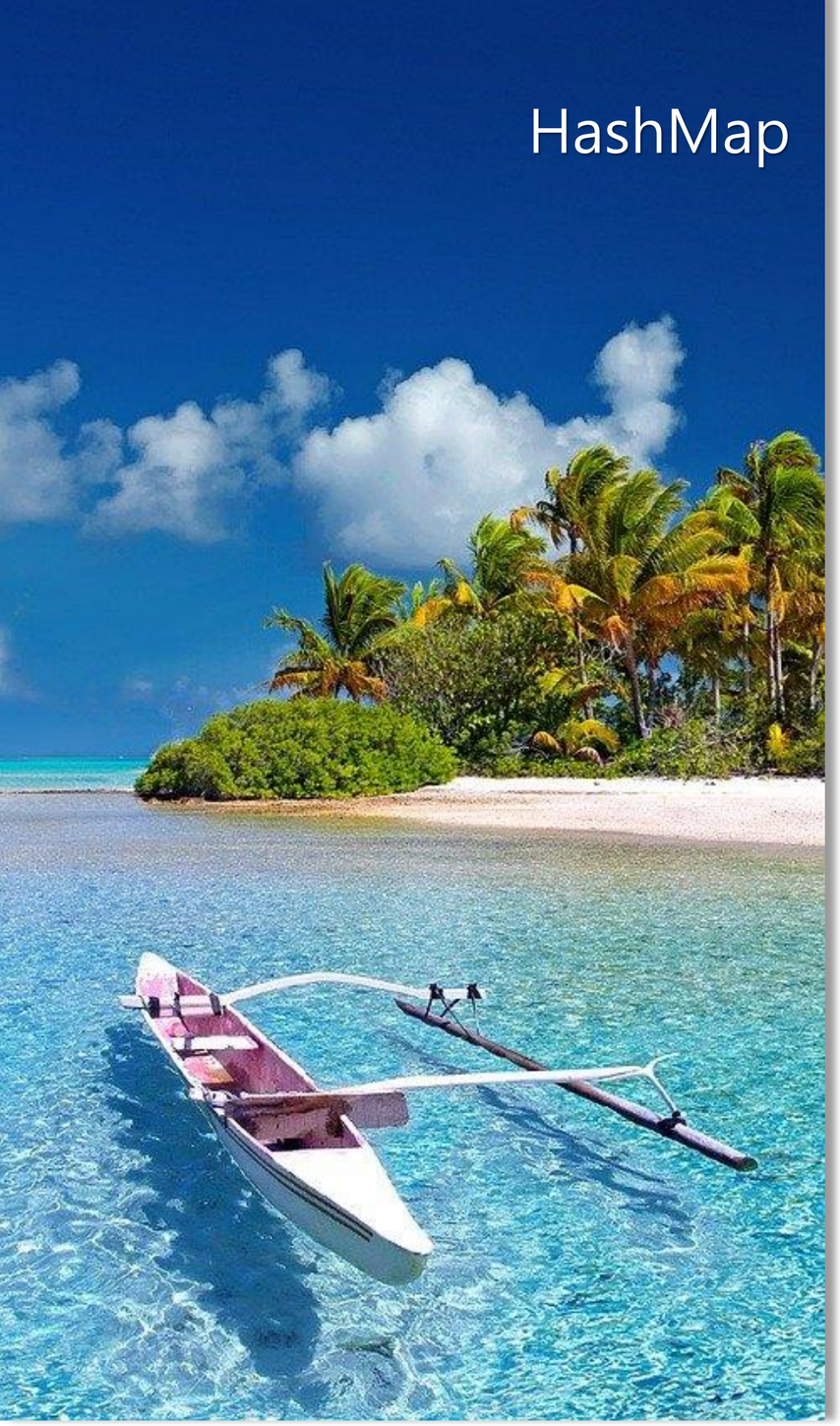
```
public class LambdaTest {
		public static void main(String[] args) {
		Map<Integer,String> map=new HashMap<Integer,String>();
		map.put(1,"Amy");
		map.put(3,"Mike");
		map.put(2,"Linda");
		map.put(4,"Jason");
		System.out.println(map);
		map.forEach(new BiConsumer<Integer,String>(){
				@Override
				public void accept(Integer t, String u) {
						System.out.println(t+"==>"+u);
				}
		});
		System.out.println("====Lambda表達式======");
		map.forEach((t,u)->System.out.println(t+"==>"+u));
		System.out.println("====方法參照======");
		map.forEach(LambdaTest::concatAndPrint);
}
		public static void concatAndPrint(Integer a,String b) {
				System.out.println(a+"==>"+b);
		}
}
```

HashMap



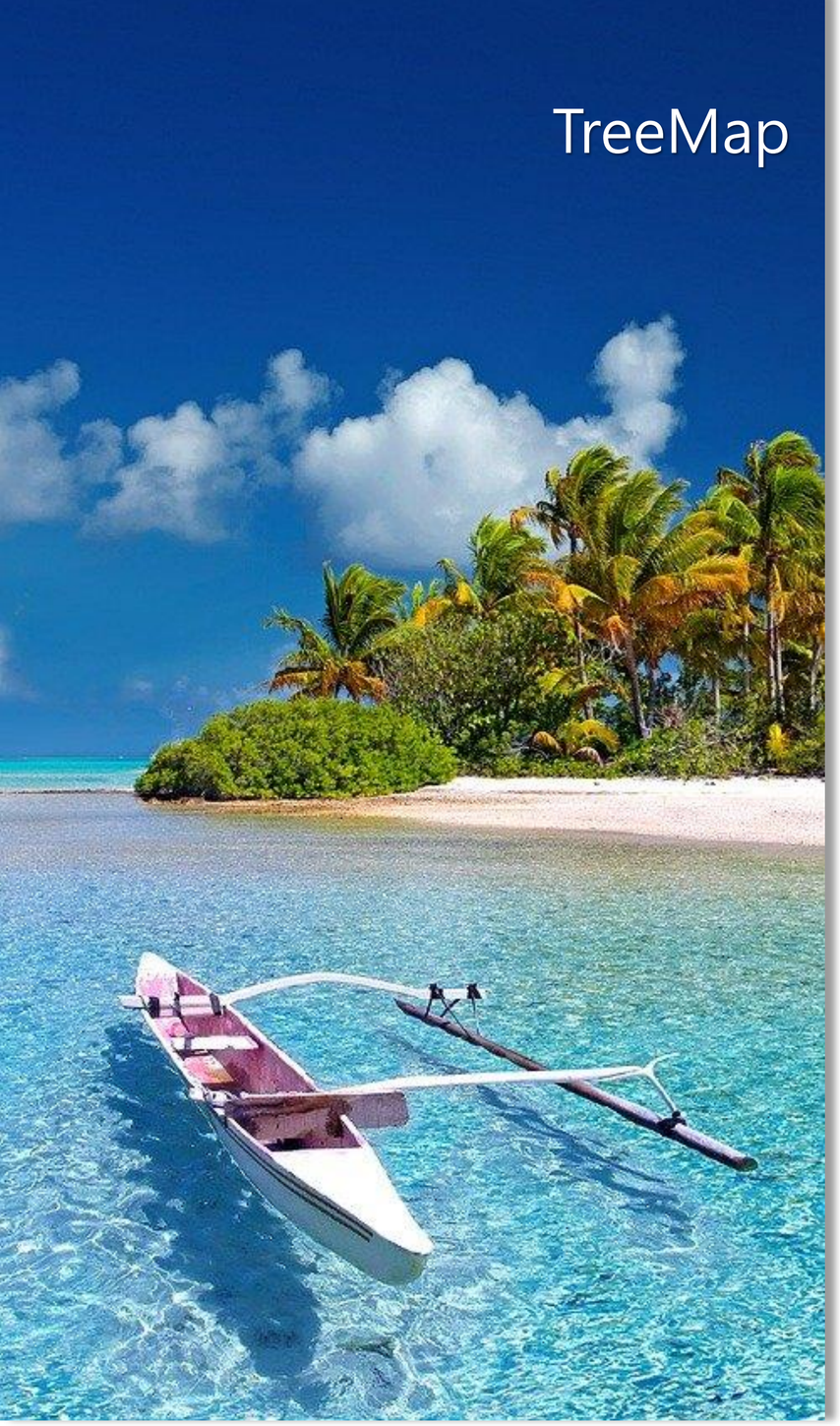
```
HashMap<Integer,String> map=new HashMap<Integer,String>();
        map.put(10001, "WORD");
        map.put(10002, "EXCEL");
        map.put(10004, "POWERPOINT");
        map.put(10008, "ACCESS");
        map.put(10005,"EXCEL");
        System.out.println("map的元素個數有："+map.size()+"個");
        System.out.println("map的元素如下：");
        System.out.println(map);
        System.out.println("map中是否有10002的Key值："+(map.containsKey(10002)?"是":"否"));
        System.out.println("map中是否有Access的value值："+(map.containsValue("Access")?"是":"否"));
        System.out.println("map中是否有ACCES的value值："+(map.containsValue("ACCESS")?"是":"否"));
        System.out.println("印出map中所有key值："+map.keySet());
        System.out.println("印出map中所有value值："+map.values());
        System.out.println("清除關鍵值為10002的對應值"+map.remove(10002));
        System.out.println(map);
 HashMap<Integer,String> map1=new HashMap<Integer,String>();
        map1.put(111, "111");
        map1.put(222, "222");
        map.putAll(map1);
        System.out.println(map);
```

HashMap



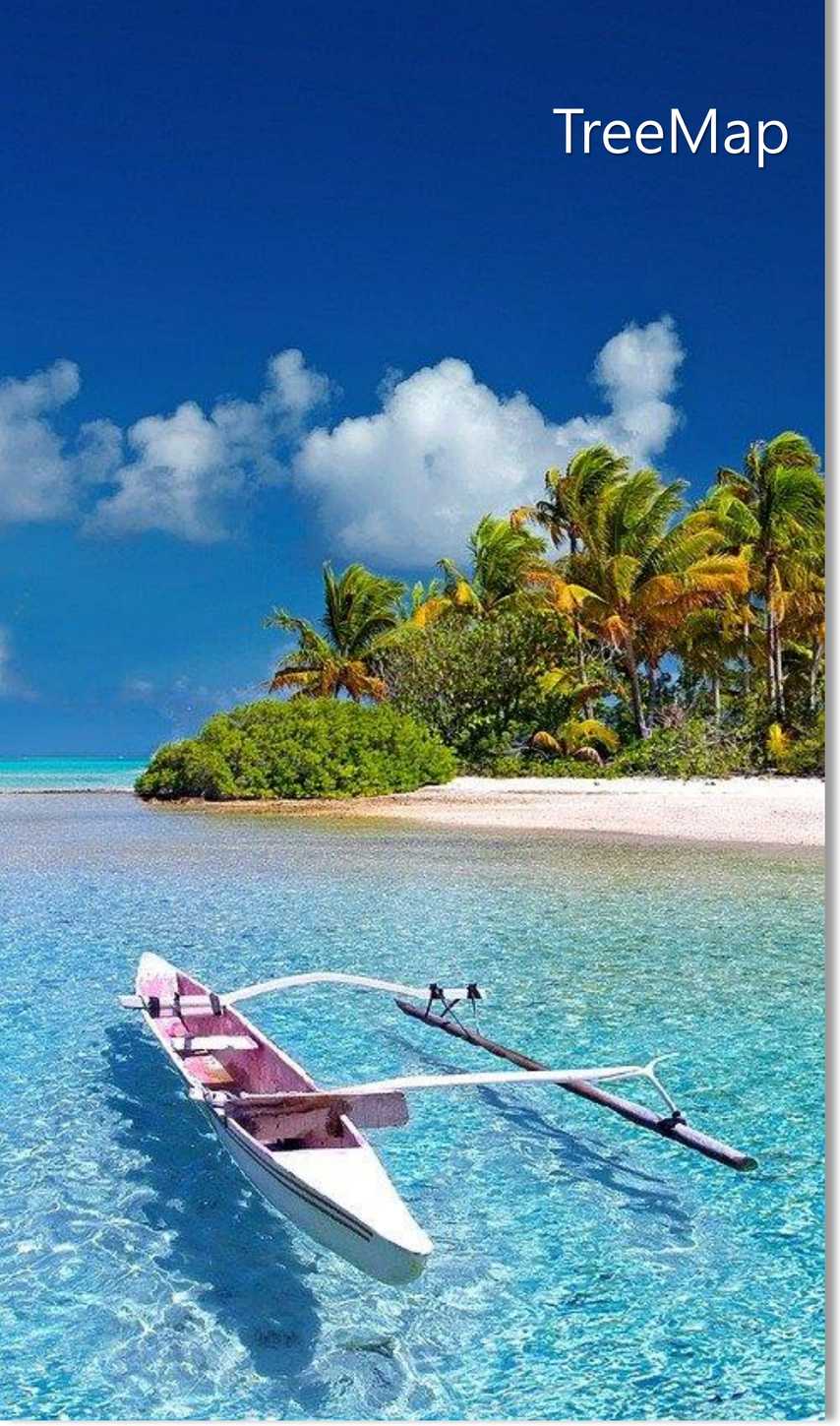
```
HashMap<Integer, String> map = new HashMap<Integer, String>();
        map.put(10001, "WORD");
        map.put(10002, "EXCEL");
        map.put(10004, "POWERPOINT");
        map.put(10008, "ACCESS");
        map.put(10005, "EXCEL2020");
        //巡訪所有key值
        System.out.println("=====巡訪所有key值 ========");
        Set<Integer> keySet = map.keySet();
        Iterator<Integer> iterator = keySet.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
        System.out.println("=====巡訪所有value值 ========");
        Collection<String> values = map.values();
        Iterator<String> iterator1 = values.iterator();
        while (iterator1.hasNext()) {
            System.out.println(iterator1.next());
        }
System.out.println("=====巡訪所有key-value值 ========");
        Set<Map.Entry<Integer, String>> entrySet = map.entrySet();
        Iterator<Map.Entry<Integer, String>> iterator2 = entrySet.iterator();
        while(iterator2.hasNext())
        {
            Object obj=iterator2.next();
            Map.Entry entry=(Map.Entry)obj;
            System.out.print(entry.getKey());
            System.out.print("對應到");
            System.out.print(entry.getValue());
            System.out.println();
             //上述程式可改為 sout(itator2.next());
        }
```

TreeMap



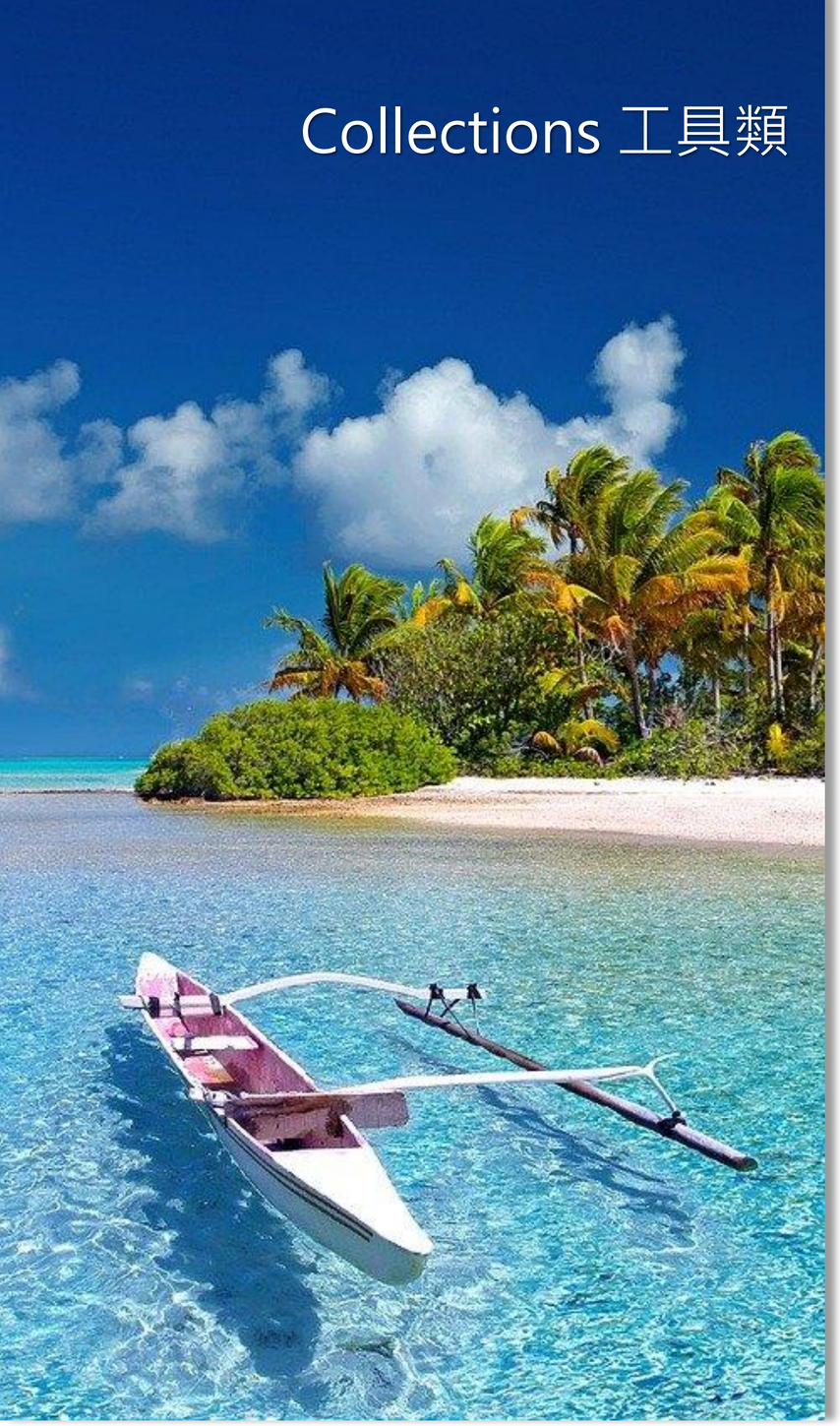
- 具有排序性
- 所置入的關鍵值必須是同一種資料型別，否則在執行時期會產生錯誤

TreeMap



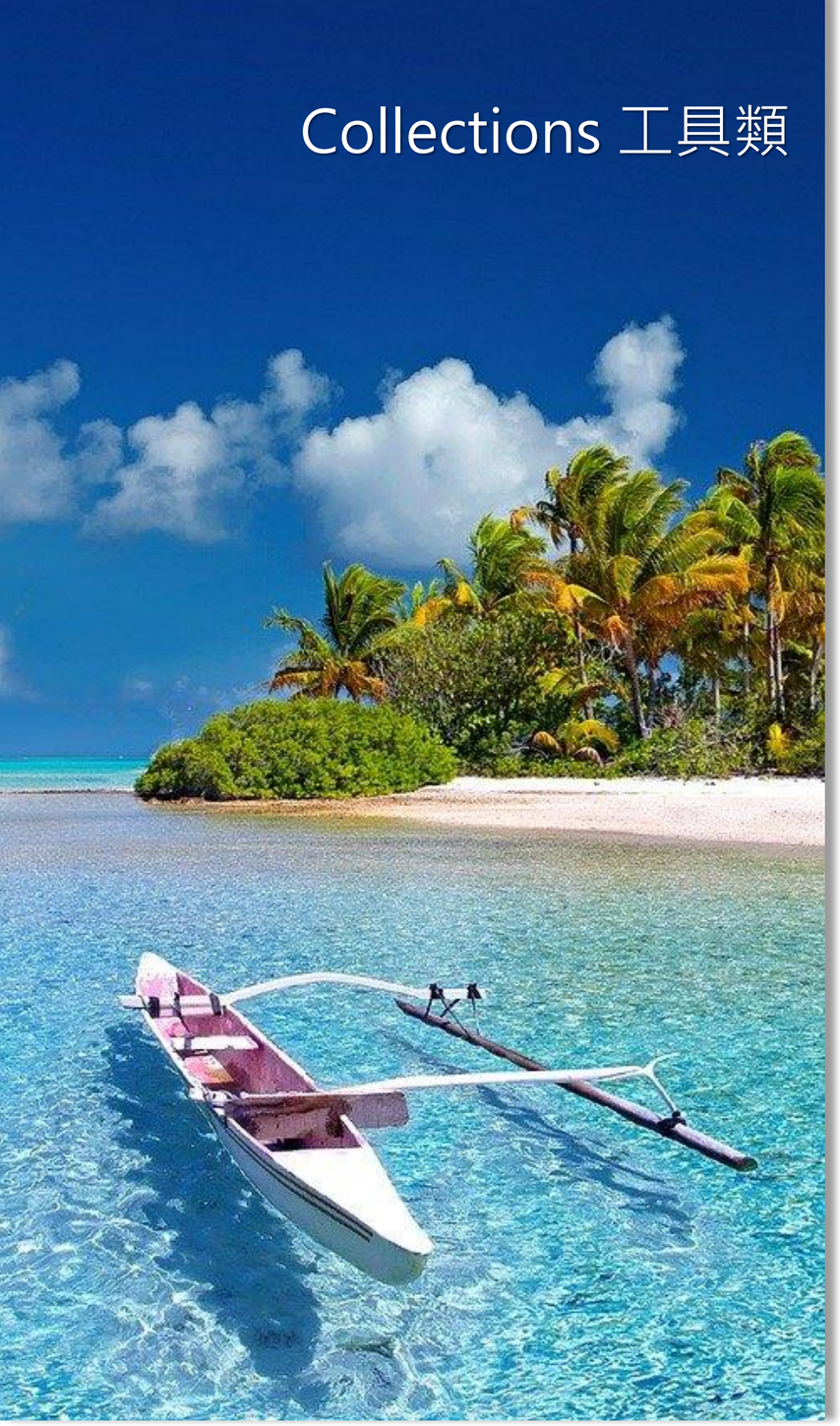
```
TreeMap<Integer,String> map=new TreeMap<Integer,String>();
        map.put(10001, "WORD");
        map.put(10002, "EXCEL");
        map.put(10004, "POWERPOINT");
        map.put(10008, "ACCESS");
        map.put(10005,"EXCEL");
        System.out.println("map的第一個元素是："+map.get(map.firstKey()));
        System.out.println("map的最後一個元素是："+map.get(map.lastKey()));
        System.out.println("map中從10002到10005的元素有"+map.subMap(10002,10005));//不包含1005
        System.out.println("map的元素個數有："+map.size()+"個");
        System.out.println("map的元素如下：");
        System.out.println(map);
        System.out.println("map中是否有10002的Key值："+(map.containsKey(10002)?"是":"否"));
        System.out.println("map中是否有10002的Key值："+(map.containsValue("Access")?"是":"否"));
        System.out.println("map中是否有10002的Key值："+(map.containsValue("ACCESS")?"是":"否"));
        System.out.println("印出map中所有key值："+map.keySet());
        System.out.println("印出map中所有value值："+map.values());
        System.out.println("清除關鍵值為10002的對應值"+map.remove(10002));
        System.out.println(map);
```

Collections 工具類



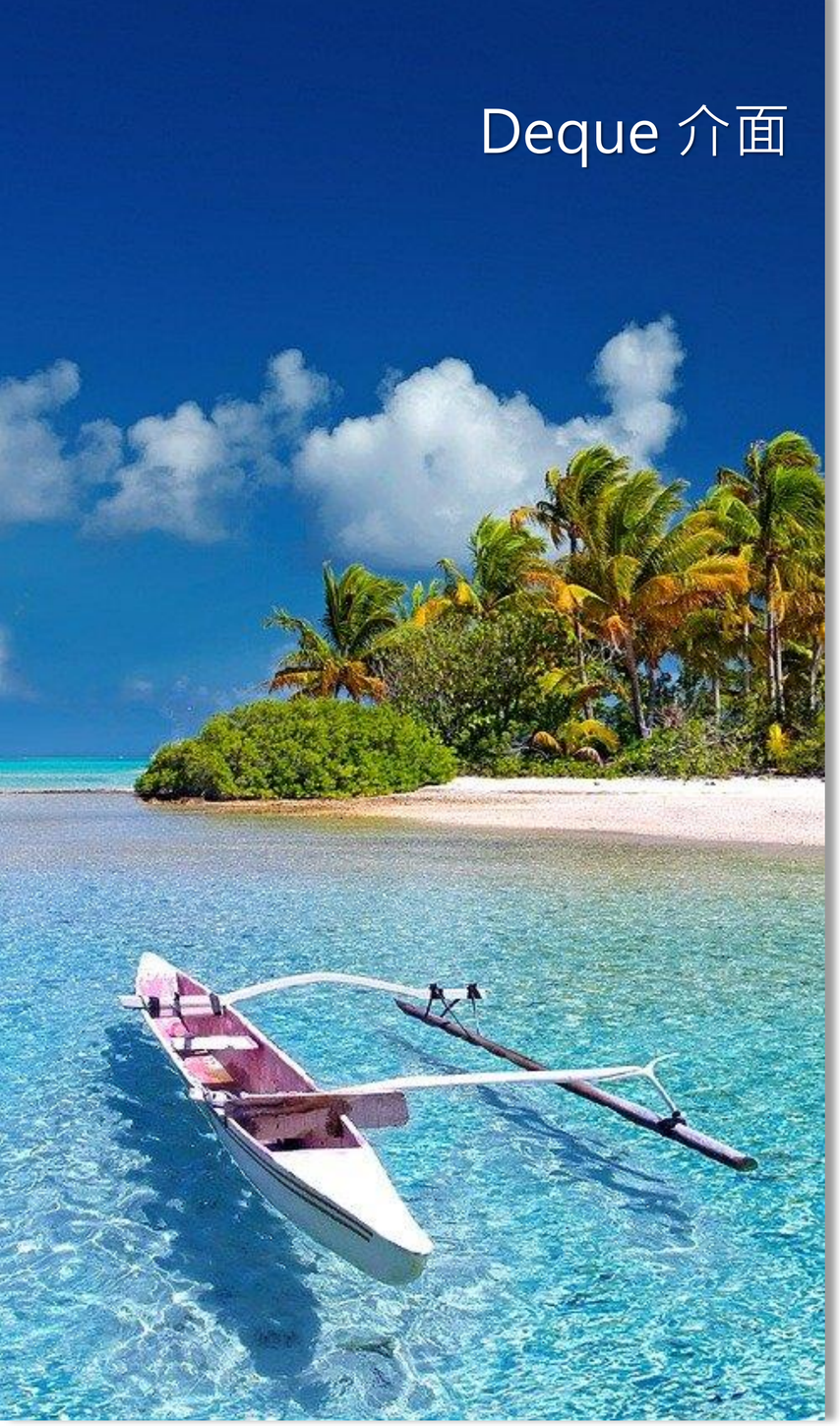
- Collections是一個操作Set List Map等集合的工具
- Collections中提供了一系列靜態的方法對集合元素進行排序,查詢和修改等操作.
- 常用方法
  - reverse( ) 反轉內容
  - shuffle( )隨機排序
  - sort( )由小到大排序
  - sort(list,Comparator)自訂排序
  - swap(list,int ,int)指定集合中二元素交換位置
  - frequency(Collection,Object)返回集合中指定元素出現的次數
  - copy(List dest,List src)將src集合中的元素複制到dest集合中

Collections 工具類

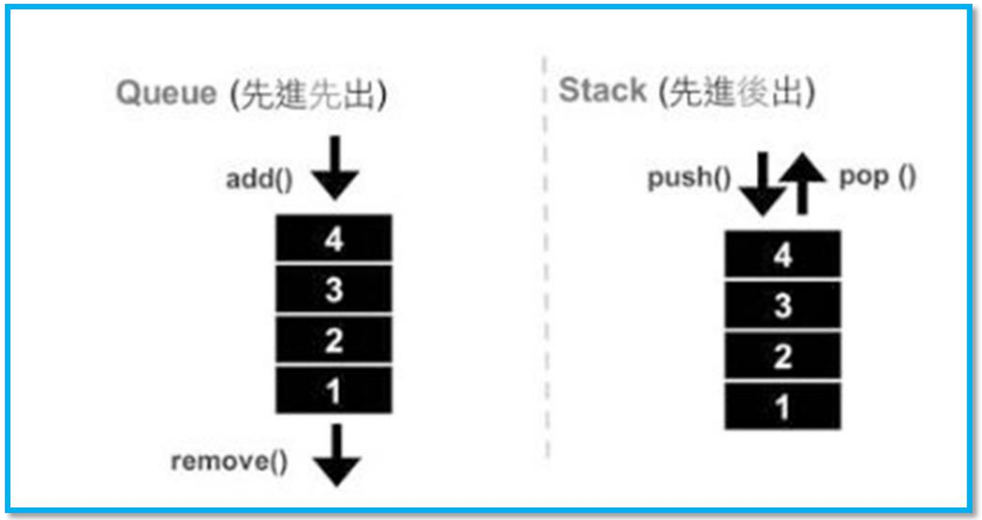


```
List list = Arrays.asList(22, 33, 55, 11, 99, 34, 88, 66);
        System.out.println(list);
        Collections.reverse(list);
        System.out.println(list);
        Collections.shuffle(list);
        System.out.println(list);
        Collections.sort(list);
        System.out.println(list);
        Collections.sort(list, new Comparator<Integer>() {
            @Override
            public int compare(Integer o1, Integer o2) {
                return Integer.compare(o2, o1);
                //return o2-o1;
            }
        });
        System.out.println(list);
        Collections.swap(list, 1, 2);
        System.out.println(list);

System.out.println(Collections.frequency(list, 34));
        //錯誤方式為List dest=new ArrayList() size為0
        List dest = Arrays.asList(new Object[list.size()]);
        Collections.copy(dest, list);
        System.out.println(dest);
```

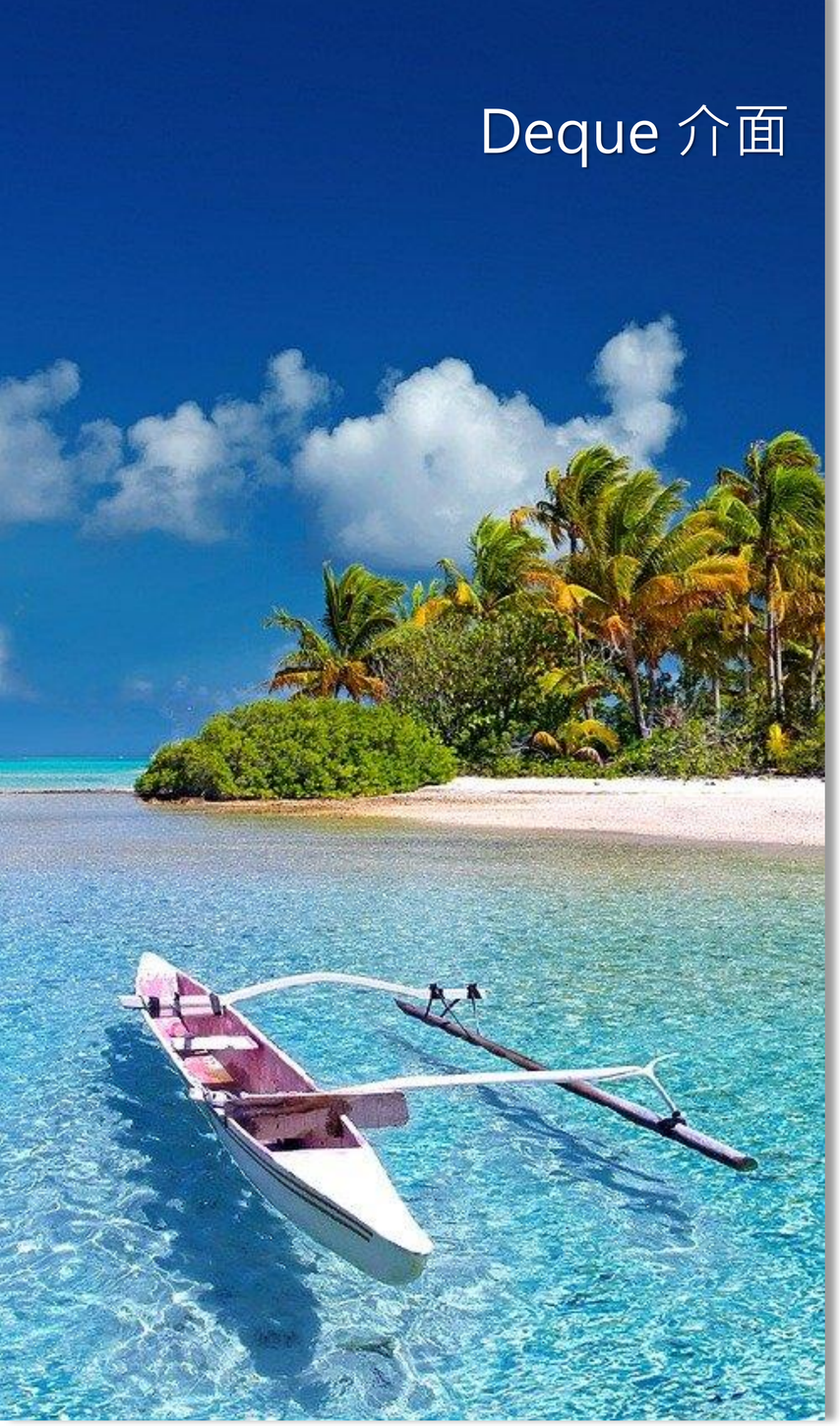
Deque 介面



- 當使用add和remove方法時,為queue行為模式,先進先出
- 使用push和pop方法時,為stack行為模式,先進後出



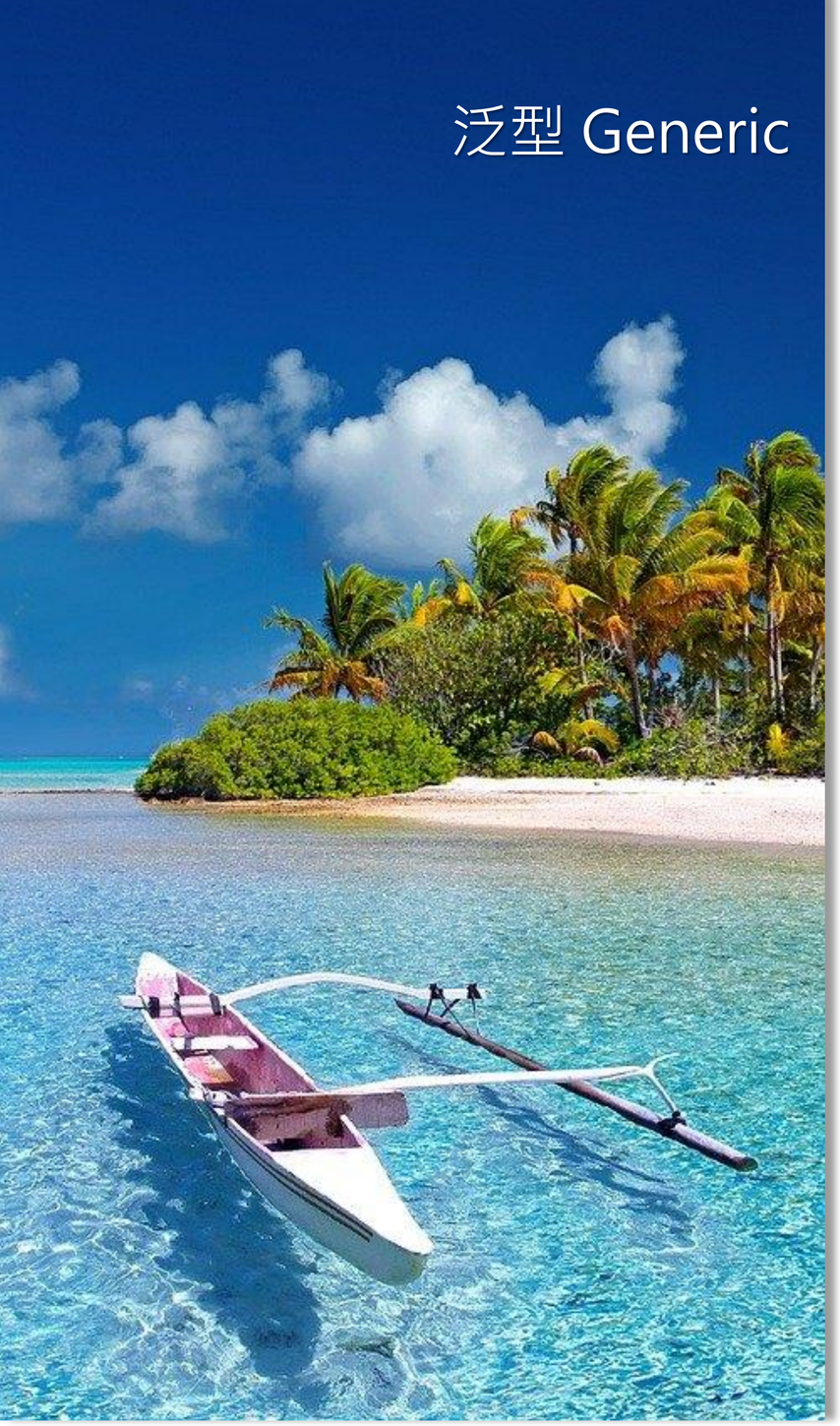
Deque 介面



```
//STACK
Deque stack=new ArrayDeque();
    stack.push("one");
    stack.push("two");
    stack.push("three");
    System.out.println(stack.pop());
//若是將pop改為peek則不會刪除,那麼內容皆存在,執行結果為three three
            three.[three,two,one]
      System.out.println(stack.pop());
      System.out.println(stack.pop());
      System.out.println(stack);

  //QUEUE
   Deque stack = new ArrayDeque();
      stack.add("one");
      stack.add("two");
      stack.add("three");
      System.out.println(stack.remove());
      System.out.println(stack.remove());
      System.out.println(stack.remove());
      System.out.println(stack);
```

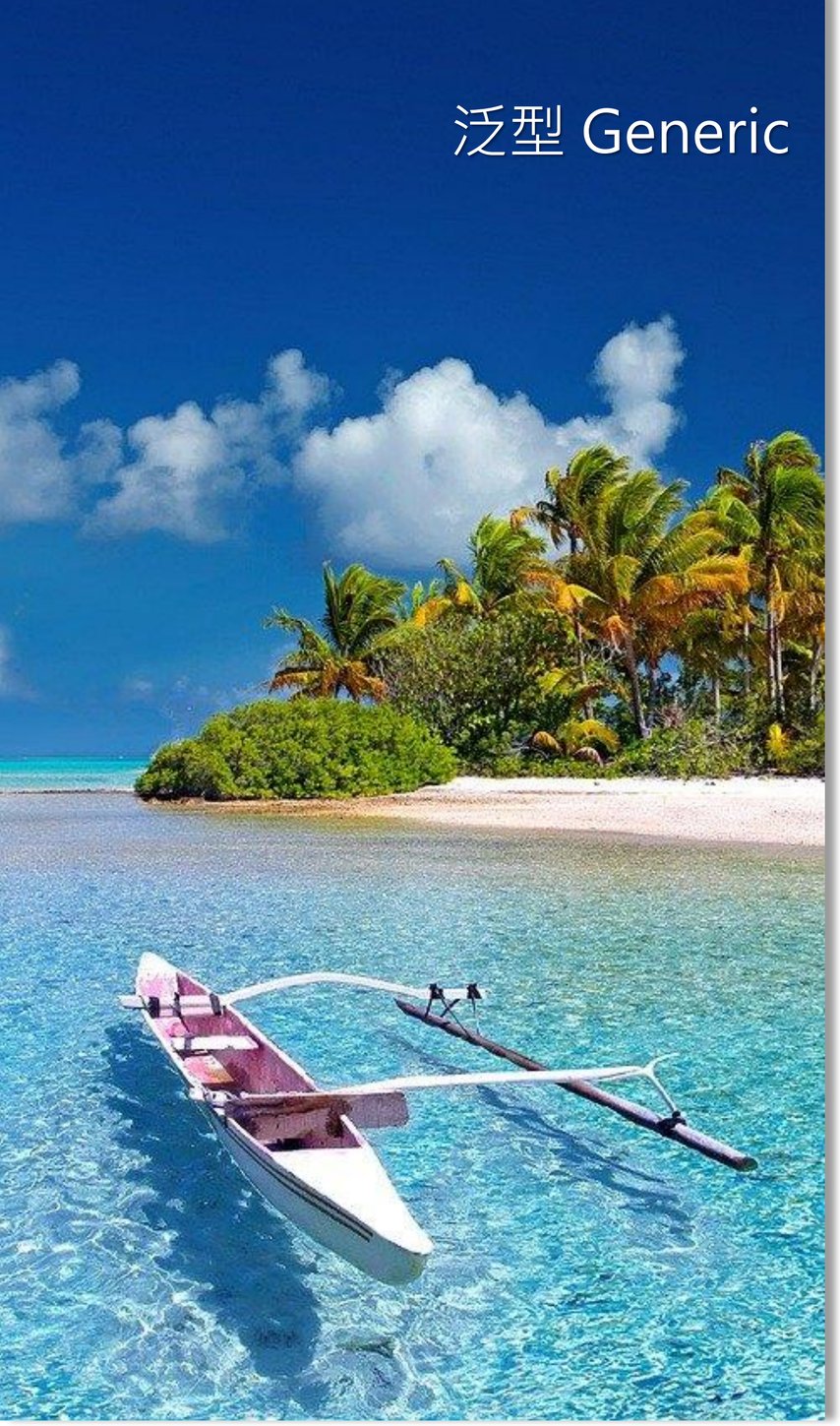
泛型 GENERIC

泛型 Generic



- 泛型:盒子標籤(明確的知道內容是什麼)如資源回收垃圾筒
- 泛型:把元素的類型設計成一個參數,這個類型的參數即為泛型
- 集合中若沒有泛型,任何類型皆可添加到集合中,類型不安全,讀取時還需要強制轉型,程式繁瑣,若是使用泛型則不需要.
- 靜態方法與例外不能使用泛型
  - class MyException<T> extends Exception泛型無法套用於例外
  - public static void show(T var){sout(T變數)} 因泛型作用於物件產生時,但類別方法先於物件,即泛型無法套用於類別屬性上
  - 建構子方法簽章處不能使用泛型Person<T>(){}//錯誤
- 泛型不同的引用不能相互賦值(Person<Integer> p,Person<String> p1,p1=p;

泛型 Generic



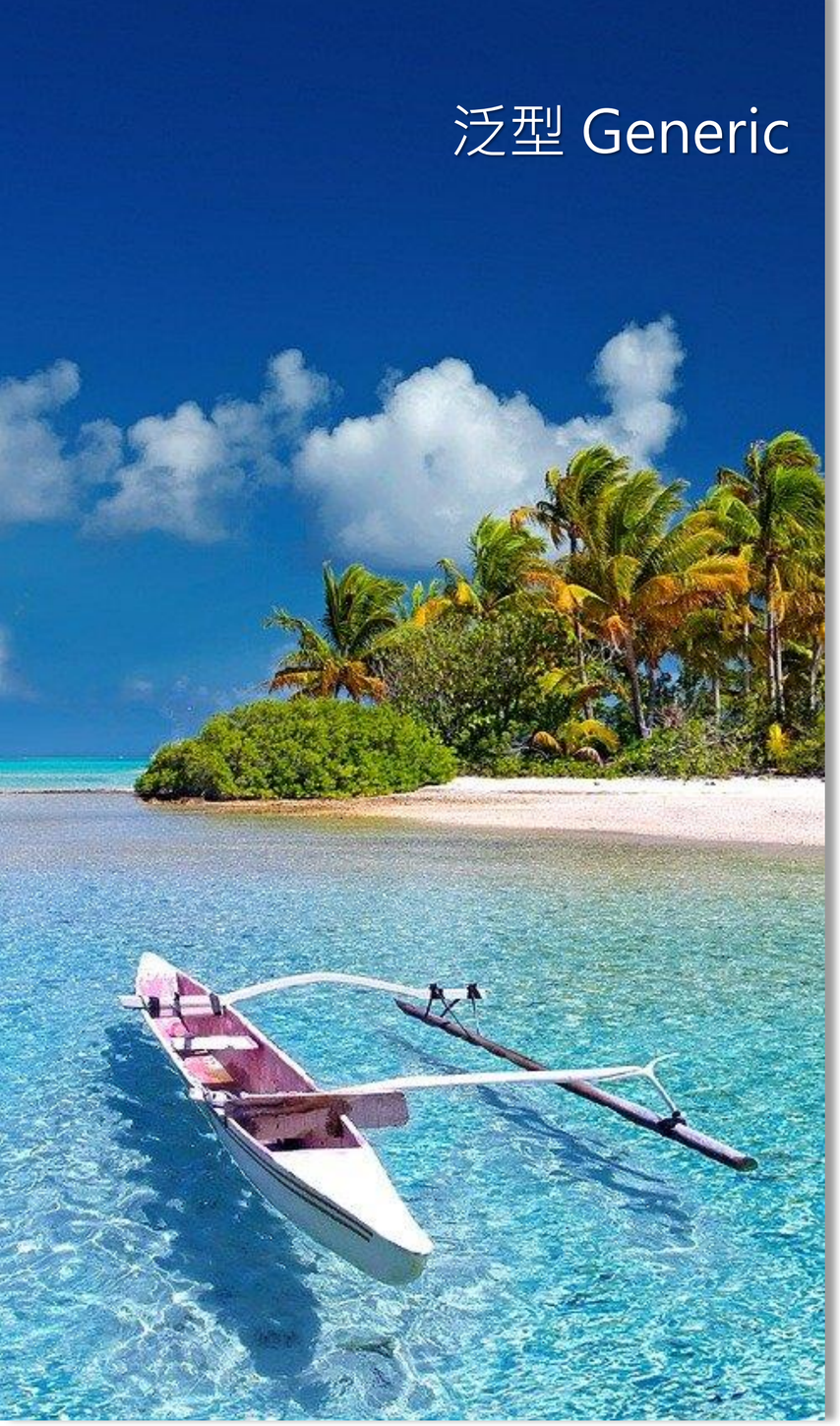
```
class Shirt{
    private Shirt shirt;
  public void add(Shirt shirt) {
      this.shirt=shirt;
  }
  public Shirt get(){
      return this.shirt;
  }
}
```

```
class UseString{
    private String message="";
  public void add(String messge)
  {
      this.message=message;
  }
  public String get(){
      return this.message;
  }
}
```

```
class UseAny<T>{
    private T t;
    public void add(T t)
    {
        this.t=t;
    }
    public T get(){
        return this.t;
    }
}
```

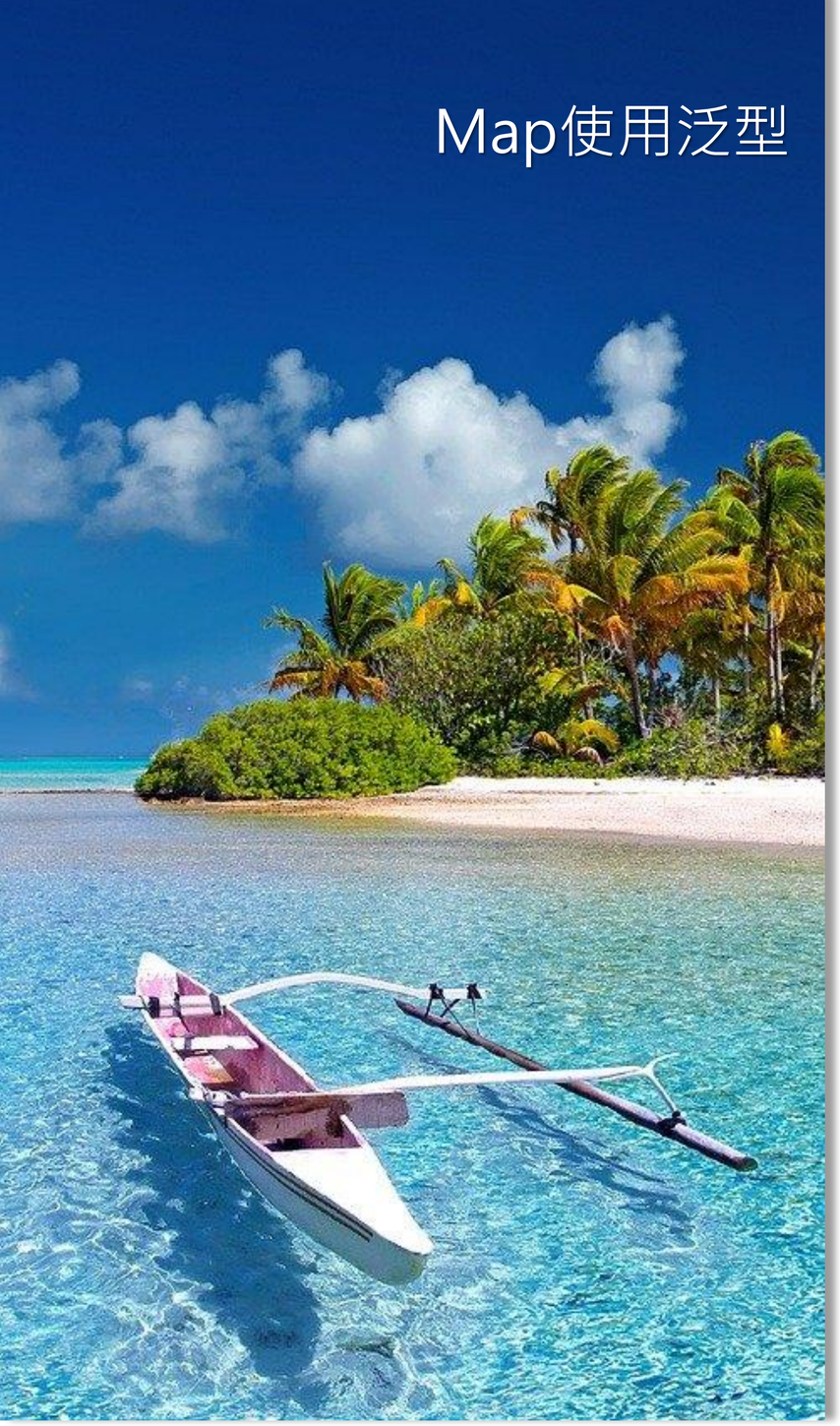
若將程式碼裡使用的符號T 置換為String 就會是範例一的類別,若置換為Shirt就是範例二的類別,符號T可以廣泛的被各種參考型別取代,就是泛型設計的概念

泛型 Generic



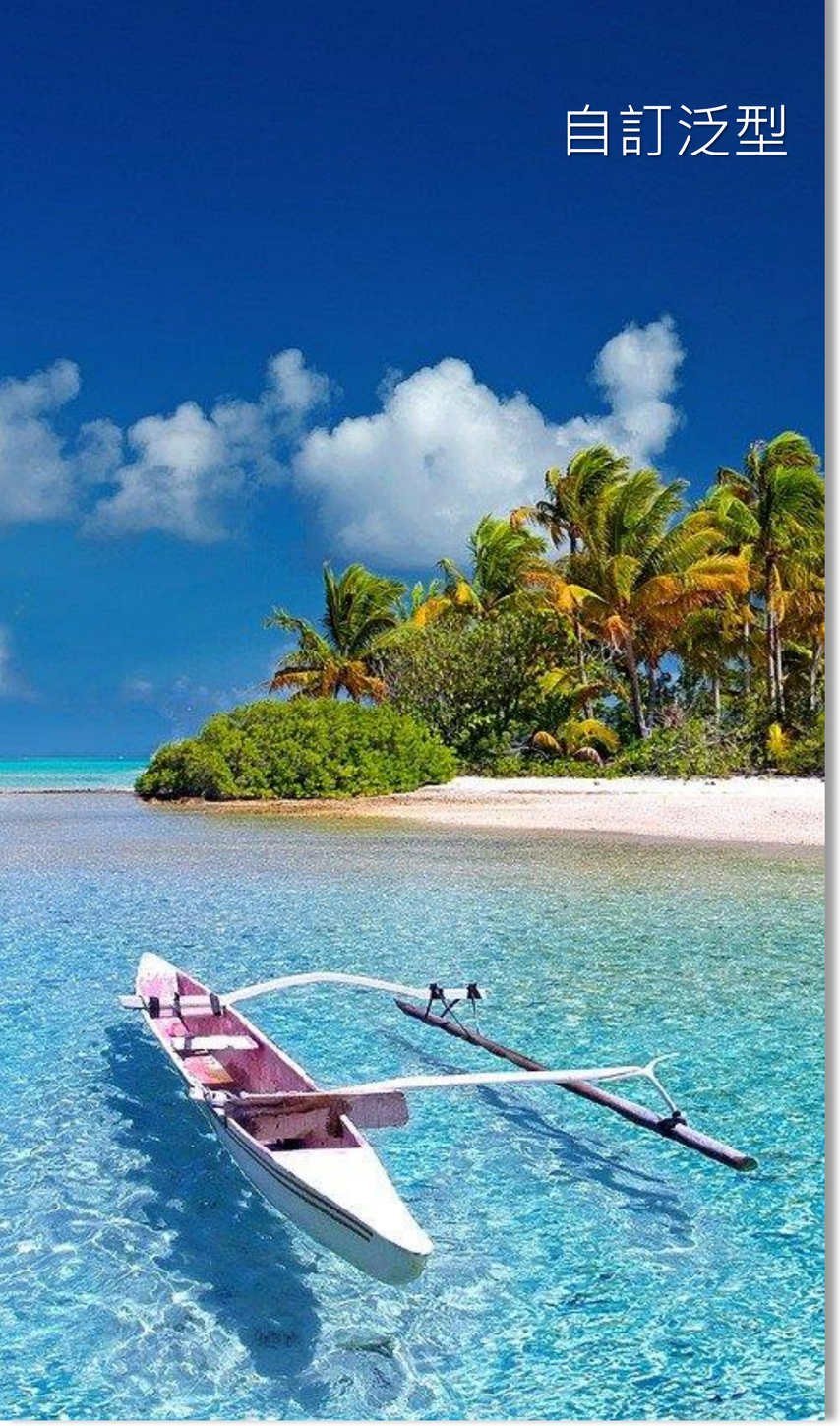
```
//未使用泛型
ArrayList aList = new ArrayList();
//使用泛型
//ArrayList<Integer> aList = new ArrayList();
        aList.add(95);
        aList.add(96);
        aList.add(97);
        aList.add(98);
        aList.add(99);
        //資料型態不安全
        aList.add("100");
        //輸出
//for(Integer score:aList)
        for (Object score : aList) {
            //會出現ClssCastException錯誤
            int stuScore = (Integer) score;
        }
```

Map使用泛型



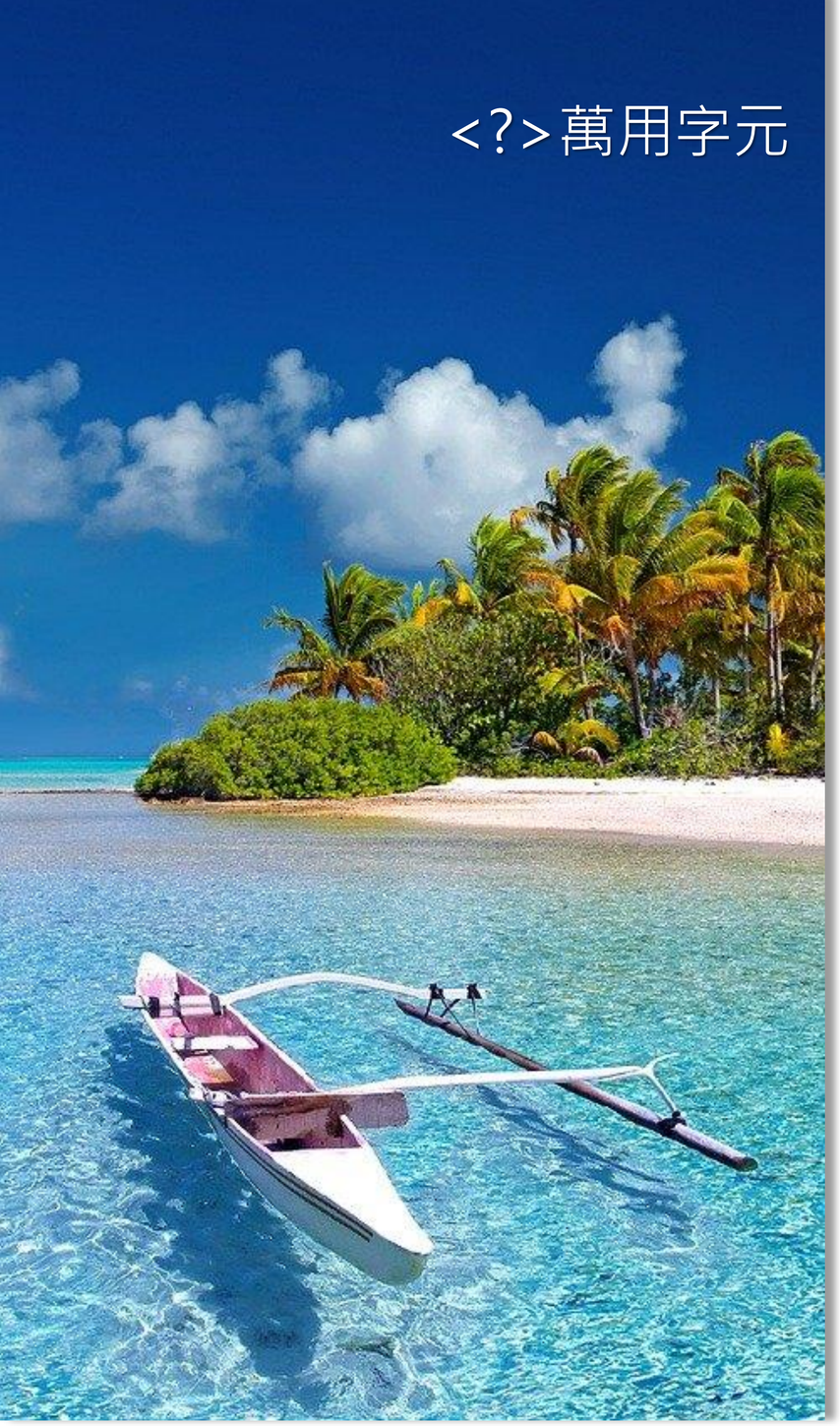
```
Map<String,Integer> map=new HashMap();
    map.put("Amy",99);
    map.put("Mike",95);
    map.put("Tom",100);
    //利用entrySet()方法巡訪map
    Set<Map.Entry<String,Integer>> entry=map.entrySet();
    Iterator<Map.Entry<String,Integer>> it=entry.iterator();
    while(it.hasNext())
    {
        Map.Entry<String,Integer> e=it.next();
        String key=e.getKey();
        Integer value=e.getValue();
        System.out.println(key+","+value);
    }
```

自訂泛型



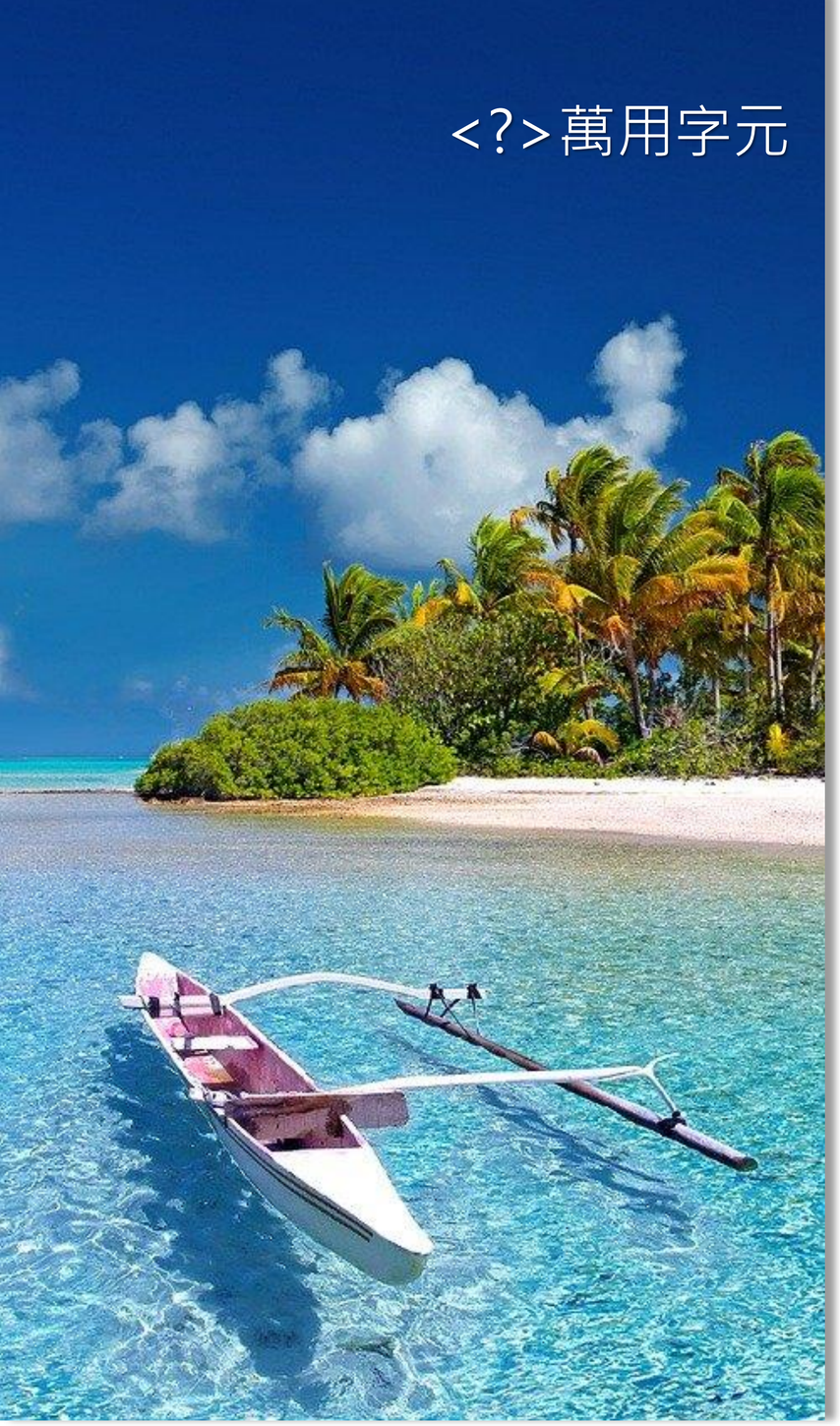
```
public static void main(String[] args) {

        Person p=new Person();
        p.setSexT(123);//此時為Object型別
        Person<String> p1=new Person();
        p1.setSexT("123");//此時必需為字串型別才可輸入
        SubPerson sp=new SubPerson();//子類別因繼承父類別,不用再宣告
        sp.setSexT(1);
//測試泛型方法
//String test[]= {"hello","ok","kitty"};
//Person<String> p1=new Person<String>();
//List<String> arrayToList = p1.arrayToList(test);
//Integer test1[]= {1,2,3,4,5,6};
//Person<String> p2=new Person<String>();
//List<Integer> arrayToList2 = p2.arrayToList(test1);
//System.out.println(arrayToList);
//System.out.println(arrayToList2);

}
}
class SubPerson extends Person<Integer>{ //因繼承父類即可使用泛型
}
class Person<T> {
   int age;
   String name;
   T sexT;
   public Person() {
   }
   public Person(String name,int age,T sexT){
      this.name=name;
      this.age=age;
      this.sexT=sexT;
   }
   public void setSexT(T sexT)
   {
      this.sexT=sexT;
   }
   public T getSexT(){
      return sexT;
   }
   @Override
   public String toString() {
      return "Person{" + "age=" + age + ", name=" + name + ", sexT=" + sexT + '}';
   }
//泛型方法,跟上面的類別泛型無關
public <E> List <E> arrayToList(E[] arr) {
ArrayList<E> ls=new ArrayList<>();
for(E x:arr)
ls.add(x);
return ls;
}
}
```

<?>萬用字元



- List<String> c=null;
- List<Object> d=null;
- //d=c;//並列關係但資料型態不同不可以相互賦值
- //解決方法如下
- List<?> list=null;
- list=c;
- list=d;
- //萬用字元無法添加元素,但可以取值
- List<String> ls=new ArrayList<>();
- ls.add("hello");
- ls.add("kitty");
- list=ls;
- //list.add("?");
- System.out.println(list.get(0));

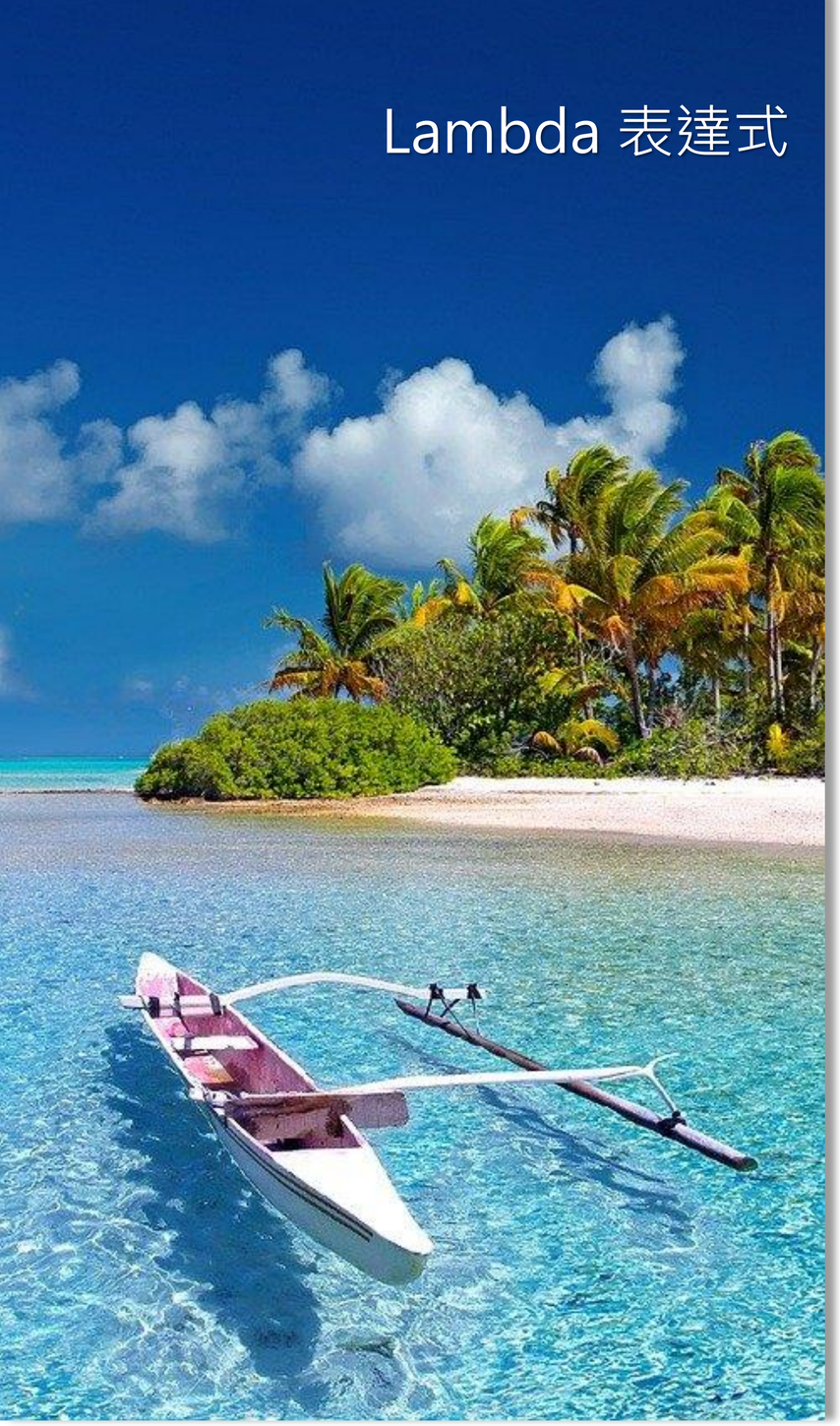
<?>萬用字元



- //有限制條件的萬用字元
- //<? extends Class> 小於等於
- //<? super Class>大於等於
- class Father{}
- class Son extends Father{}
- public static void main(String[] args) {
- List<? extends Father> list1=null;
- List<? super Father> list2=null;
- List<Father> list3=null;
- List<Son> list4=null;
- List<Object> list5=null;
- list1=list3;
- list1=list4;
- //list1=list5;
- list2=list3;
- //list2=list4;
- list2=list5;
- }

LAMBDA 表達式

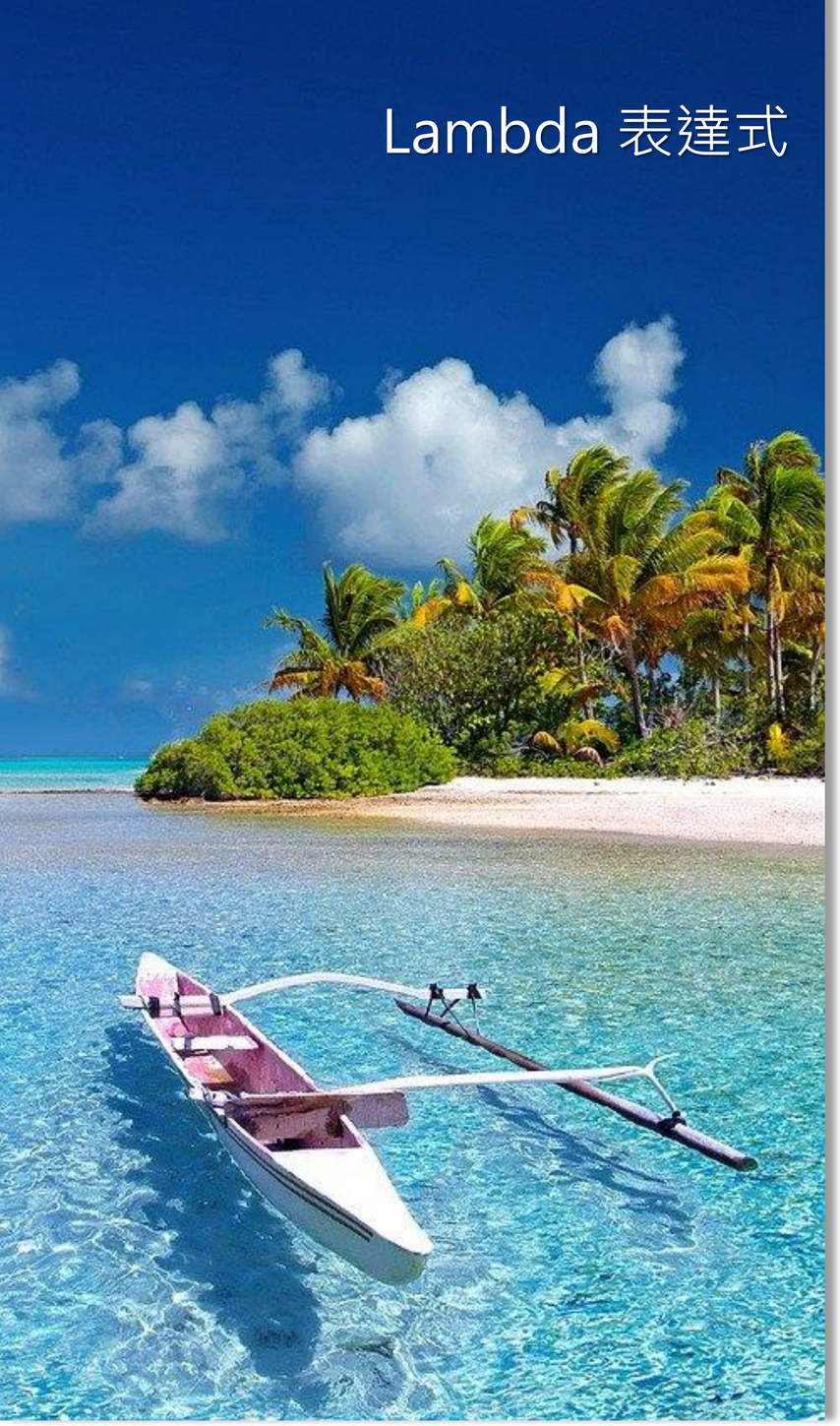
Lambda 表達式



一

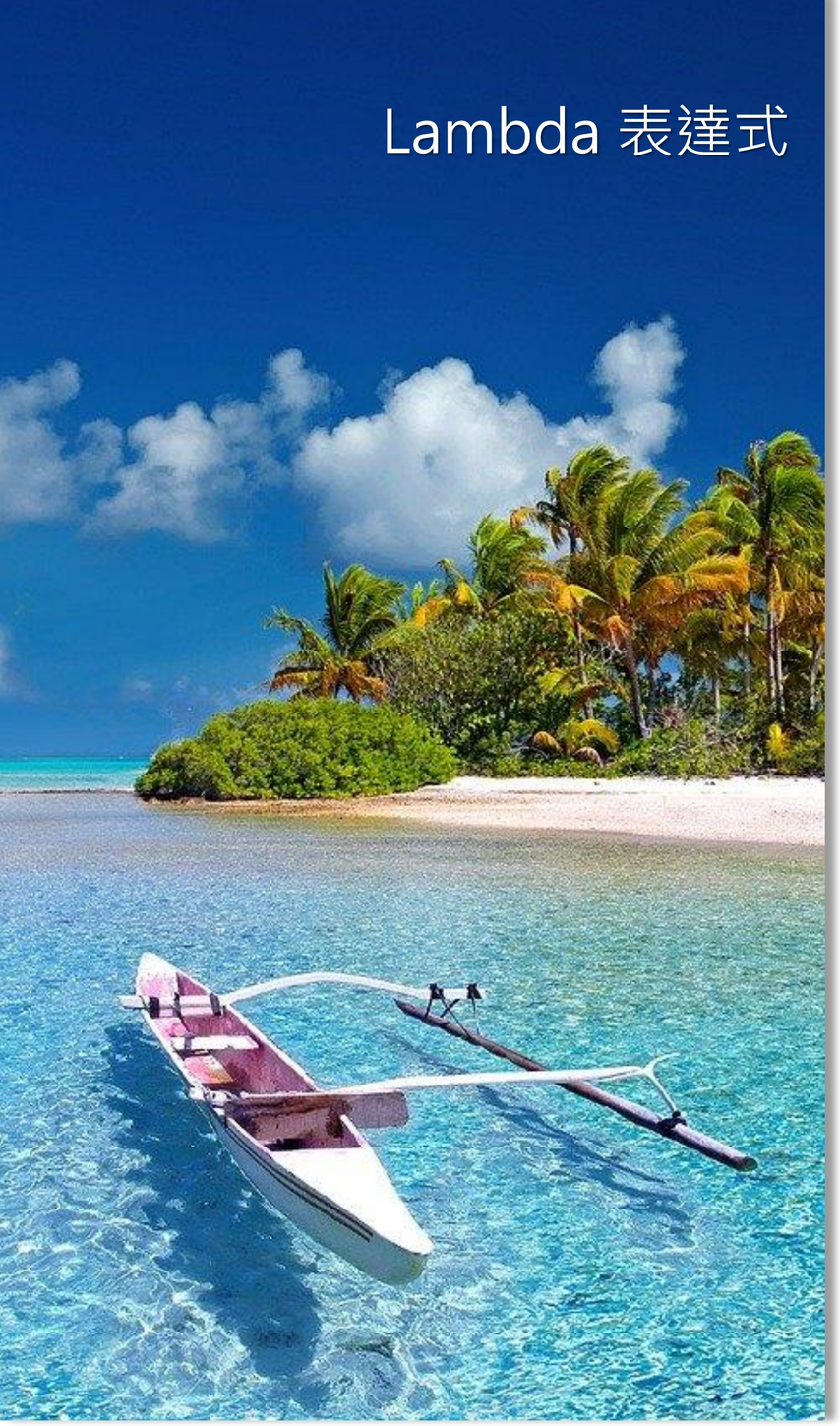
- Lambda是一個匿名函數,我們可以把Lambda表達式理解為一段可以傳遞資訊的代碼,將代碼像數據一樣的進行傳遞,使用它可以寫出更簡潔,更靈活的代碼,作為一種快速的代碼風格,使JAVA的語言表達能力可以得到提升
- 架構在函數式介面的語法
- Lambda表達式的本質:作為介面的實例

Lambda 表達式



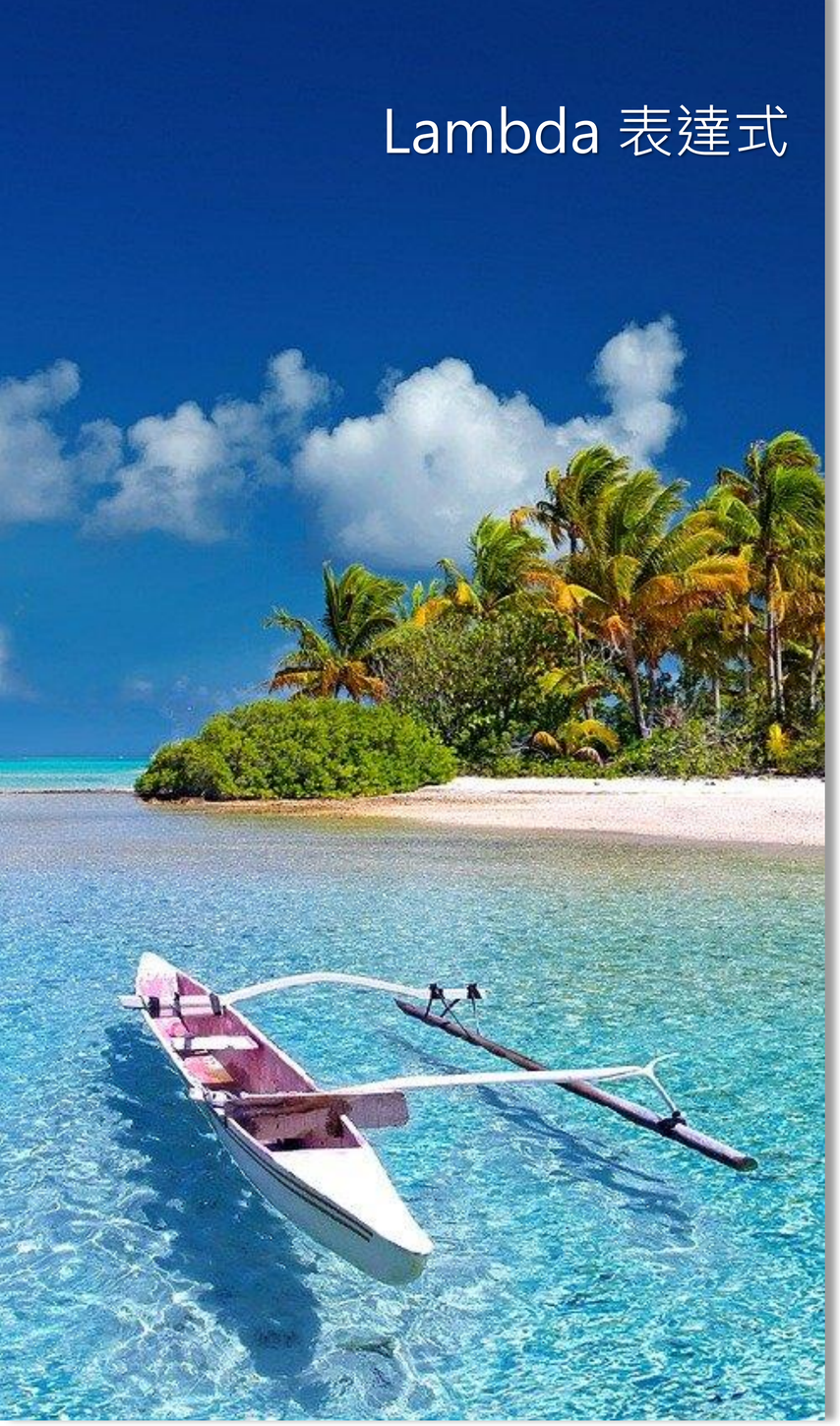
```
public static void main(String[] args) {
    //方式一,產生Dog類別物件,呼叫方法
    System.out.println("===產生Dog類別物件,呼叫方法=====");
    Dog doggy=new Dog();
    doggy.run();
    //方式二,使用匿名類別方式
    System.out.println("\n===使用匿名類別方式=====");
    Pet kitty=new Pet(){
      public void run(){
        System.out.println("我是匿名的貓,我會跑");
      }
    };
    kitty.run();
  //方式三,使用Lambda表達式
    System.out.println("\n===使用Lambda表達式=====");
    Pet micky=()->System.out.println("我是Lambda的老鼠,我會跑");
    micky.run();
  }
}
//宣告類別實作介面
class Dog implements Pet{
  @Override
  public void run(){
    System.out.println("我是傳統的狗,我會跑");
  }
}
//宣告介面
interface Pet{
  void run();
}
```

Lambda 表達式



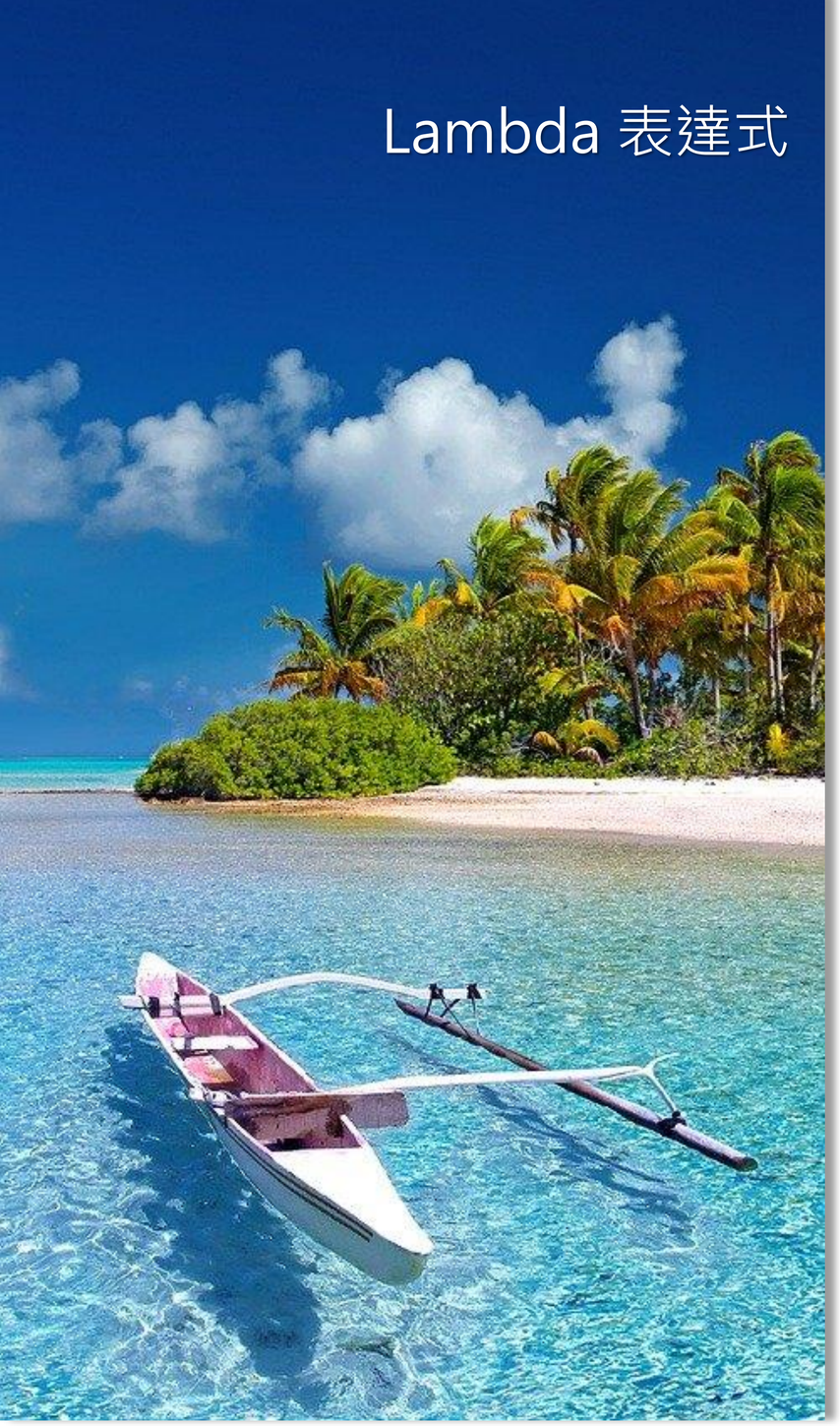
```
System.out.println("===使用匿名類別==============");
    Comparator<Integer>com=new Comparator<Integer>(){
        @Override
        public int compare(Integer o1, Integer o2) {
            return Integer.compare(o1, o2);
        }
    };
    System.out.println(com.compare(20,20));
    System.out.println("\n===使用Lambda表達式==========");
    Comparator<Integer> com1=(o1,o2) ->Integer.compare(o1, o2);
    System.out.println(com.compare(21,20));
    System.out.println( "\n===使用方法參照==============");
    Comparator<Integer> com2=Integer::compare;
    System.out.println(com2.compare(2,20));
```

Lambda 表達式



- 匿名函式表達式
- 運算符 -> 將表達式分為二部份
- 本質:為實作介面的物件
- 語法(input)->{body }
  - 左邊為方法輸入參數,即為介面中之抽象方法之參數
  - 右邊為方法實作本體

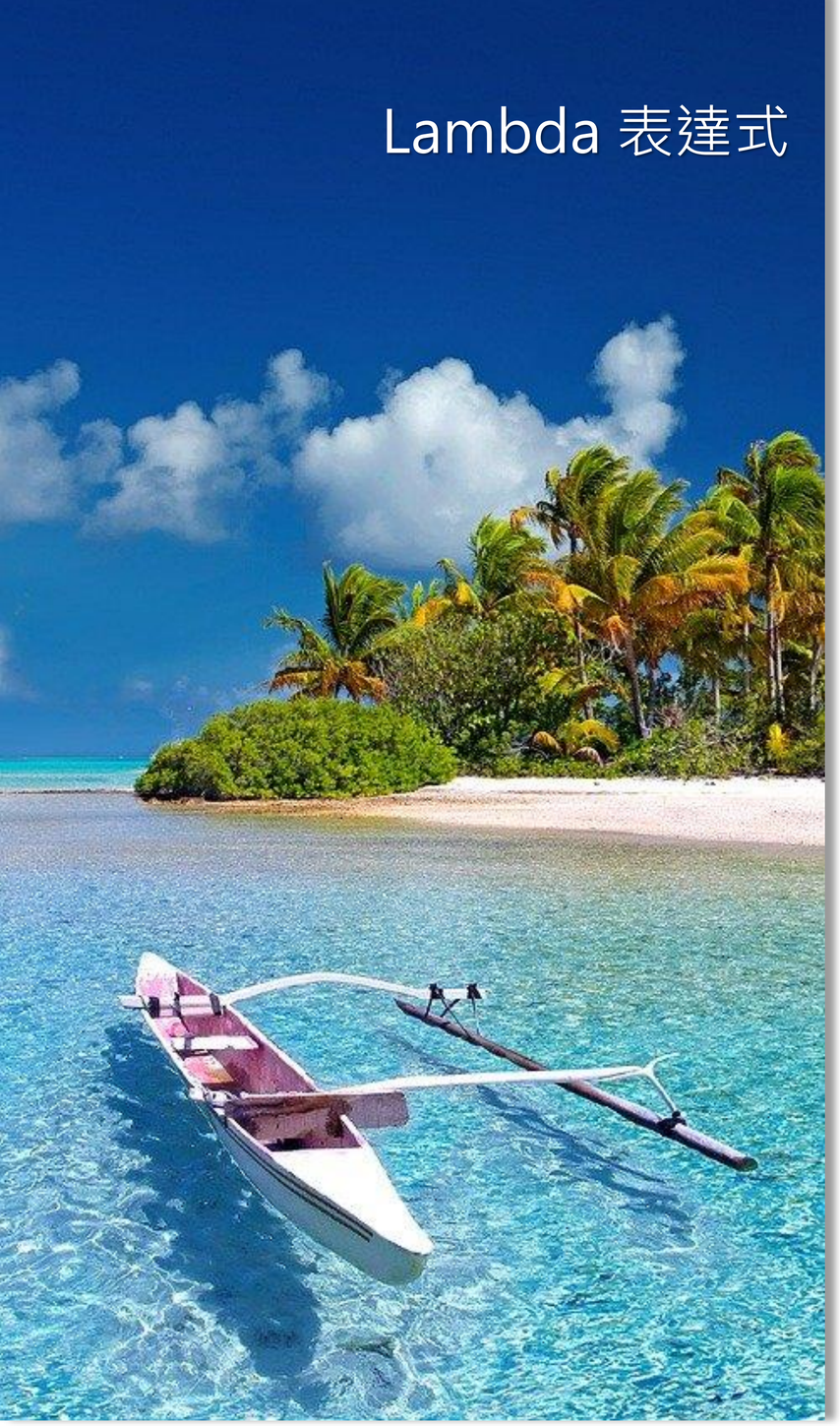
Lambda 表達式



一

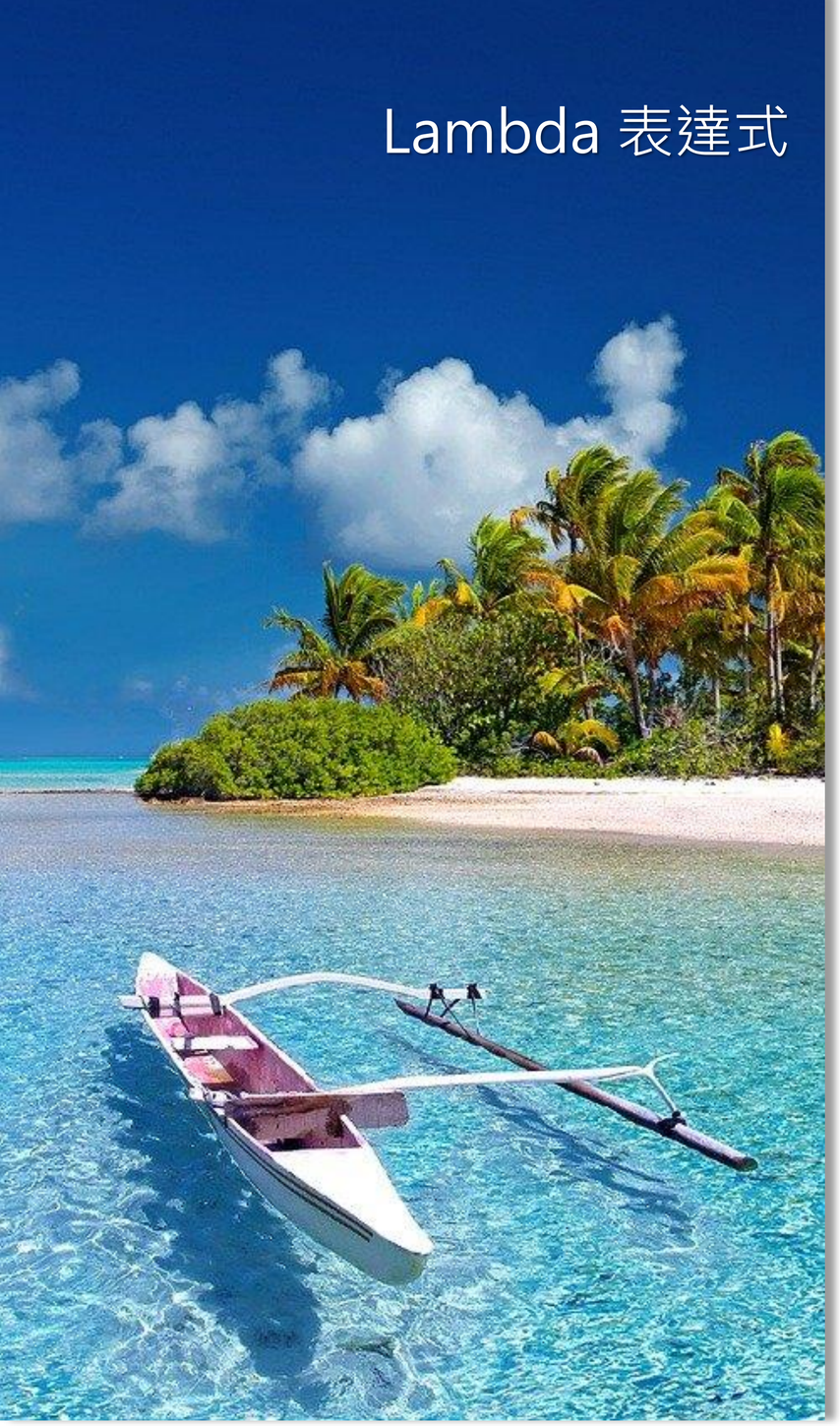
- Lambda表達式六種格式
- 輸入:方法簽章小括號的變化
  - 無參數 ( )
  - 有一個參數 (T var) , (var) , var
    - 同迴圈,若敘述只有一行大括號可省略,參數若只有一個 小括號也可省略
  - 有二個參數 (T var1 ,T var 2) , (var1 , var2)
    - 類型推定,所以同一個參數一樣,資料型態可省略不寫
- 輸出:方法本體大括號的變化
  - 敘述只有一行,無返回值 void 可將大括號省略
  - 敘述只有一行,有返回值 return ,大括號與 return 相依為命 , 二個要嘛同時出現 要嘛同時消失 不能只出現一個.
  - 敘述有多行,不管是 void 還是 return 皆無法省略大括號

Lambda 表達式



- 格式一:沒有參數 沒有返回值
- Pet micky=()->System.out.println("我是Lambda的老鼠,我會跑");
- micky.run();

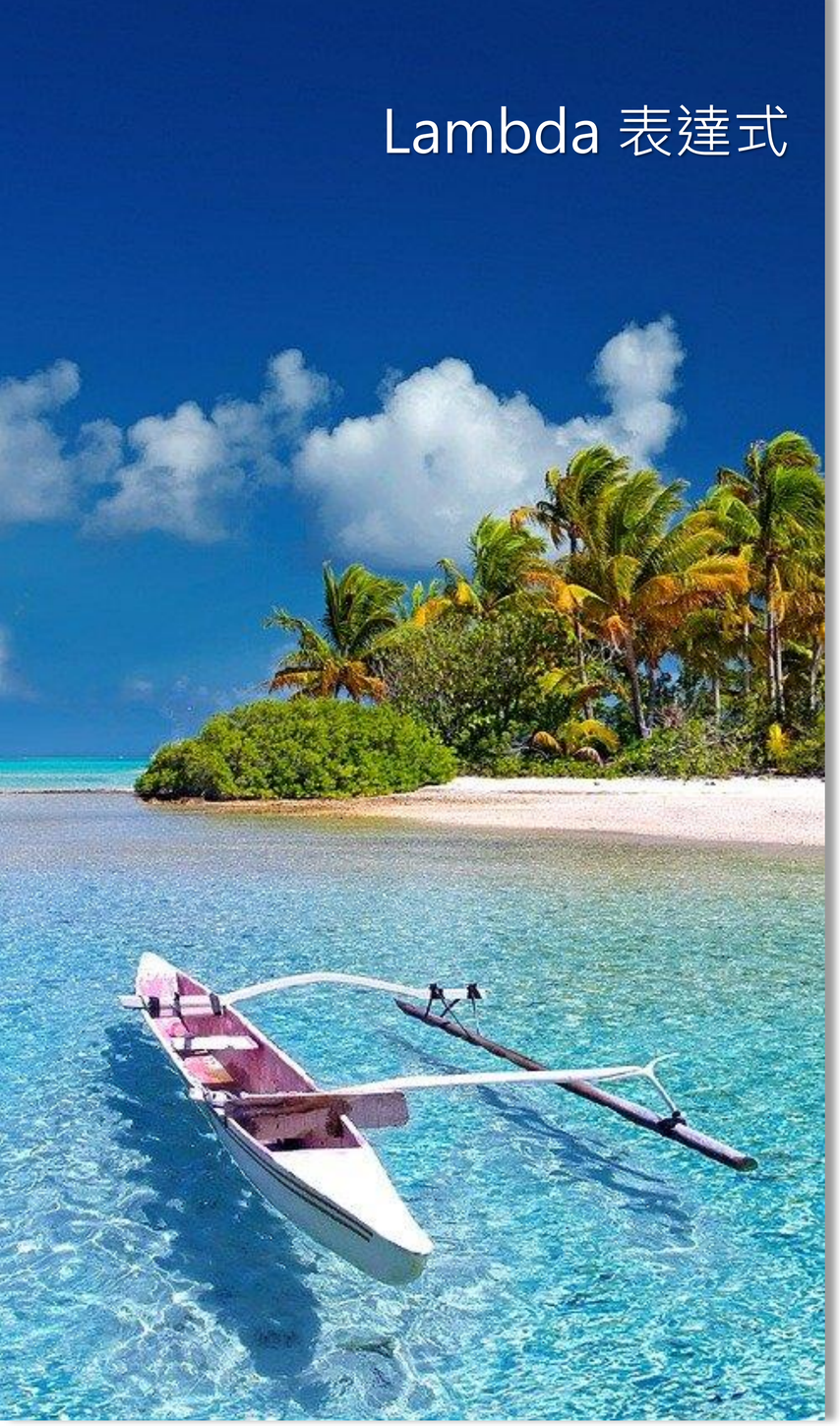
Lambda 表達式



- 格式二:有一個參數 但沒有返回值
- 格式三:有一個參數,類型推斷 但沒有返回值

```
System.out.println("====傳統==============");
    Consumer<String> con=new Consumer<String>(){
      @Override
      public void accept(String t) {
        System.out.println(t);
      }
    };
    con.accept("我是傳統消費類");
    System.out.println("\n====Lambda==============");
    Consumer<String> con1=( String t)->System.out.println(t);
Consumer<String> con2=(t)->System.out.println(t);//類型推斷
//方法參照 Consumer<String> con2=System.out::println
    con1.accept(“大家好Lambda表達式”);
```

Lambda 表達式

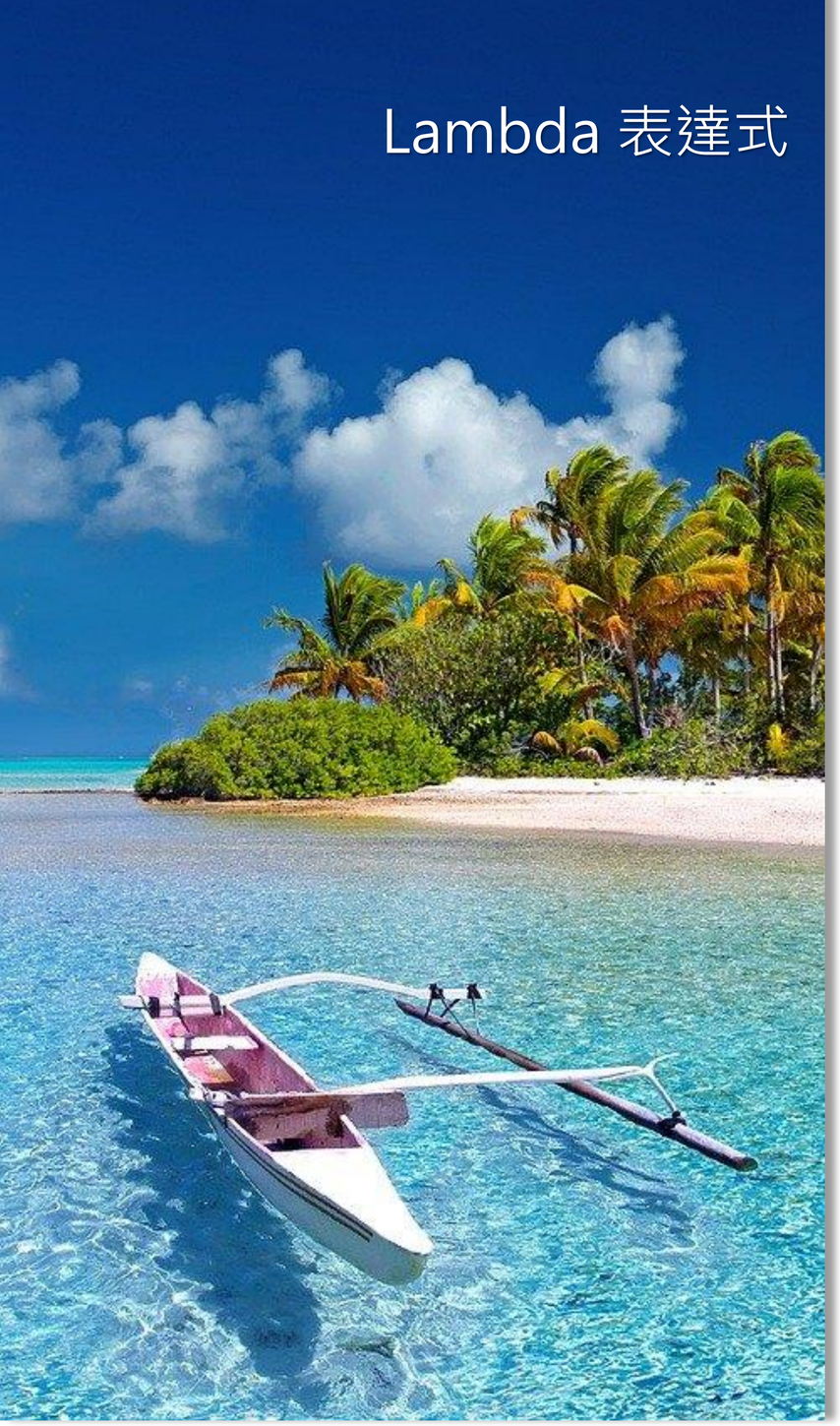


- 格式四:若只有一個參數,可省略括號 但沒有返回值

```
System.out.println("====傳統==============");
    Consumer<String> con=new Consumer<String>(){
        @Override
        public void accept(String t) {
            System.out.println(t);
        }
    };
    con.accept("我是傳統消費類");
    System.out.println("\n====Lambda==============");
    Consumer<String> con1=t->System.out.println(t);
```

- con1.accept(“大家好Lambda表達式”);

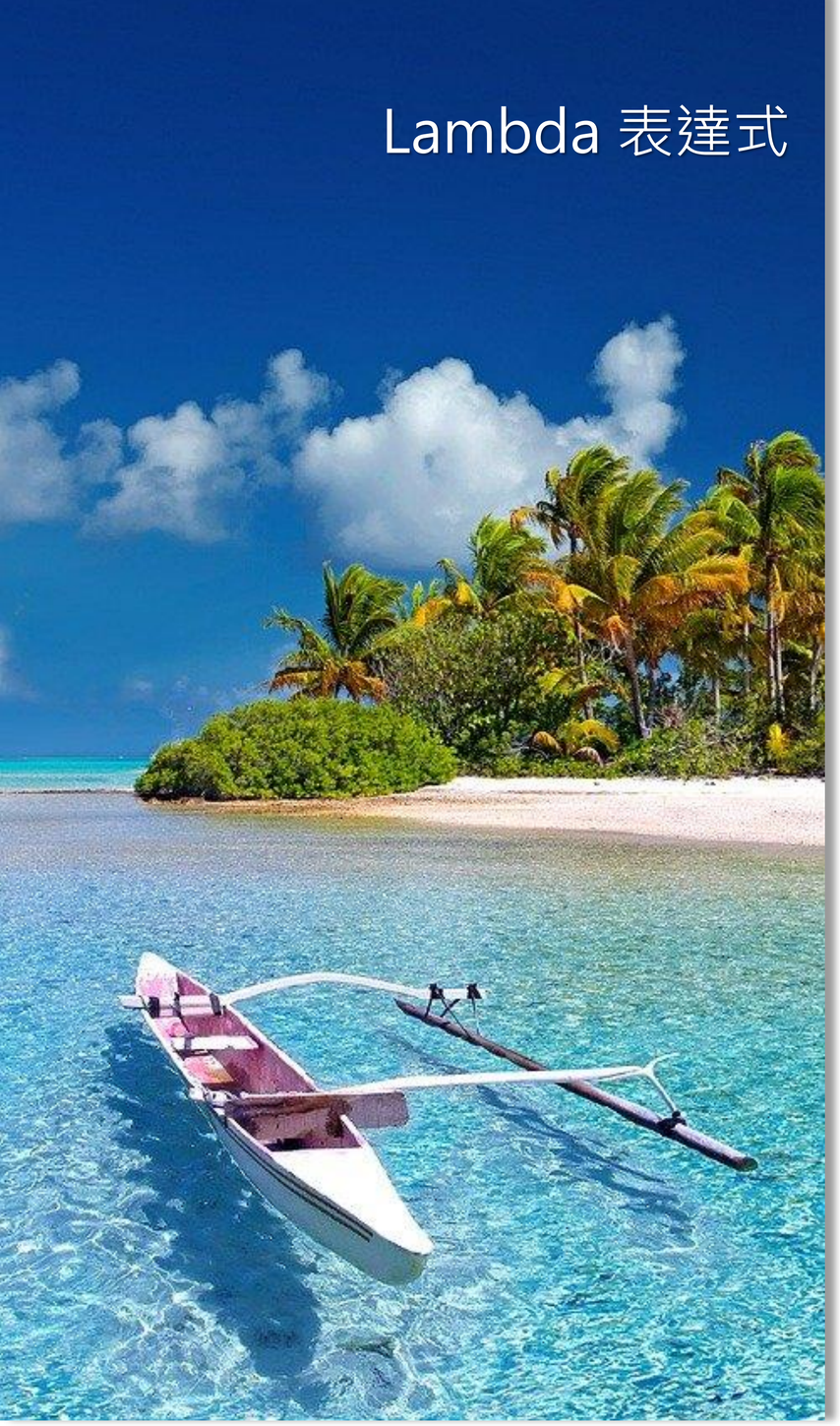
Lambda 表達式



- 格式五:方法本體若只有一個敘述,return與大括號皆可省略
- 若不想省略return則大括號也不能省略,且一行敘述後要分號,大括號後也要加分號

```
System.out.println("===使用匿名類別==============");
    Comparator<Integer>com=new Comparator<Integer>(){
        @Override
        public int compare(Integer o1, Integer o2) {
return Integer.compare(o1, o2);
        }
    };
    System.out.println(com.compare(20,20));

    System.out.println("\n===使用Lambda表達式==============");
    Comparator<Integer> com1=(o1,o2) ->Integer.compare(o1, o2);
System.out.println(com1.compare(21,20));
```

Lambda 表達式

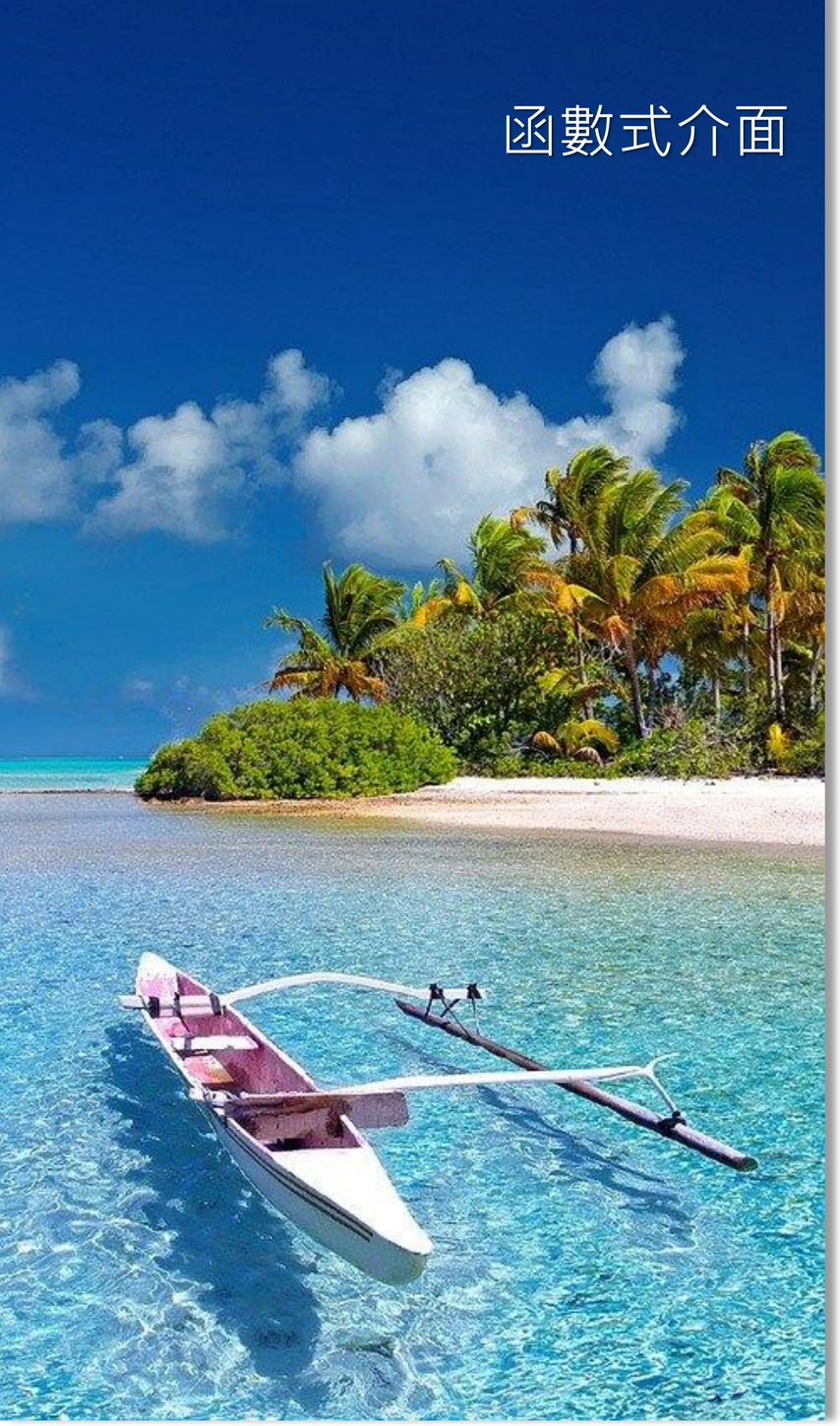


- 格式六:多個參數,多行敘述大括號不能省略

```
System.out.println("===使用匿名類別==============");
    Comparator<Integer>com=new Comparator<Integer>(){
        @Override
        public int compare(Integer o1, Integer o2) {
            System.out.println(o1);
            System.out.println(o2);
            return Integer.compare(o1, o2);
        }
    };
    System.out.println(com.compare(20,20));

    System.out.println("\n===使用Lambda表達式==============");
    Comparator<Integer> com1=(o1,o2) ->{
            System.out.println(o1);
            System.out.println(o2);
            return Integer.compare(o1, o2);
        };
    System.out.println(com.compare(21,20));
```

函數式介面 FUNCTIONAL INTERFACE

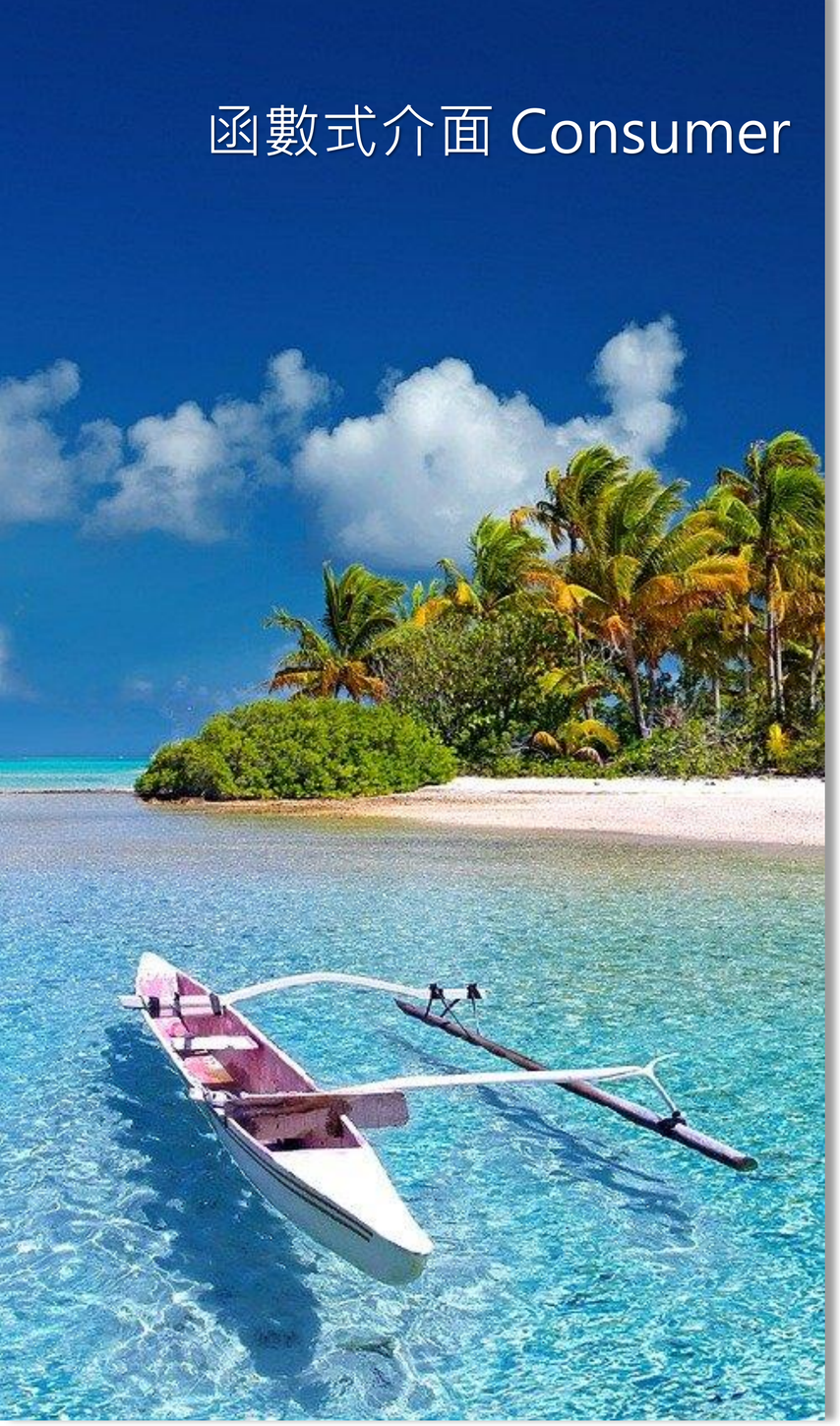
函數式介面



- 介面中只有一個抽象方法
- Java內置四大核心函數式介面

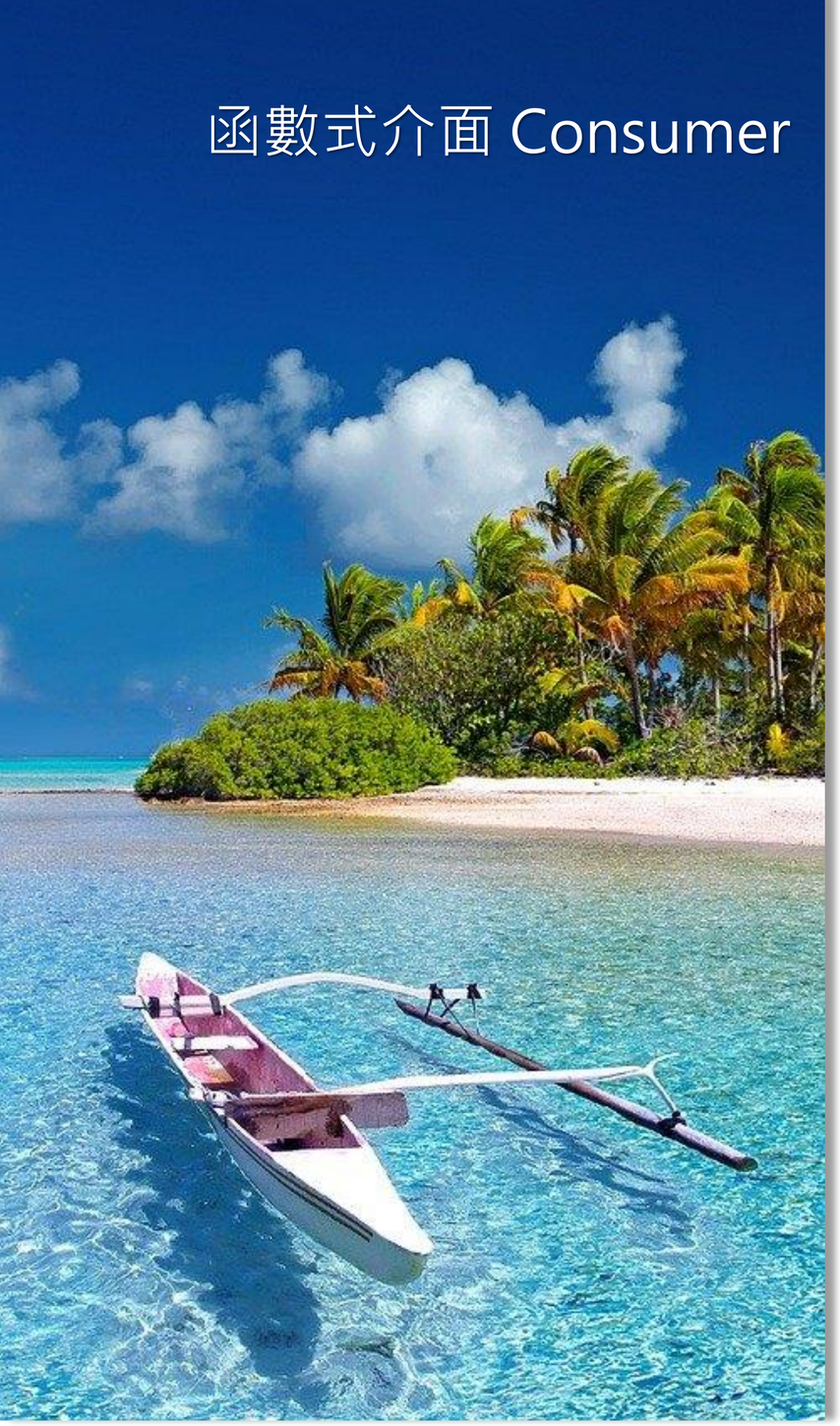
函數式介面	參數類型	返回類型	用途
Consumer<T>	T	Void	消費型,沒有回傳 void accept(T t)
Supplier<T>	無	T	供給型,提供T型別的實例/物件T get()
Functional<T,R>	T	R	功能型,由T型別轉成U型別 R apply(T t)
Predicate<T>	T	boolean	評斷型,回傳Boolean Boolean test(T t)

函數式介面 Consumer



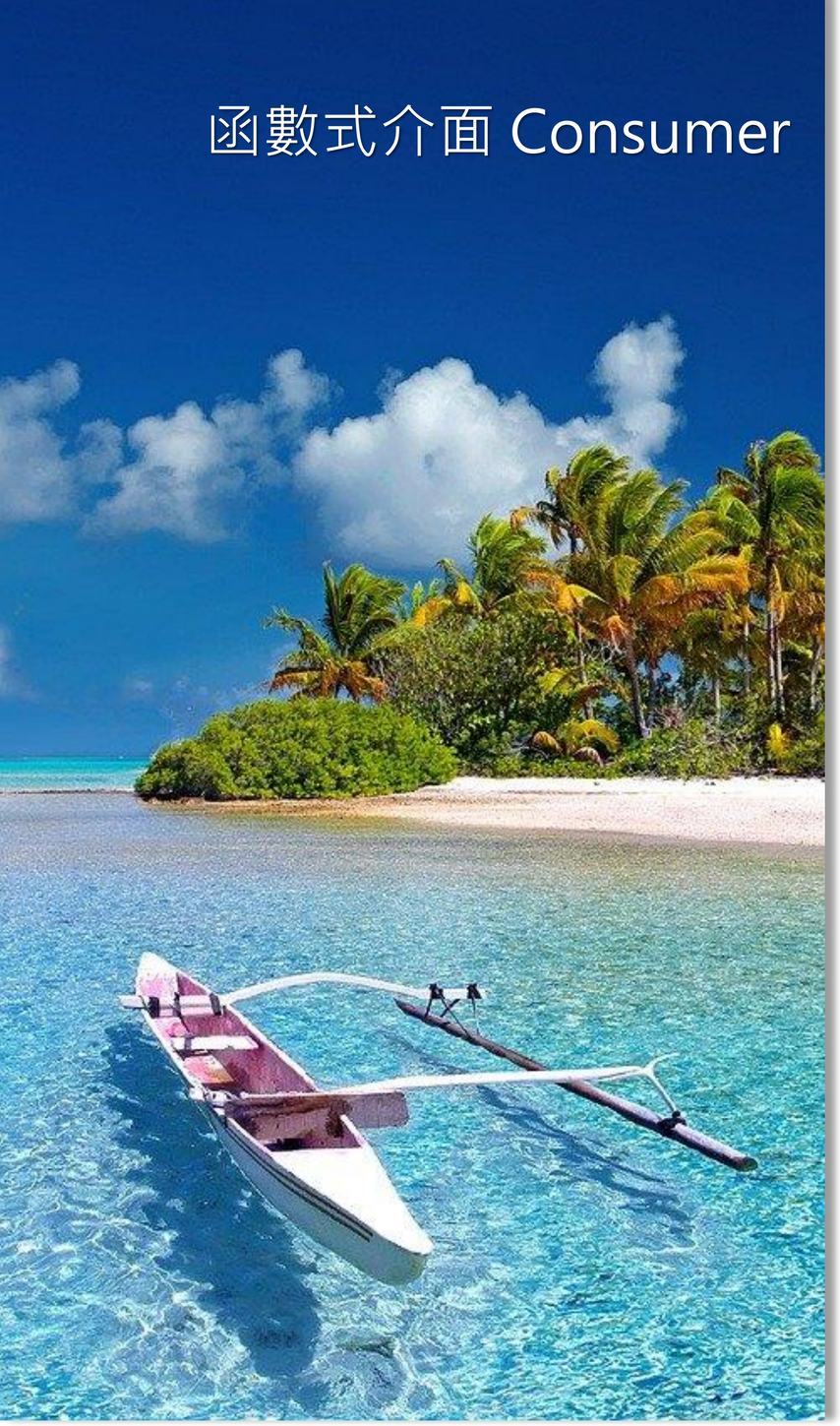
```
public static void main(String[] args) {
        shopping(500, new Consumer<Integer>() {
            @Override
            public void accept(Integer t) {
                System.out.println("消費金額為" + t);
            }
        });
    System.out.println("============改成Lambda表達式================");
        shopping(800, t -> System.out.println("消費金額為" + t));
    }
public static void shopping(int money,Consumer<Integer> con){
    con.accept(money);
}
```

函數式介面 Consumer



```
List<String> s = Arrays.asList("mike ", "amy ", "jimmy ", "linda ");
Consumer<String> c = System.out::print;
Consumer<String> again = c.andThen(x -> System.out.print(x.toUpperCase()));
Consumer<String> oneMore = again.andThen(x -> System.out.print(x.substring(0,2)+" "));
s.forEach(oneMore);
System.out.println();
```

函數式介面 Consumer

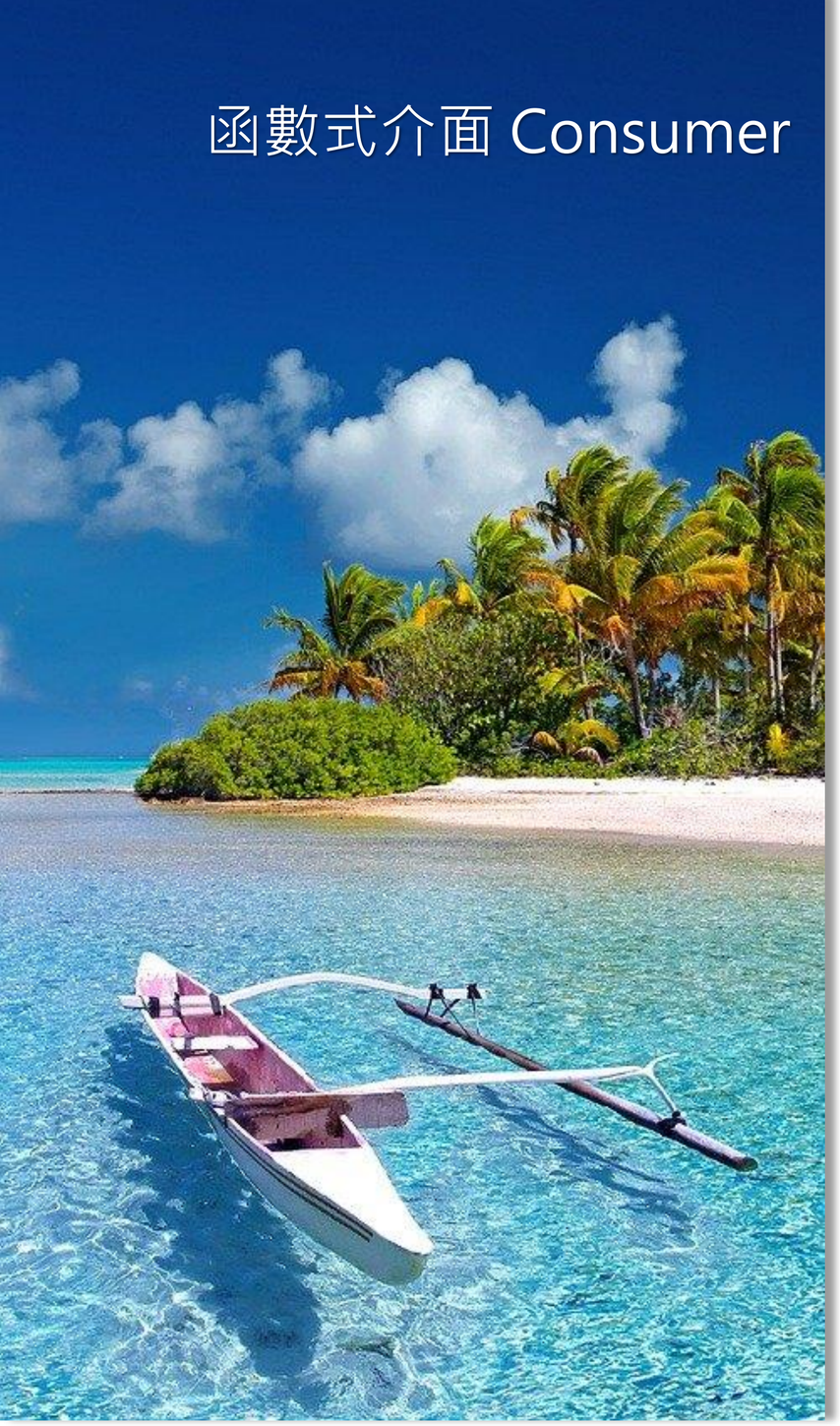


Which statement on line 1 enables this code to compile? *

```
//line 1
var fruits=List.of("apple","orange","banana","lemon");
fruits.forEach(function);
```

- A. Function<String,String> function=x->x.substring(0,2);
- B. Supplier<String> function=( )->fruits.get(0);;
- C. Consumer<String> function=(String f)->{System.out.println(f);};
- D. Predicate<String> function=a->a.equals("banana");

函數式介面 Consumer

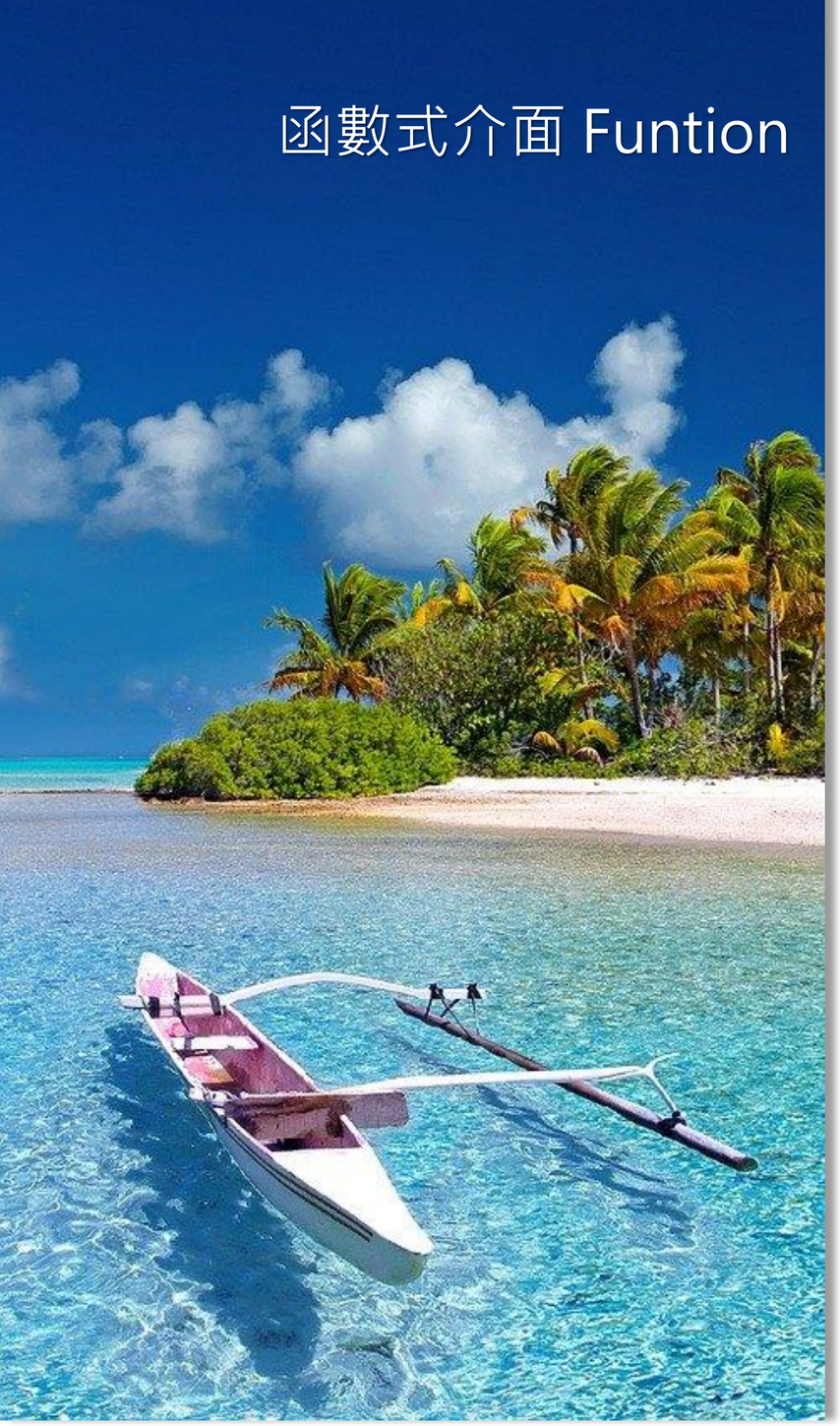


What is the output?

```
public static void main(String[] args) {
    List fruits=Arrays.asList("apple","orange","banana");
    Consumer<String> c=System.out::print;
    Consumer<String> output=c.andThen(x->System.out.println(":"+x.toUpperCase()));
    fruits.forEach(output);
    }
```

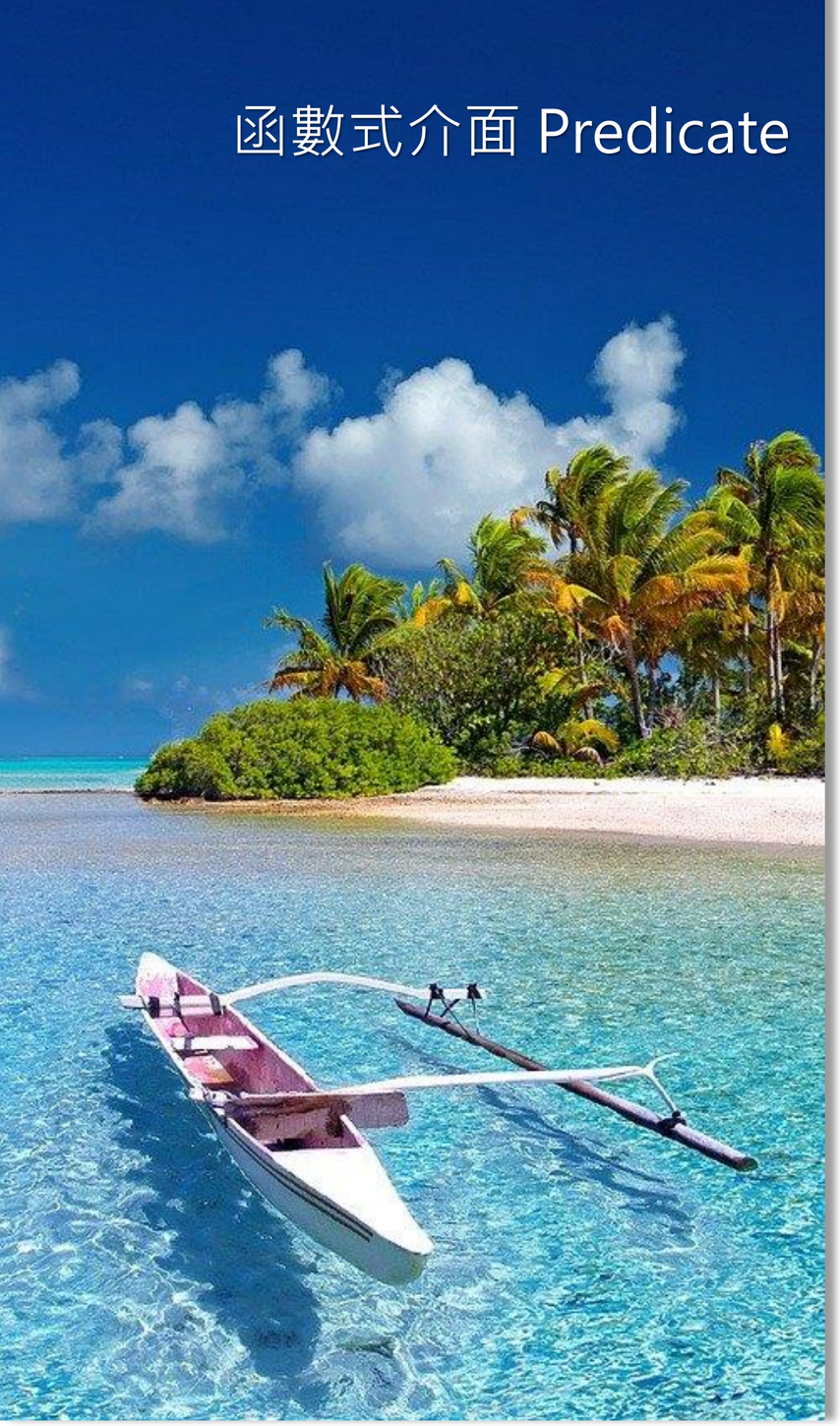
- A. :APPLE:ORANGE:BANANAappleorangebanana
- B. :APPLE:ORANGE:BANANA
- C. APPLE:apple ORANGE:orange BANANA:banana
- D. appleorangebanana:APPLE:ORANGE:BANANA
- E. apple:APPLE orange:ORANGE banana:BANANA

函數式介面 Funtion



```
System.out.println("===使用匿名類別==========");
        Function<Double, Long> fun = new Function<Double, Long>() {
            @Override
            public Long apply(Double t) {
                return Math.round(t);
            }
        };
        System.out.println(fun.apply(5.3));
        System.out.println("===使用Lambda表達式==========");
        Function<Double, Long> fun2 = x -> Math.round(x);
        System.out.println(fun2.apply(5.6));
        System.out.println("===使用方法參照==========");
        Function<Double, Long> fun3 = Math::round;
        System.out.println(fun3.apply(5.2));
```

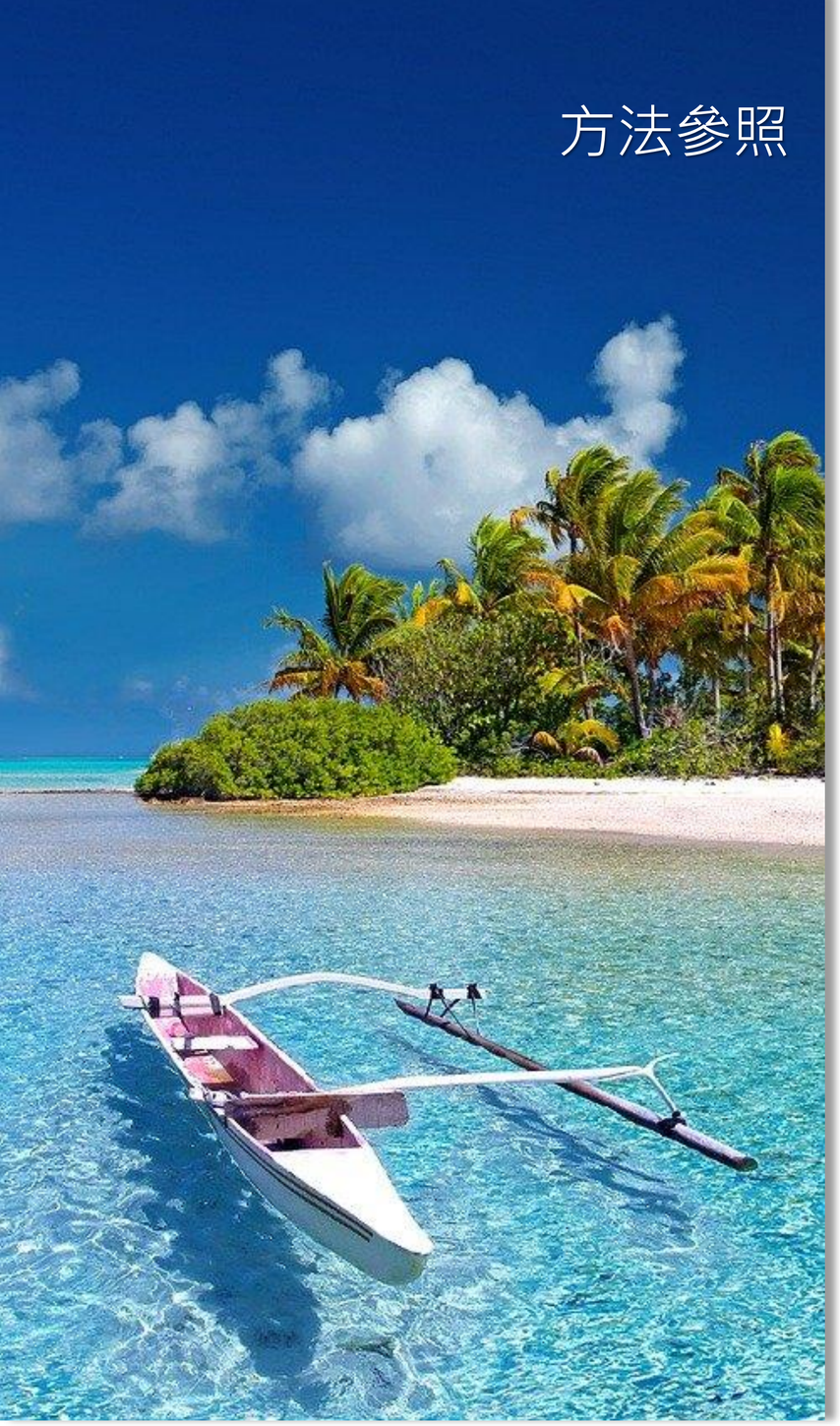
函數式介面 Predicate



```
public static void main(String[] args) {
        System.out.println("=====傳統寫法==============");
        List<String> list = Arrays.asList("張惠妹", "周杰倫", "張三豐", "周華健", "周子瑜");
        List<String> filterStr = filterString(list, new Predicate<String>() {
            @Override
            public boolean test(String t) {
                return t.contains("周");
            } });
        System.out.println(filterStr);
        System.out.println("=====Lambda寫法==============");
        List<String> filterStr1 = filterString(list, t -> t.contains("張"));
        System.out.println(filterStr1);
    }
    //依據給定的規則,篩選集合中的字串,此規則由Predicate的方法決定,此方法先定義
    public static List<String> filterString(List<String> list, Predicate<String> pre) {
        ArrayList<String> filterList = new ArrayList();
        for (String s : list) {
            if (pre.test(s)) {
                filterList.add(s);
            }}
        return filterList;
    }}
```

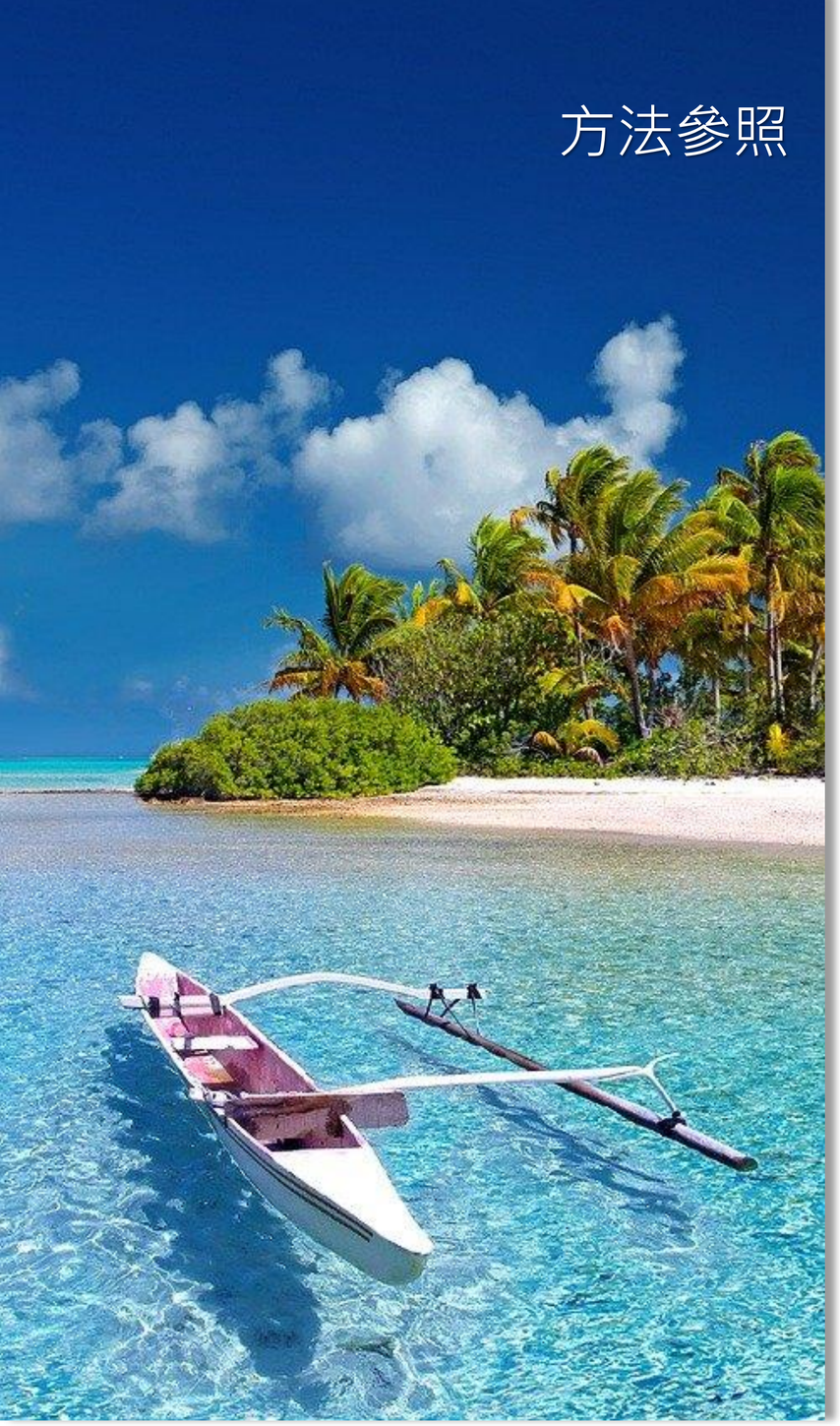
方法參照 METHOD REFERENCE

方法參照



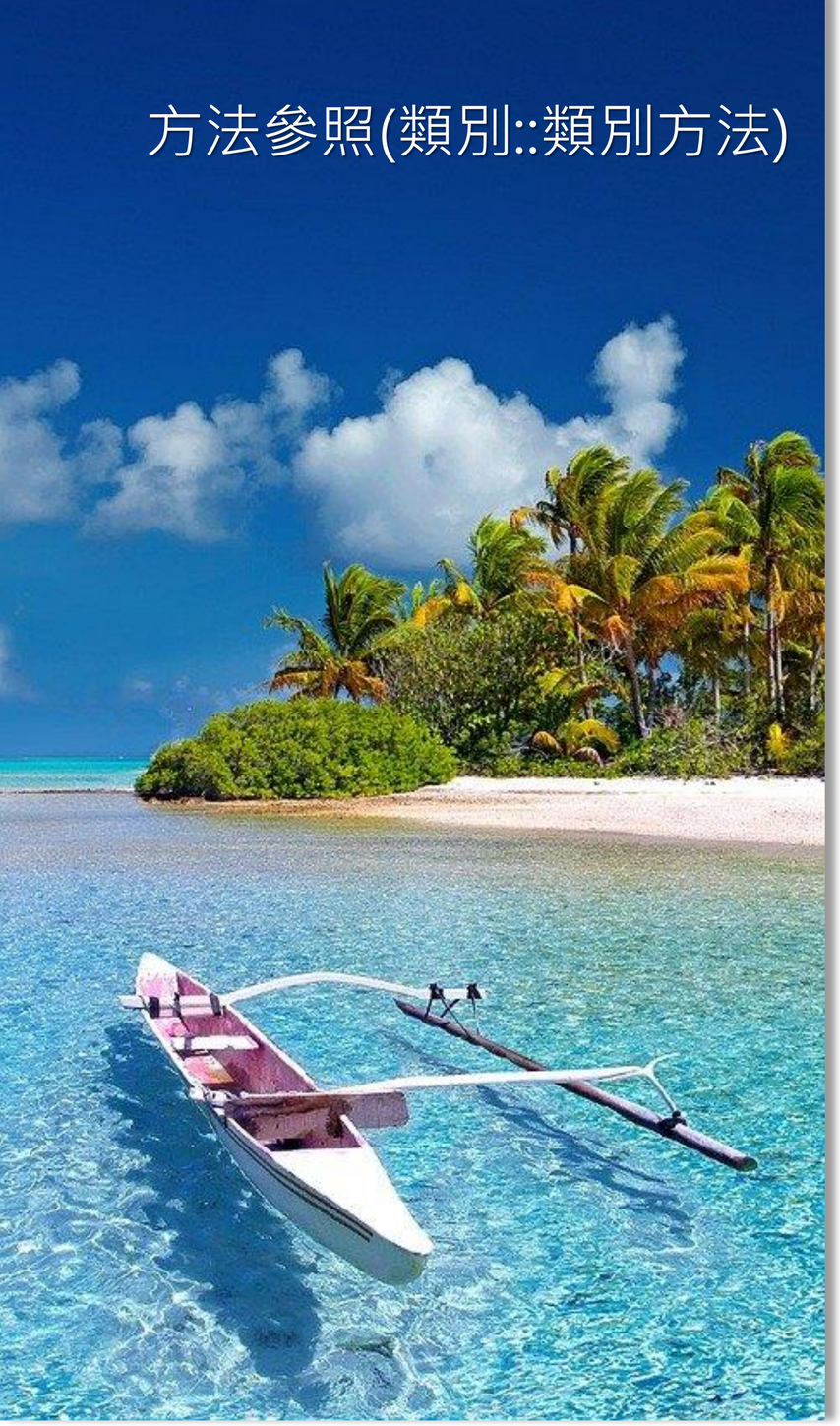
- Lambda表達式分為三個部分
  - 方法參數 箭頭等號 方法內容
  - (x) -> System.out.println(“lambda”);
- 若Lambda本體已有對應之方法，可將lambda表示式再改寫為方法參照(method reference)，讓語法更簡潔
- 方法參照的使用要求,要求介面中的抽象方法的參數及返回值類型需與參照的方法參數及返回值類型皆相同
- 語法 (調用者)類別或物件::方法名稱
- 依據被呼叫的方法種類和來源,可分成下類四種
  - 方法是靜態(static)方法，來自某類別
  - 方法是物件(instance)方法，來自lambda表達式之外的參考變數
  - 方法是物件(instance)方法，來自lambda表達式之內的參考變數
  - 使用new 呼叫建構子

方法參照



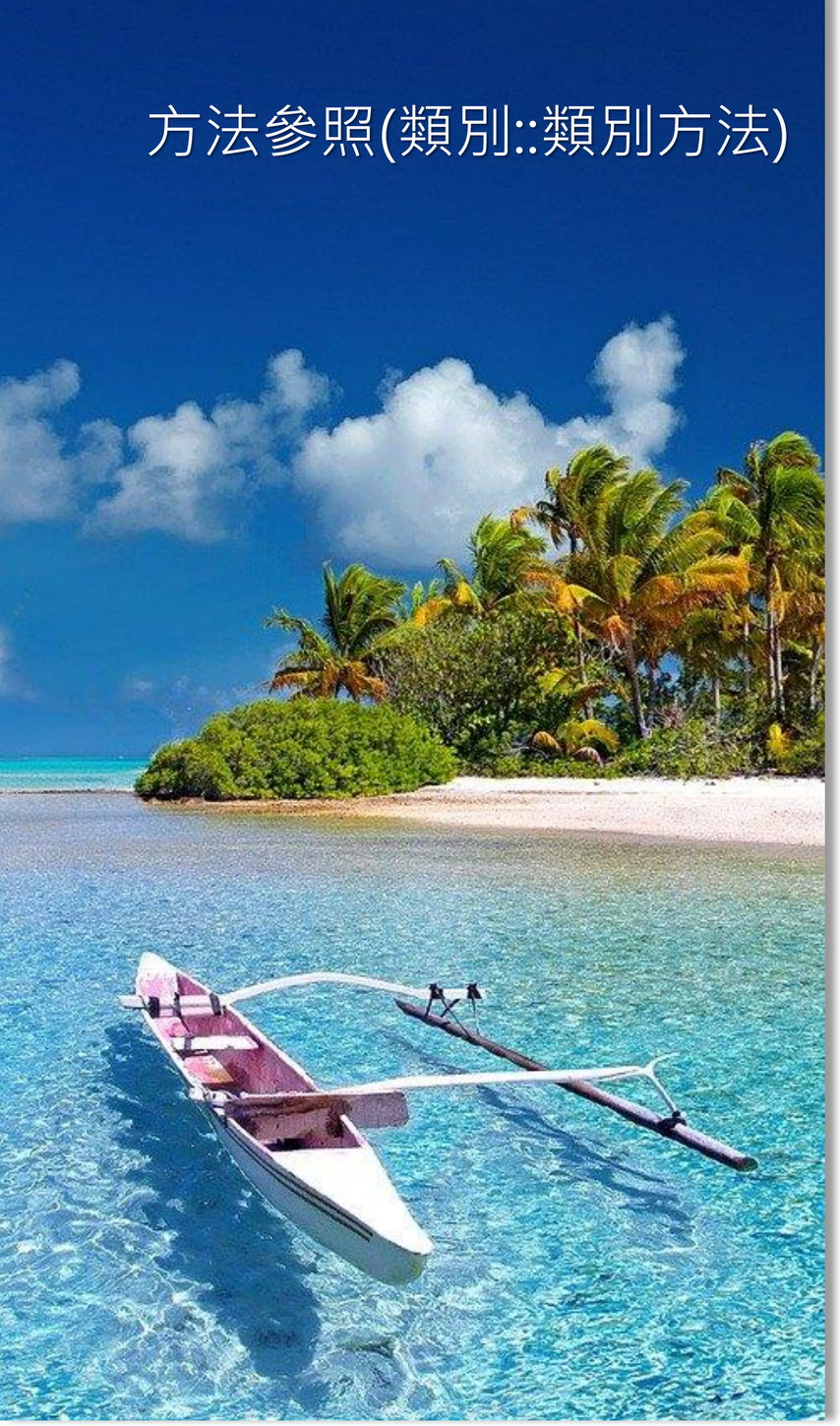
```
public static void main(String[] args) {

    Consumer<String> r = new Consumer<String>() {
        @Override
        public void accept(String t) {
            System.out.println(t);
        }
    };
    r.accept("輕鬆考取java證照");
    System.out.println("\n==使用Lambda表達式======\n");
    Consumer<String> r1 = t -> System.out.println(t);
    r1.accept("Lambda=>輕鬆考取java證照");

    System.out.println("\n==使用方法參照======\n");
    Consumer<String> r2 = System.out::println;
    r1.accept("方法參照=>輕鬆考取java證照");
}
```

方法參照(類別::類別方法)



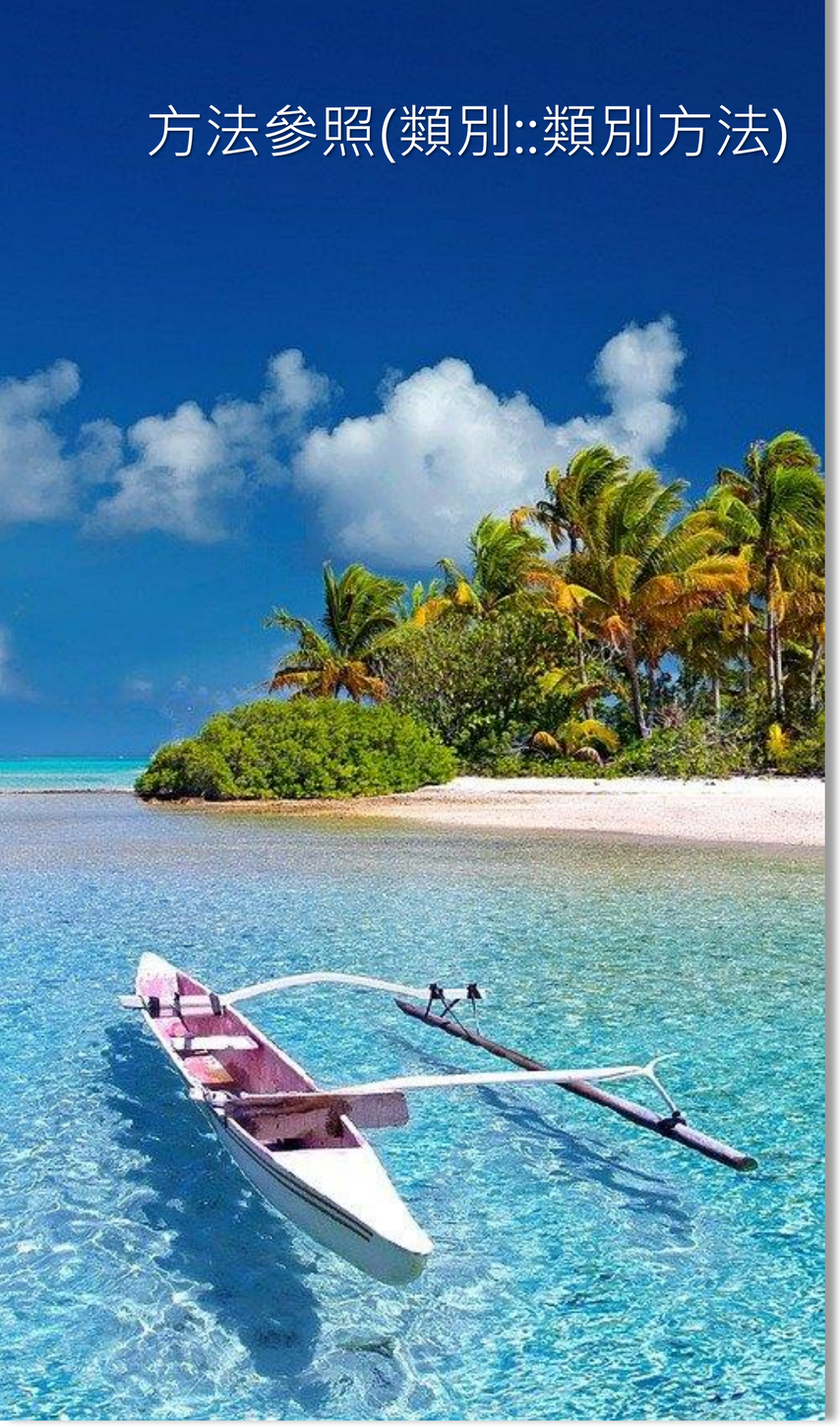
```
//Comparator 中的抽象方法 int compare(t1,t2)
//參照到 Integer中的類別方法 int compare(t1,t2)
System.out.println("===使用匿名類別==========");
    Comparator<Integer> com1=new Comparator<Integer>(){
        @Override
        public int compare(Integer o1, Integer o2) {
            return Integer.compare(o1,o2);
        }
    };
    System.out.println(com1.compare(50, 10));
    System.out.println("===使用Lambda表達式==========");
    Comparator<Integer> com2=(Integer o1, Integer o2)-> { return Integer.compare(o1,o2);};
    System.out.println(com2.compare(5, 10));
    System.out.println("===使用方法參照==========");
    Comparator<Integer> com3=Integer::compare;
    System.out.println(com3.compare(5, 5));
```

方法參照(類別::類別方法)



```
System.out.println("===使用匿名類別==========");
        Function<Double, Long> fun = new Function<Double, Long>() {
            @Override
            public Long apply(Double t) {
                return Math.round(t);
            }
        };
        System.out.println(fun.apply(5.3));
        System.out.println("===使用Lambda表達式==========");
        Function<Double, Long> fun2 = x -> Math.round(x);
        System.out.println(fun2.apply(5.6));
        System.out.println("===使用方法參照==========");
        Function<Double, Long> fun3 = Math::round;
        System.out.println(fun3.apply(5.2));
```

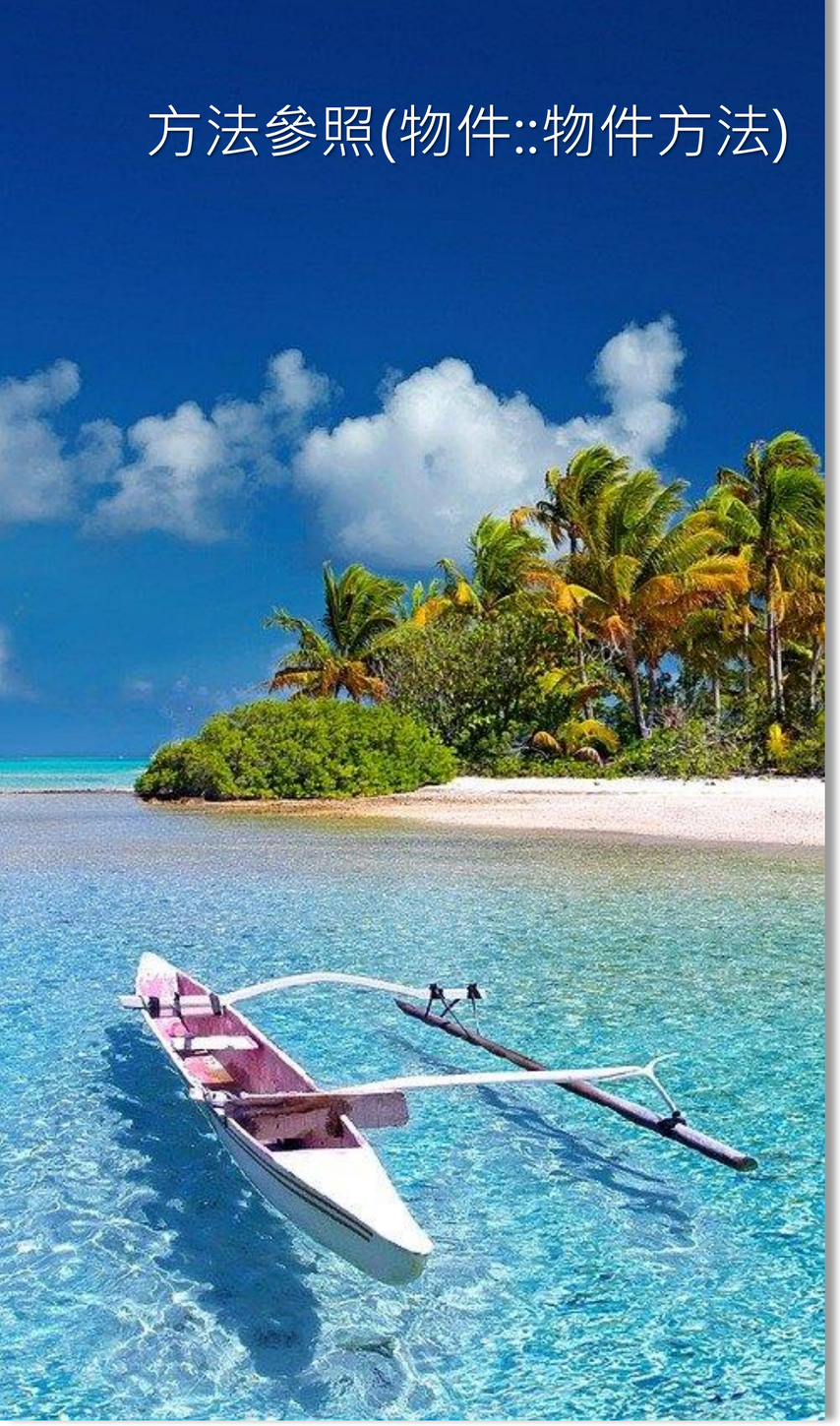
方法參照(類別::類別方法)



```
//以下程式貼至主方法內
List<String> names = Arrays.asList("Dog", "cat", "tiger", "animal", "Zoo");
        Collections.sort(names, new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                //呼叫自訂的排序方法
                return SortMethod.byLength(o1, o2);
            }
        });
        System.out.println(names.toString());
        System.out.println("=========以Lambda表達式改寫==================");
        Collections.sort(names, (o1, o2) -> SortMethod.byAlphaOrder(o1, o2));
        System.out.println(names.toString());
        System.out.println("=========以靜態方法參照改寫==================");
        Collections.sort(names, SortMethod::byAlphaOrderIgnoreCase);
        System.out.println(names.toString());
```

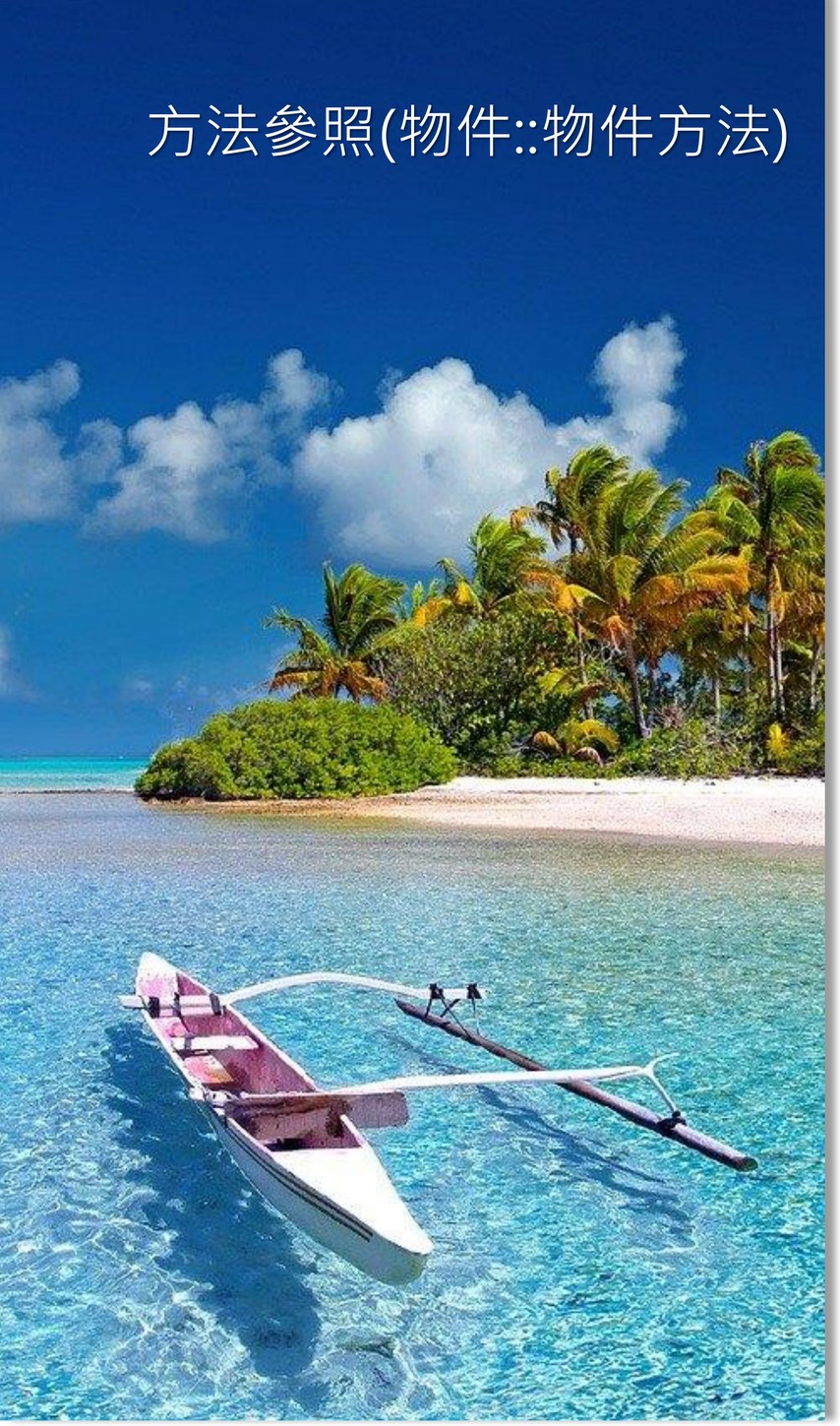
```
//宣告類別
class SortMethod {
//依字串長度排序

    static int byLength(String s1, String s2) {
        return s1.length() - s2.length();
    }
//依字母順序排序

    static int byAlphaOrder(String s1, String s2) {
        return s1.compareTo(s2);
    }
//依字母順序排序不分大小寫

    static int byAlphaOrderIgnoreCase(String s1, String s2) {
        return s1.compareToIgnoreCase(s2);
    }
}
```

方法參照(物件::物件方法)



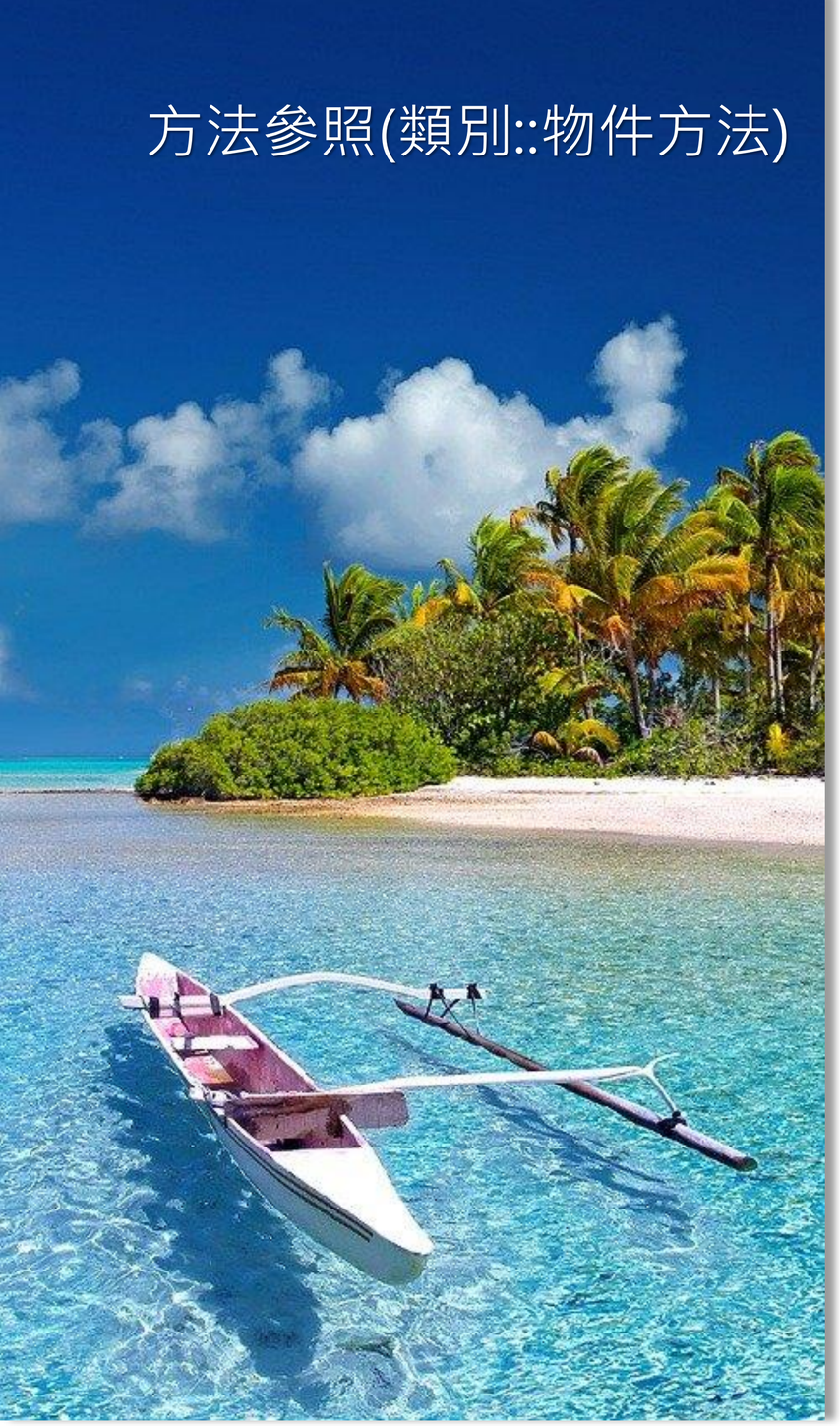
```
public static void main(String[] args) {
  Person p = new Person("Mike");
        Supplier<String> sup = new Supplier<String>() {
            @Override
            public String get() {
                return p.getName();
            }
        };
        System.out.println(sup.get());

        System.out.println("\n==使用Lambda表達式======\n");
        Supplier<String> sup1 = () -> {
            return p.getName();
        };
        System.out.println(sup1.get());
        System.out.println("\n==使用方法參照======\n");
        Supplier<String> sup2 = p::getName;
        System.out.println(sup1.get());
    }
}

class Person {
    String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Person(String name) {
        this.name = name;
    }
}
```

方法參照(物件::物件方法)



- //以下程式貼至主方法內
- List<Student> student = Arrays.asList(new Student(1, "mike", 30), new Student(2, "amy", 15), new Student(3, "james", 55));
- StudentSortMethod ssm = new StudentSortMethod();
- System.out.println("====使用Lamda語法，依姓名排序========");
- Collections.sort(student, (a, b) -> ssm.byName(b, a));
- System.out.println(student.toString());
- System.out.println();
- System.out.println("====使用方法參照語法，依年齡排序========");
- Collections.sort(student, ssm::byAge);
- System.out.println(student.toString());
- //宣告類別
- class Student {
- int id = 0;
- String name = null;
- int age = 0;
- public Student(int id, String name, int age) {
- this.name = name;
- this.id = id;
- this.age = age;
- }
- @Override
- public String toString() {
- String str = String.format("學號：%d 姓名：%s 年齡：%d ", id, name, age);
- return str;
- }
- }
- //用來排序的類別
- class StudentSortMethod {
- //依姓名排序
- public int byName(Student s1, Student s2) {
- return s1.name.compareTo(s2.name);
- }
- //依年齡排序
- public int byAge(Student s1, Student s2) {
- return s1.age - s2.age;
- }
- }

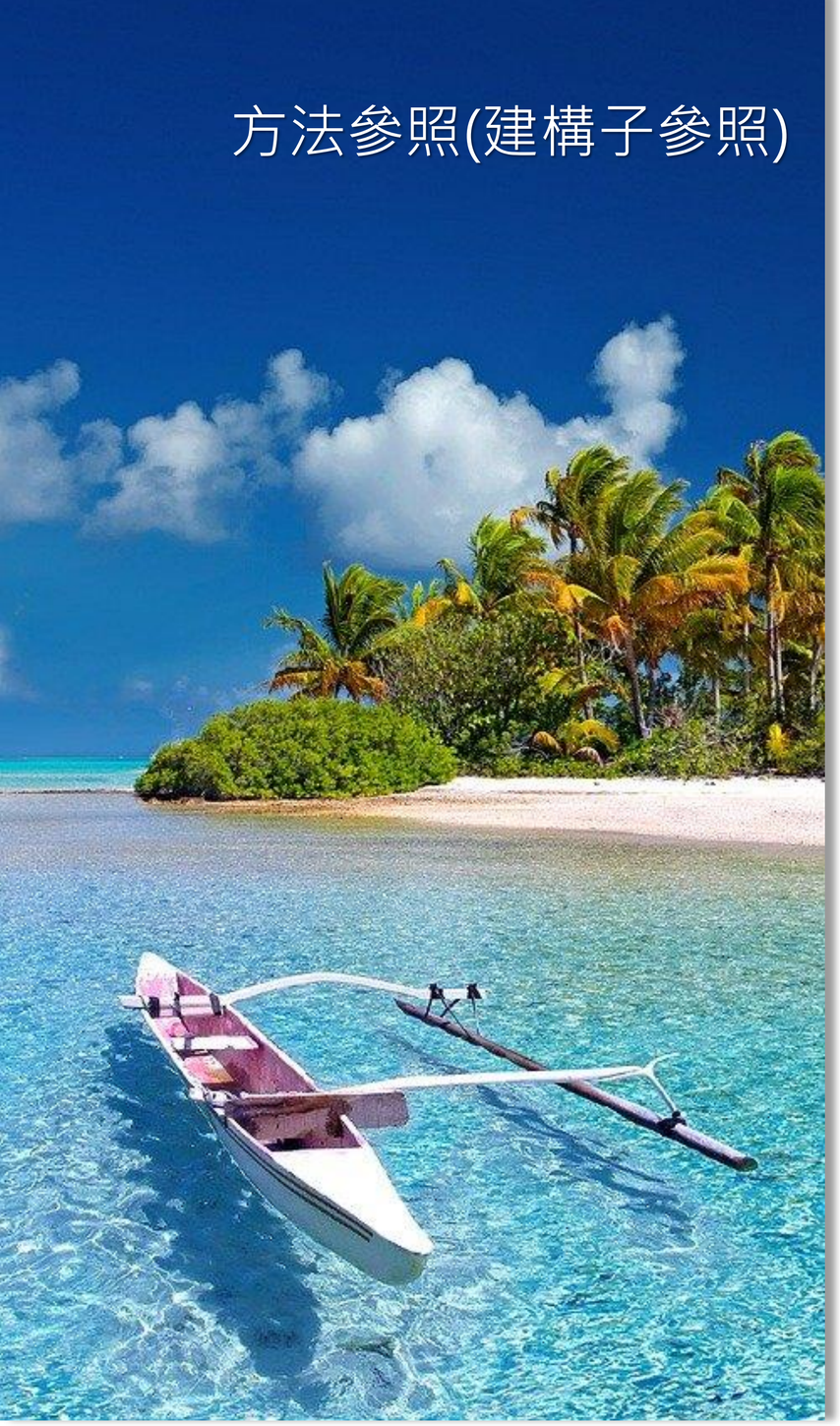
方法參照(類別::物件方法)



- //以下程式貼至主方法內

```
//匿名類別
        Comparator<String> com = new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                return o1.compareTo(o2);
            }
        };
        System.out.println(com.compare("java", "Java"));
        //lambda表達式
        Comparator<String> com1 = (o1, o2) -> o1.compareTo(o2);
        System.out.println(com1.compare("java", "java"));
        //方法參照
        Comparator<String> com2 =String::compareTo;
        System.out.println(com2.compare("java", "java"));
```

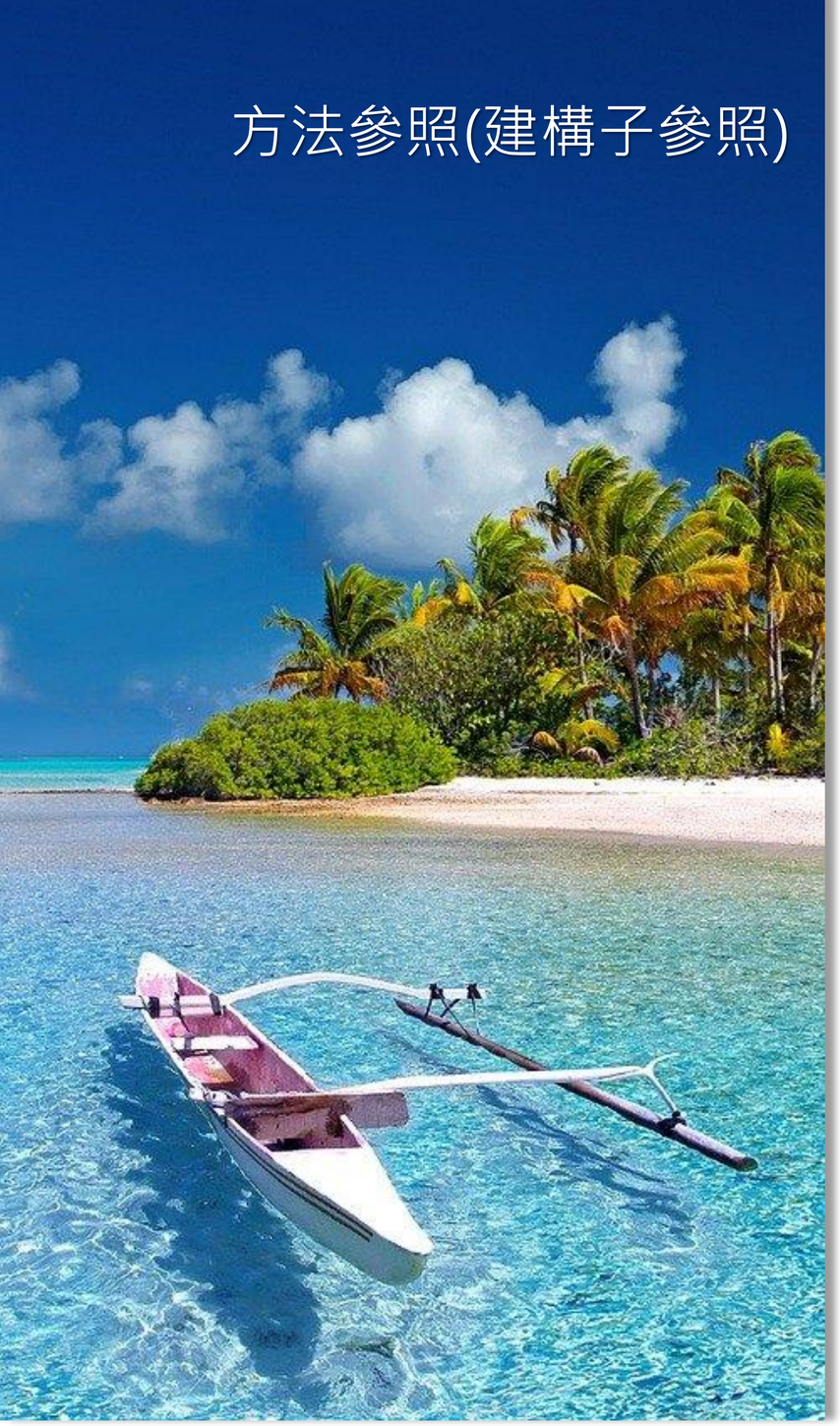
方法參照(建構子參照)



- //以下程式貼至主方法內

```
System.out.println("==========匿名寫法==========");
        Supplier<Student> sup = new Supplier<Student>() {
            @Override
            public Student get() {
                return new Student();
            }
        };
        sup.get();
        System.out.println();

System.out.println("==========Lamda表達式寫法==========");
        Supplier<Student> sup1 = ()->new Student();
        sup1.get();
        System.out.println();

System.out.println("==========建構子參照寫法==========");
        Supplier<Student> sup2 = Student::new;
        sup2.get();
//宣告類別
class Student {
    int id = 0;
    String name = null;
    int age = 0;
    public Student(int id, String name, int age) {
        this.name = name;
        this.id = id;
        this.age = age;
    }
    public Student() {
        System.out.println("我是student沒有參數的建構子");
    }
    @Override
    public String toString() {
        String str = String.format("學號：%d 姓名：%s 年齡：%d  ", id, name, age);
        return str;
    }
}
```

方法參照(建構子參照)

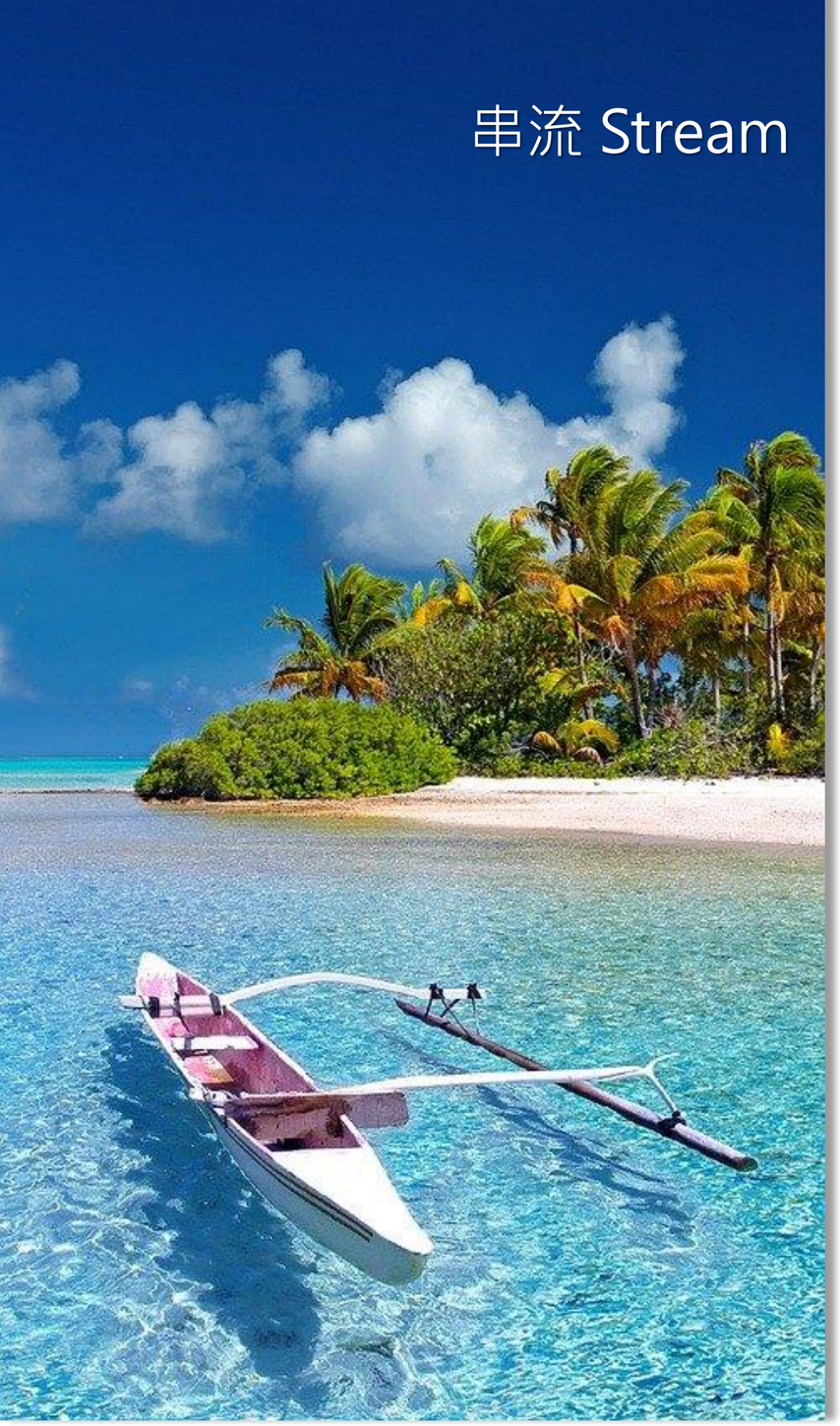


- //以下程式貼至主方法內

```
Supplier<Student> s=new Supplier<>() {
@Override
public Student get() {
return new Student();
}
};
s.get();
System.out.println("=======使用lambda表達式==============");
Supplier<Student> s1=()->new Student();
s1.get();
System.out.println("=======使用方法參照==============");
Supplier<Student> s2=Student::new;
s2.get();
//有一個參數的建構子參照
Function<String,Student> s3=st->new Student(st);
s3.apply("mike");
Function<String,Student> s4=Student::new;
s4.apply("jacky");
//有二個參數的建構子參照
System.out.println("==========有二個參數的建構子參照============");
BiFunction<String,Integer,Student> s5=(x,y)->new Student(x,y);
s5.apply("mike",25);
BiFunction<String,Integer,Student> s6=Student::new;
s6.apply("Linda",35);
//宣告類別
class Student{
String name;
int age;
Student(){
System.out.println("沒有參數的建構子");
}
Student(String name){
this.name=name;
System.out.println("有一個參數的建構子A");
}
Student(String name,int age){
this.name=name;
this.age=age;
System.out.println("有二個參數的建構子A,B");
System.out.println(toString());
}
@Override
public String toString() {
return "Student [姓名=" + name + ", 年齡=" + age + "]";
}
}
```

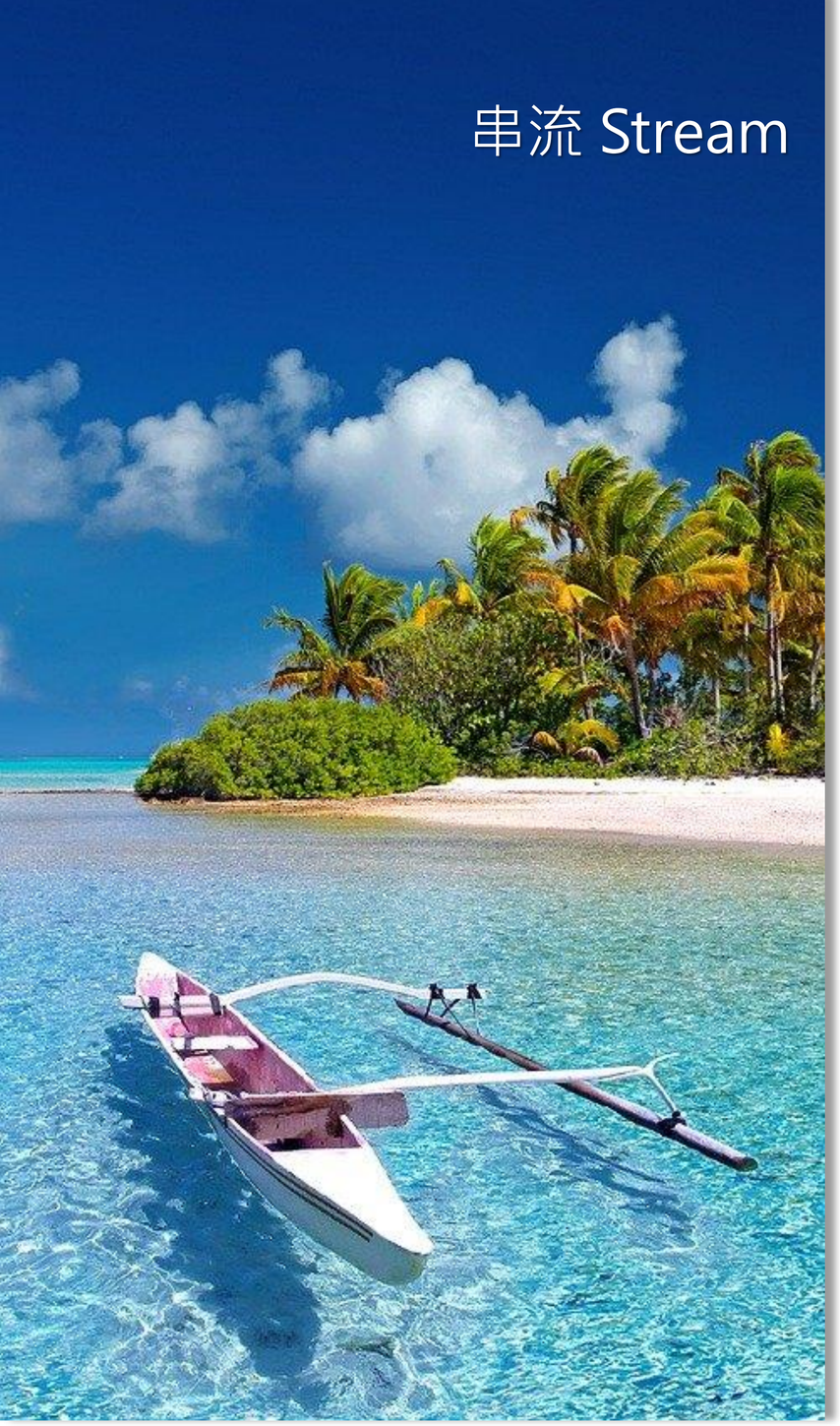
串流 STREAM

串流 Stream



- 集合是用來儲存資料,Stream是用來計算,類似於使用SQL語言針對資料庫的查詢,操作
- Stream自己不會儲存元素
- Stream不會改變來源對象,只是返回一個持有結果的新Stream
- Stream的操作三個步驟
  - 建立Stream:一個資料來源(集合,陣列),取得一個串流
  - 中間操作:一個中間的操作鏈,對資料來源的數據進行處理
  - 終止操作(終端操作):一旦執行終止操作,就執行中間操作鏈,並產生結果,之後,即不會再被使用

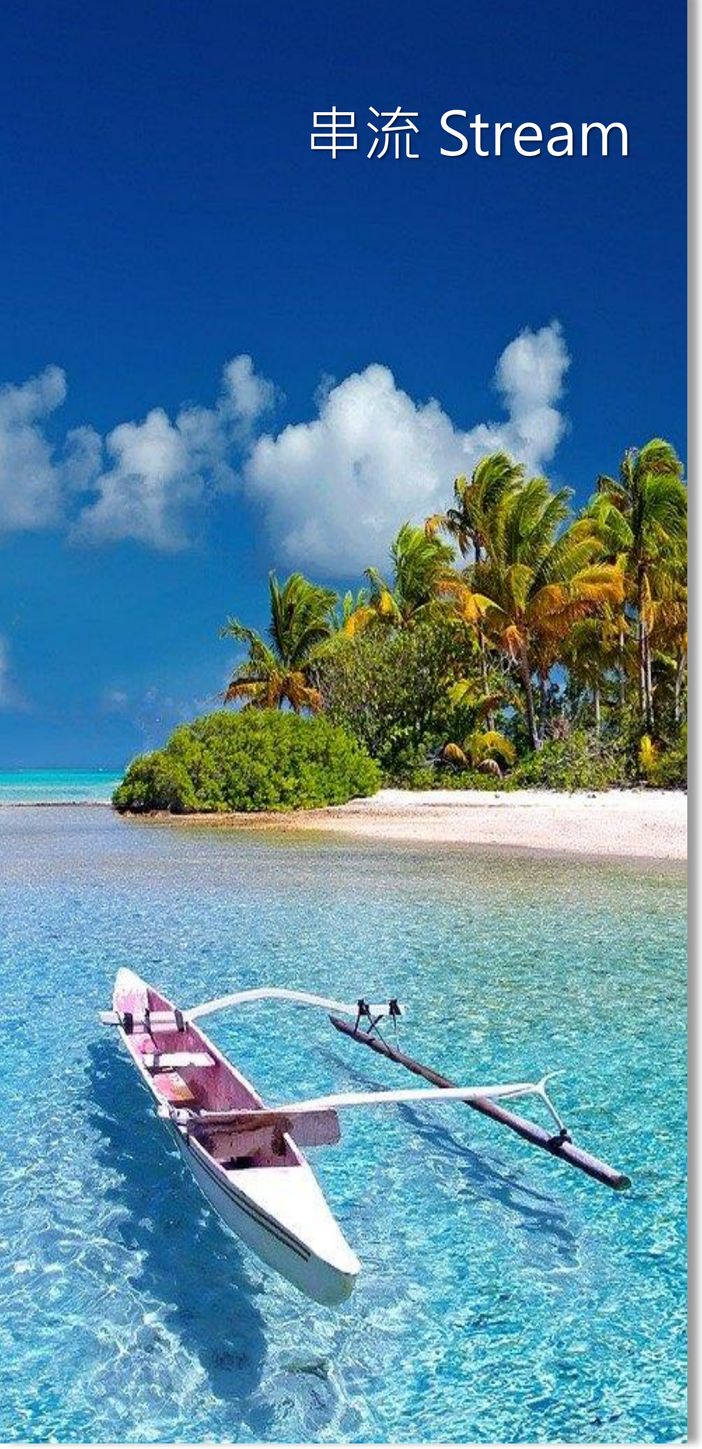
串流 Stream

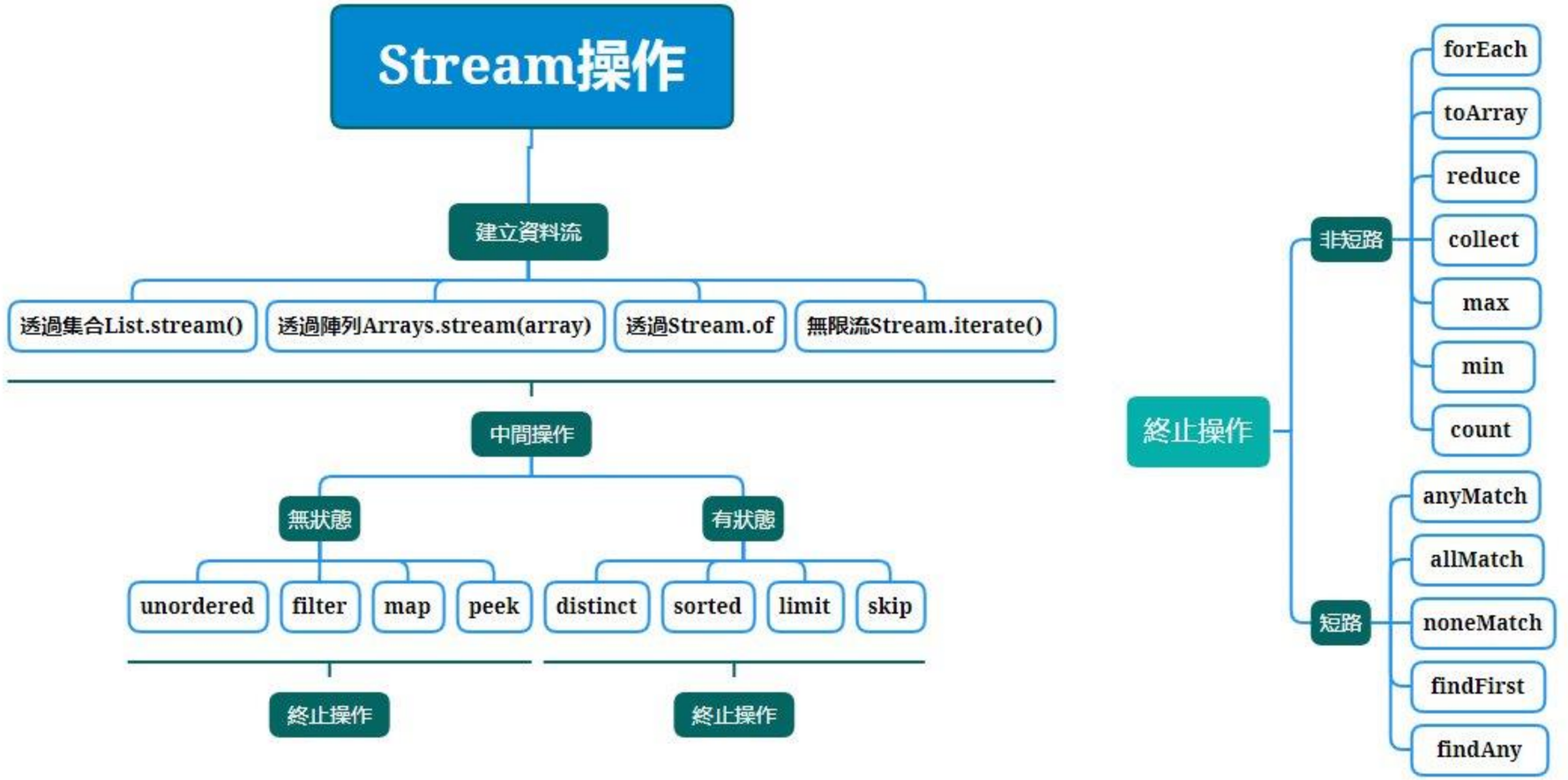


一

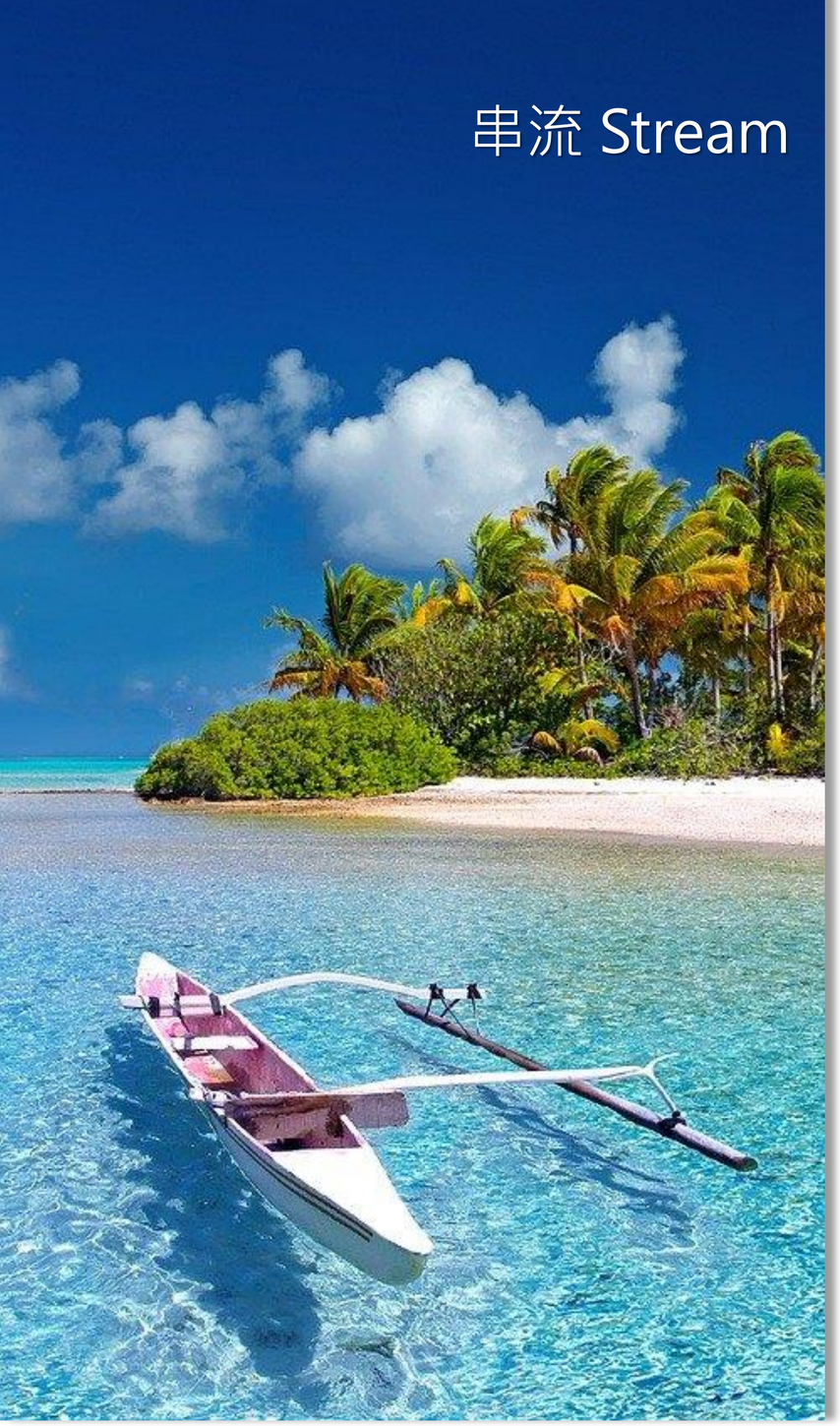
- 中間操作(Intermediate)
  - 無狀態(Stateless):元素的處理不受之前元素的影響
  - 有狀態(Stateful):指該操作只有拿到所有元素後才會再繼續下去
- 終止操作(Terminal)
  - 短路操作(Short-circuiting):指只要有符合條件的元素就可以有輸出 如 A || B,只要A為true則不用再判斷B
  - 非短路操作(Unshort-circuiting):必需處理所有元素才能有輸出

串流 Stream




Stream操作
建立資料流
透過集合List.stream()
透過陣列Arrays.stream(array)
透過Stream.of
無限流Stream.iterate()
中間操作
無狀態
有狀態
unordered
filter
map
peek
distinct
sorted
limit
skip
終止操作
終止操作
終止操作
非短路
短路
forEach
toArray
reduce
collect
max
min
count
anyMatch
allMatch
noneMatch
findFirst
findAny

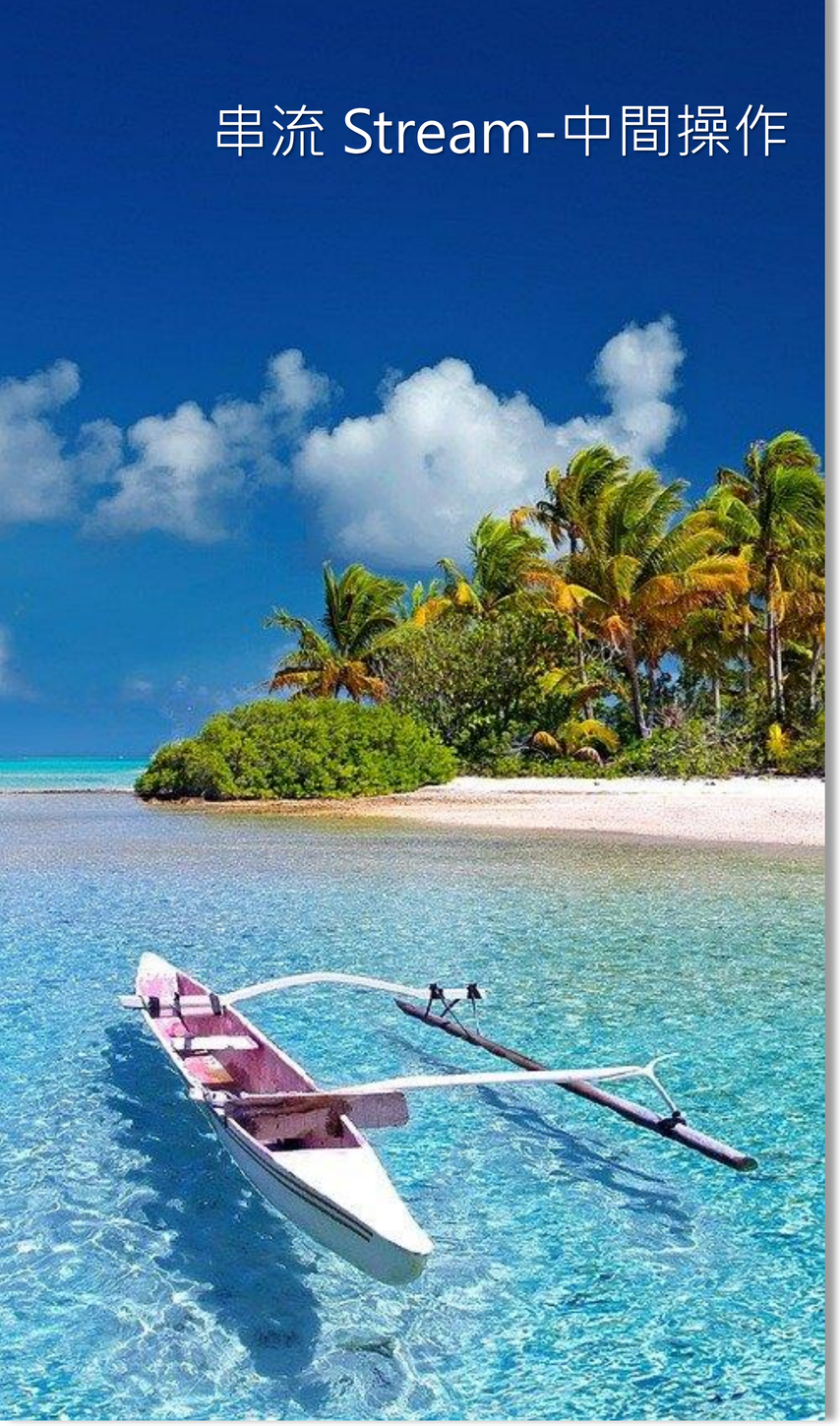
串流 Stream



一

- 建立Stream
  - ■透過集合
    - List<String> list=Arrays.asList("mike","amy","linda","jackson","kenny");
    - Stream<String> stream=list.stream();
  - ■透過陣列
    - int[] arr={1,2,3,4,5,6,7,8};
    - IntStream stream1 = Arrays.stream(arr);
  - ■透過Stream的of( )方法
    - Stream<Integer> stream2 = Stream.of(1,2,3,4,5,6,7,8);
  - ■無限串流
    - 透過巡訪器
      - Stream.iterate(0,t->t+2).limit(11).forEach(System.out::println);
    - 透過產生器
      - Stream.generate(Math::random).limit(11).forEach(System.out::println);

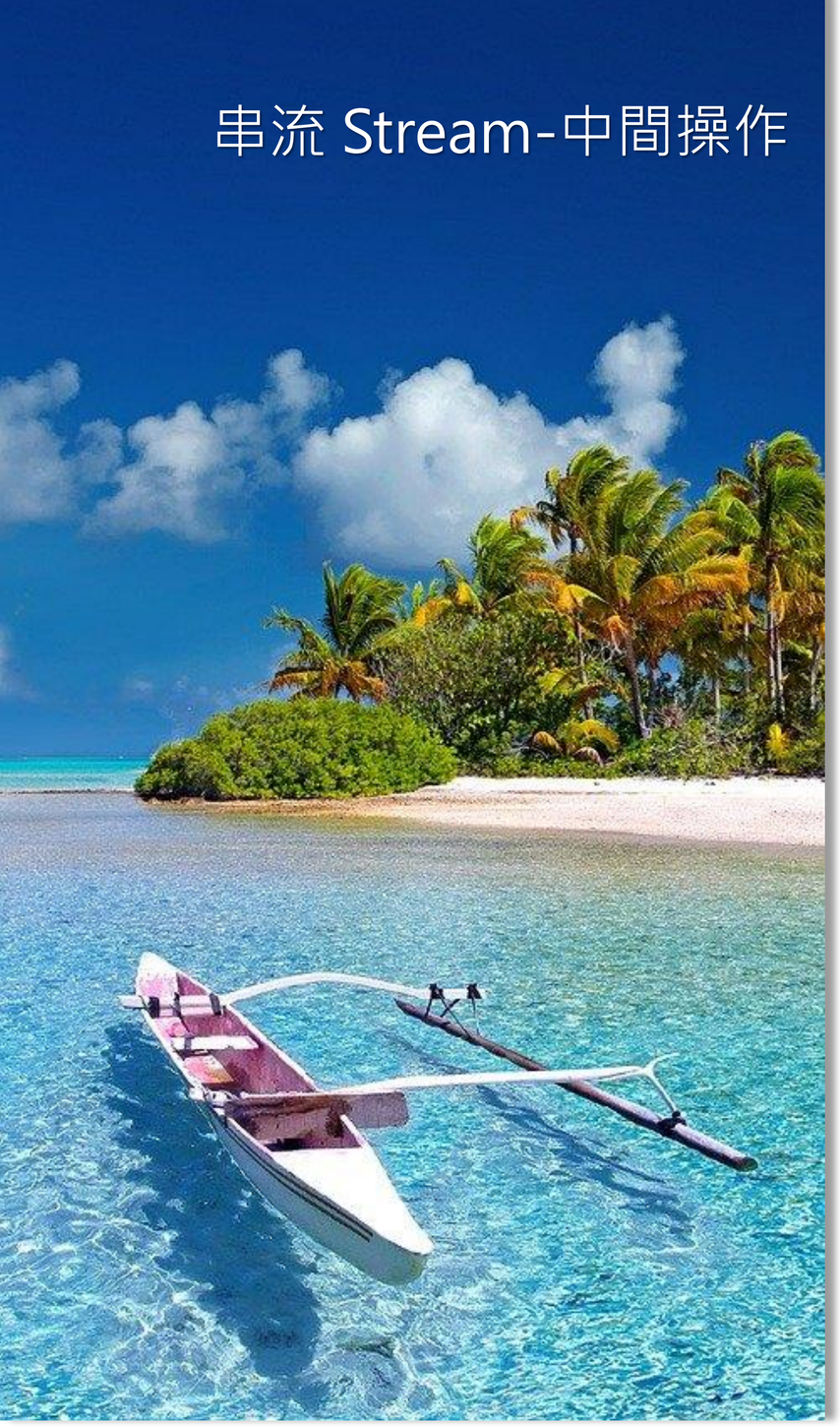
串流 Stream-中間操作



- 中間操作
  - Filter
  - Limit
  - Skip
  - distinct

```
String str[]={"mike","amy","tom","jimmy","linda","jason","james","james","linda"};
Stream<String> stream = Arrays.stream(str);
stream.filter(s->s.contains("i")).forEach(System.out::println);
System.out.println("==============================");
stream = Arrays.stream(str);
stream.filter(s->s.contains("i")).limit(1).forEach(System.out::println);
System.out.println("==============================");
stream = Arrays.stream(str);
stream.filter(s->s.contains("i")).skip(1).forEach(System.out::println);
System.out.println("==============================");
stream = Arrays.stream(str);
stream.distinct().forEach(System.out::println);

//forEach()自訂輸出 forEach(s->System.out.print(s+"\t"));無換行之寫法
```

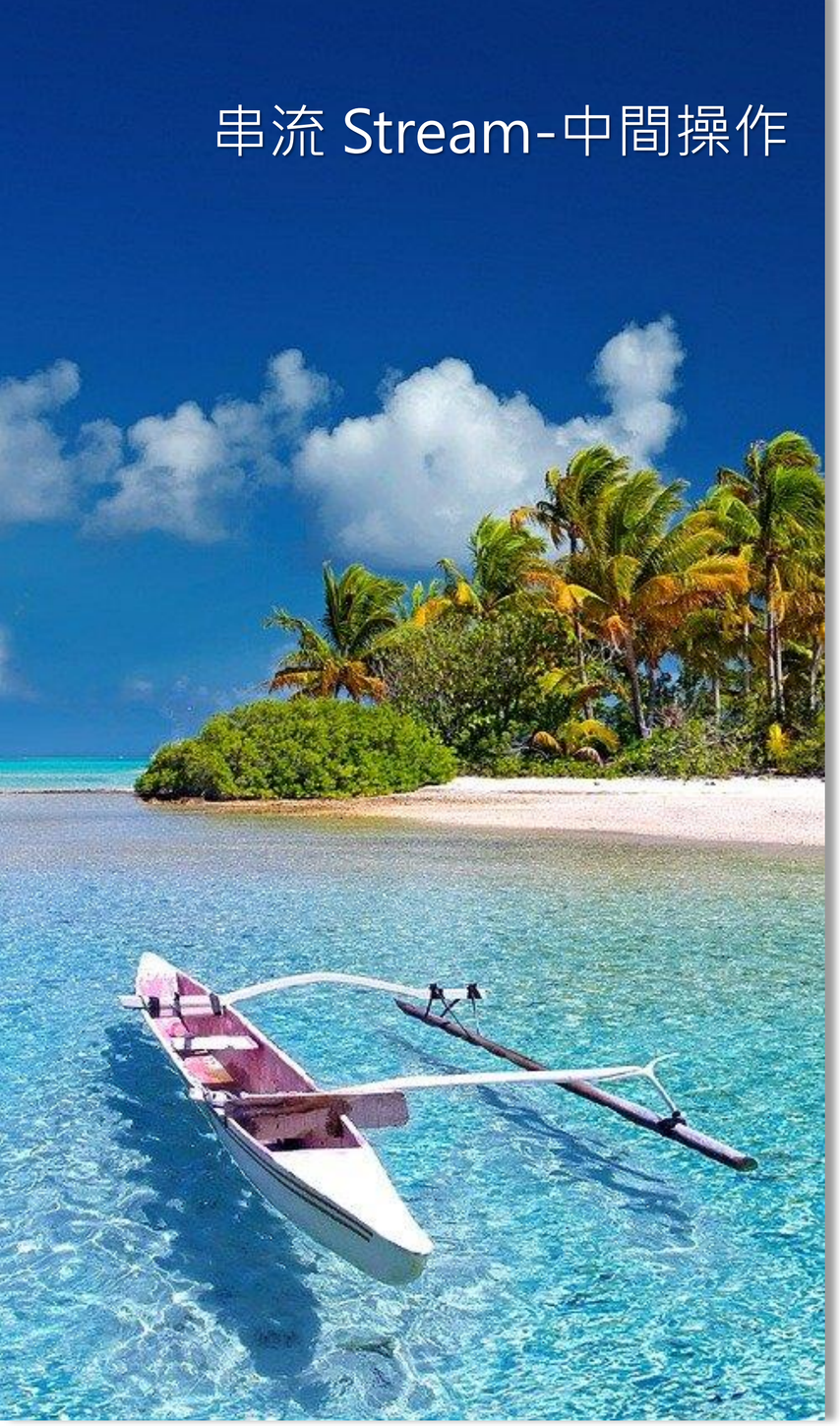
串流 Stream-中間操作



- 中間操作
  - map
  - flatMap

```
List<String> list = Arrays.asList("mike", "amy", "linda", "jackson", "kenny", "linda");
        list.stream().map(s -> s.toUpperCase()).forEach(s->{
            System.out.print(s+"\t");
        });
        System.out.println();
        System.out.println("====印出長度超過4個字的姓名,並轉成大寫========");
        list.stream().filter(s->s.length()>4).map(s ->
s.toUpperCase()).forEach(System.out::println);
```

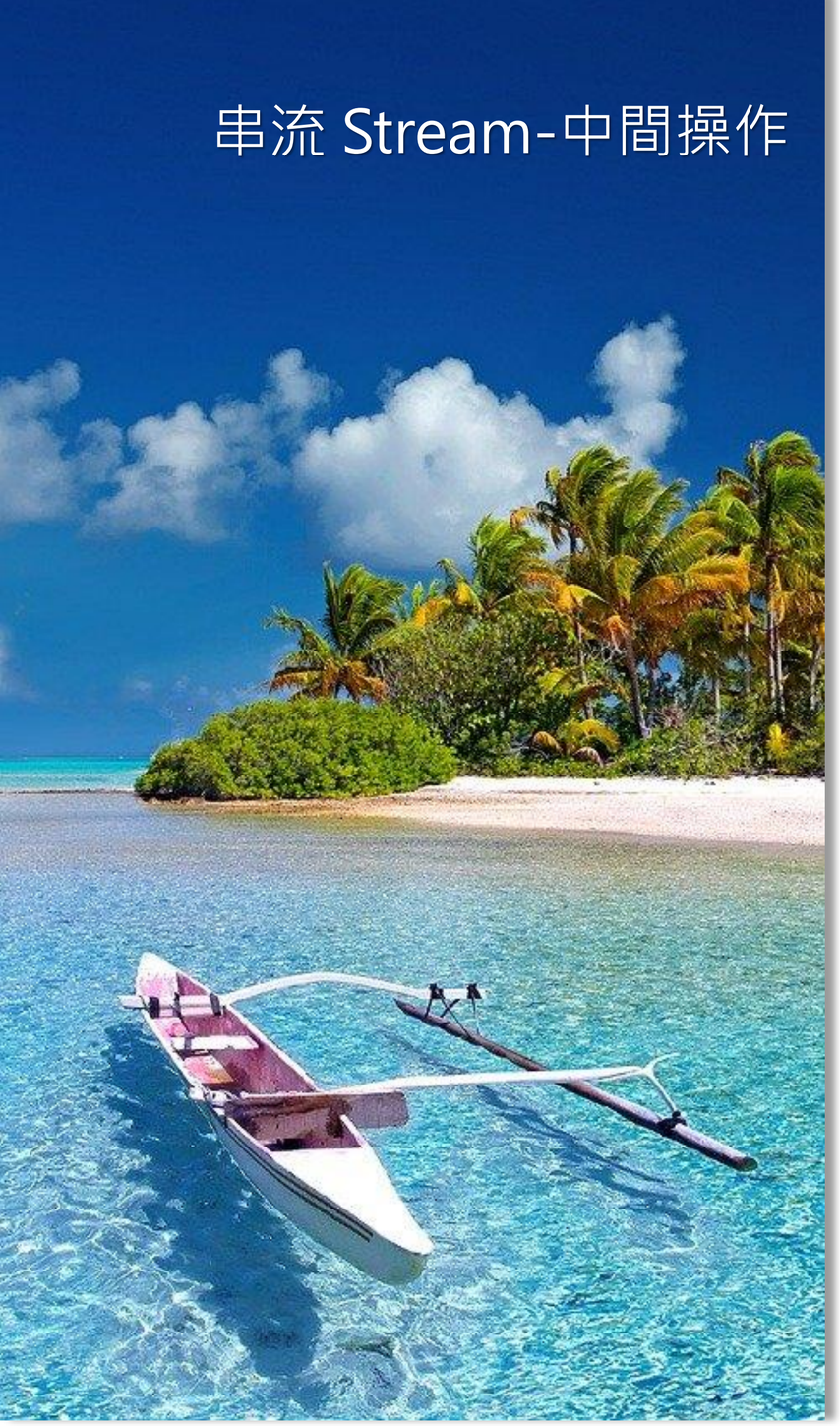
串流 Stream-中間操作



- 中間操作
  - map
  - flatMap

```
List<String> list = Arrays.asList("mike", "amy", "linda", "jackson", "kenny");
        Stream<Stream<Character>> map = list.stream().map(LamdaTest::fromStringToStream);
        //map.forEach(System.out::println)印出物件,此時map還是Stream物件
        map.forEach(s -> {
            s.forEach(System.out::println);
        });
        System.out.println("====利用flatMap================");
//方法參照 類別::靜態方法
        Stream<Character> flatMap = list.stream().flatMap(LamdaTest::fromStringToStream);
        flatMap.forEach(System.out::println);
    }

    public static Stream<Character> fromStringToStream(String str) {
        ArrayList<Character> list = new ArrayList<>();
        for (Character c : str.toCharArray()) {
            list.add(c);
        }
        return list.stream();
    }
```

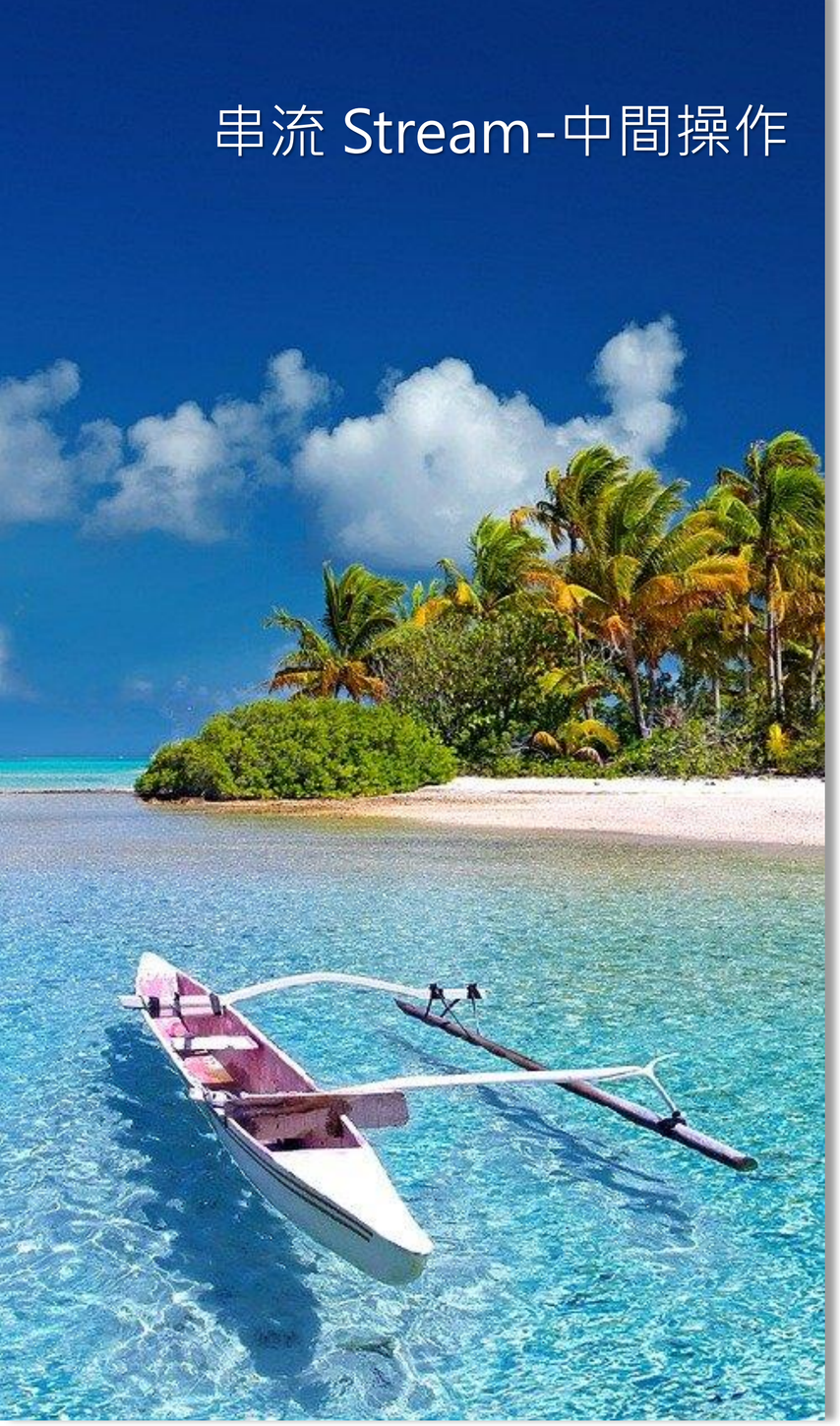
串流 Stream-中間操作



- 中間操作
  - sorted

```
List<Integer> list = Arrays.asList(11, 55, 23, 66, 33, 21, 99, 10, 8, 56);
        list.stream().sorted().forEach(s -> System.out.print(s + "\t"));
        System.out.println("\n============================\n");
        list.stream().sorted((o1, o2) -> Integer.compare(o2, o1)).forEach(s ->
System.out.print(s + "\t"));
        System.out.println();
```

串流 Stream-中間操作

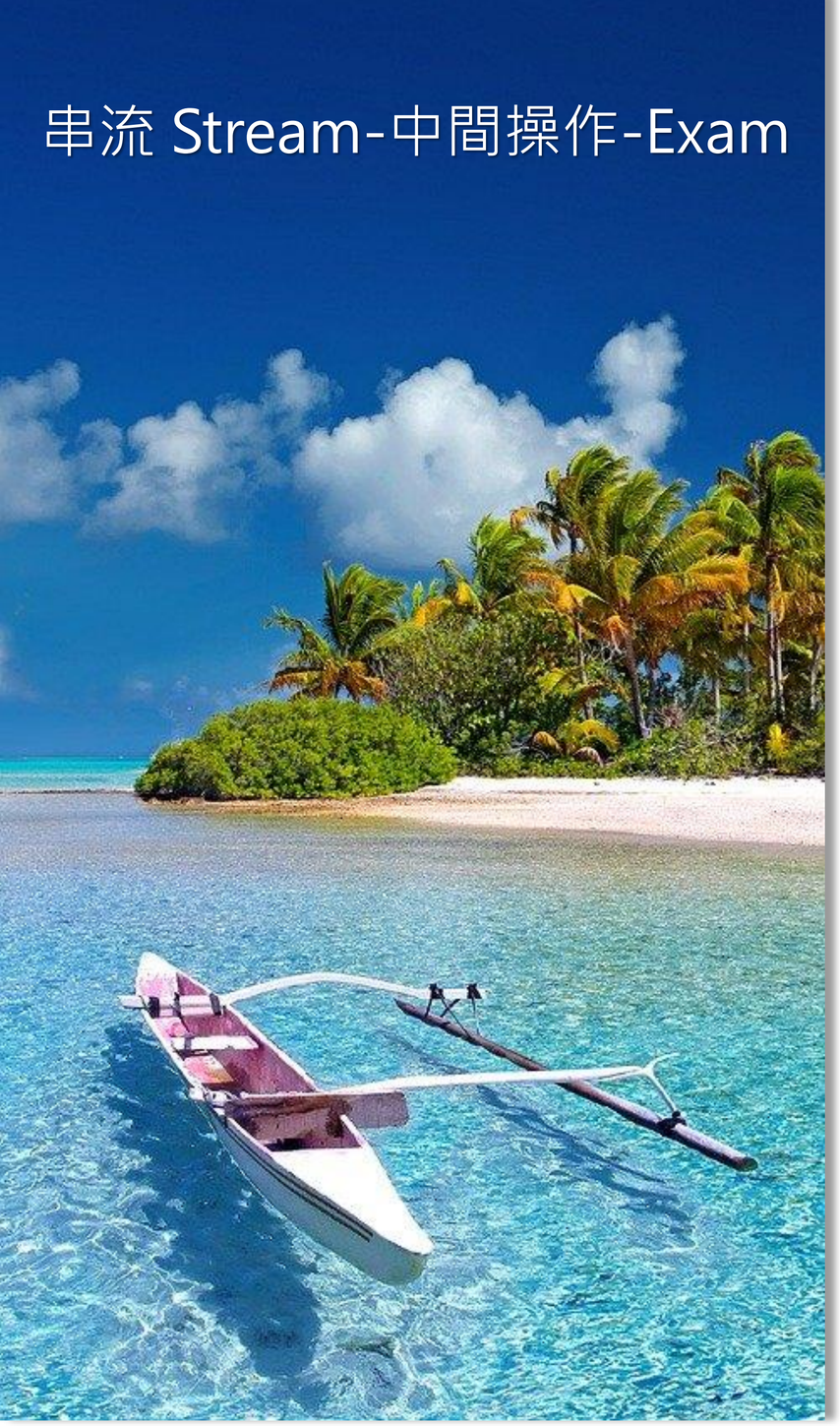


- 中間操作
  - peek

- //peek是一個中間操作,若沒有終止操作是不會有輸出的,其將結果變成一個新的流,如下範例
- Stream<Integer> peek = Stream.of(10,20,30).peek(System.out::println);
- peek.collect(Collectors.toList());

- List<Integer> collect = Stream.of(50,11,1,13)
- .peek(e->System.out.println("peek值:"+e))
- //.filter(x->x>=13)…
- //.sorted()
- .peek(System.out::println)
- .collect(Collectors.toList());
- System.out.println(collect);
- //對於count終止操作無作用

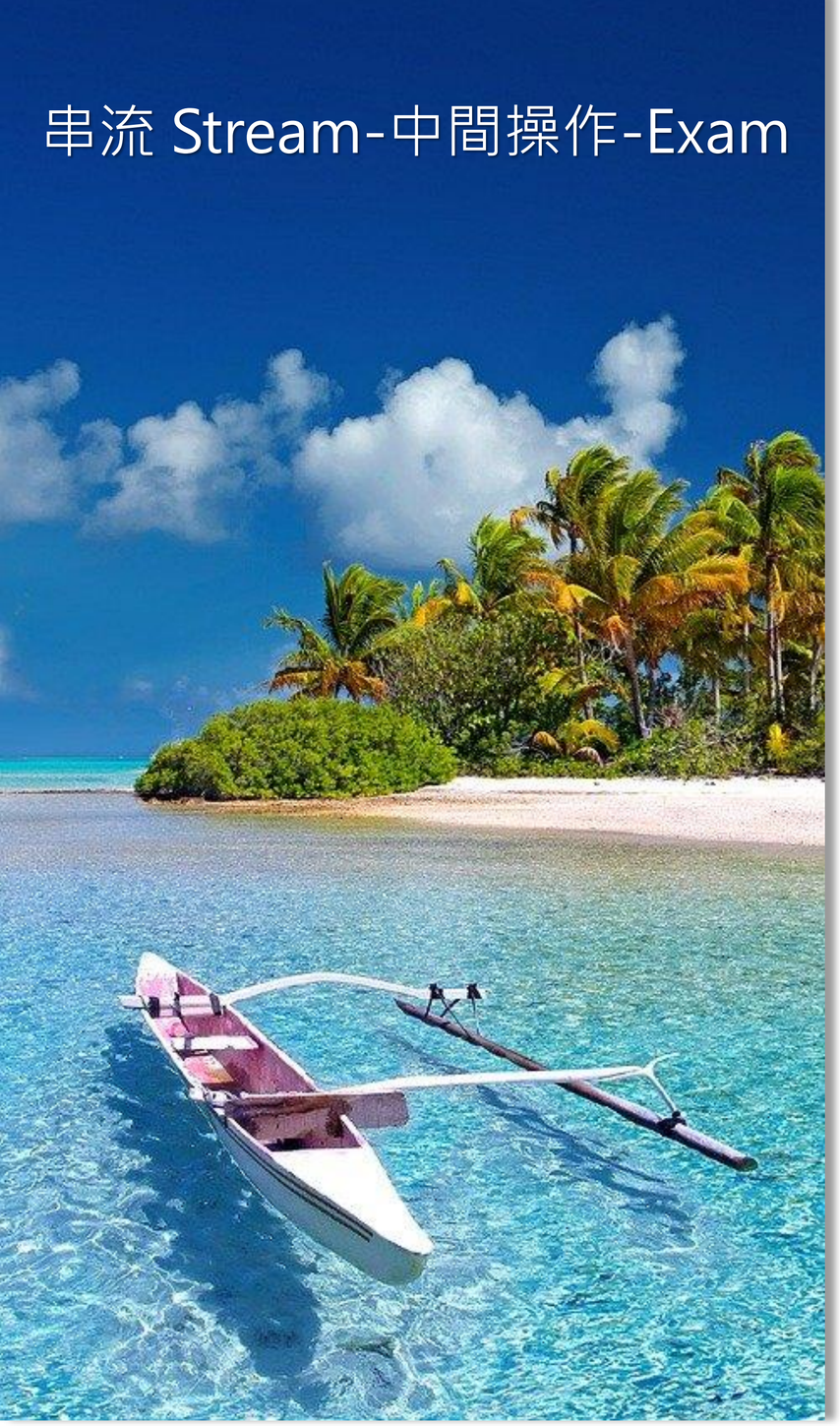
串流 Stream-中間操作-Exam



Why would you choose to use a peek operation instead of a forEach operation on a Stream ?

- A. to remove an item from the end of the stream
- B. to remove an item from the beginning of the stream
- C. to process the current item and return void
- D. to process the current item and return a stream

串流 Stream-中間操作-Exam



```
public class Final08 {
    public static void main(String[] args) {
        List<String> fruits=List.of("banana","orange","apple","lemon");
        Stream<String> s1 = fruits.stream();
        Stream<String> s2 = s1.peek(i->System.out.print(i+" "));
        System.out.println("----");
        Stream<String> s3 = s2.sorted();
        Stream<String> s4 = s3.peek(i->System.out.print(i+" "));
        System.out.println("----");
        String strFruits=s4.collect(Collectors.joining(","));
    }
}
```

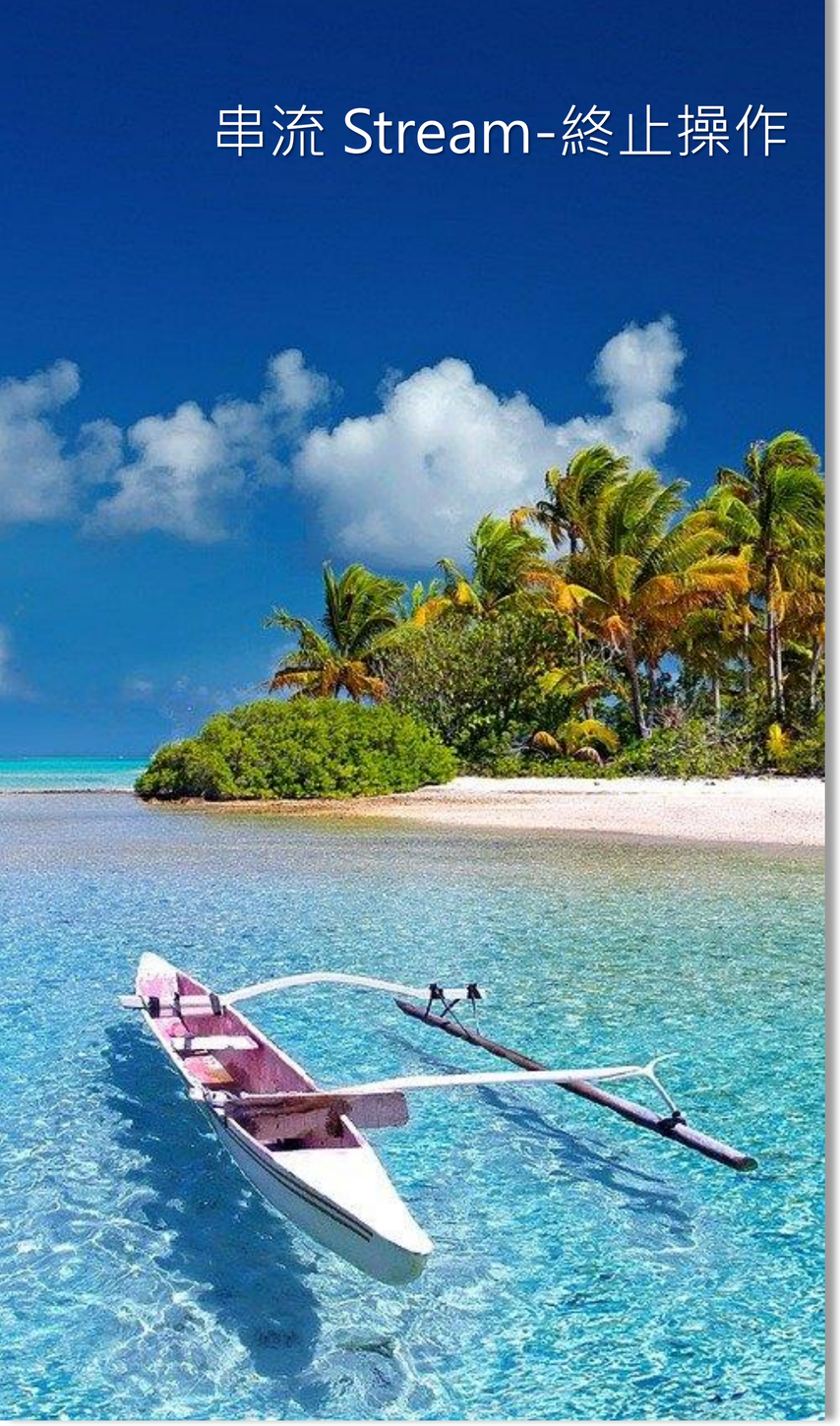
A)
```
-----
-----
```

B)
```
-----
-----
banana orange apple lemon apple banana lemon orange
```

C)
```
-----
banana orange apple lemon
-----
apple banana lemon orange
```

D)
```
banana orange apple lemon
-----
apple banana lemon orange
-----
```

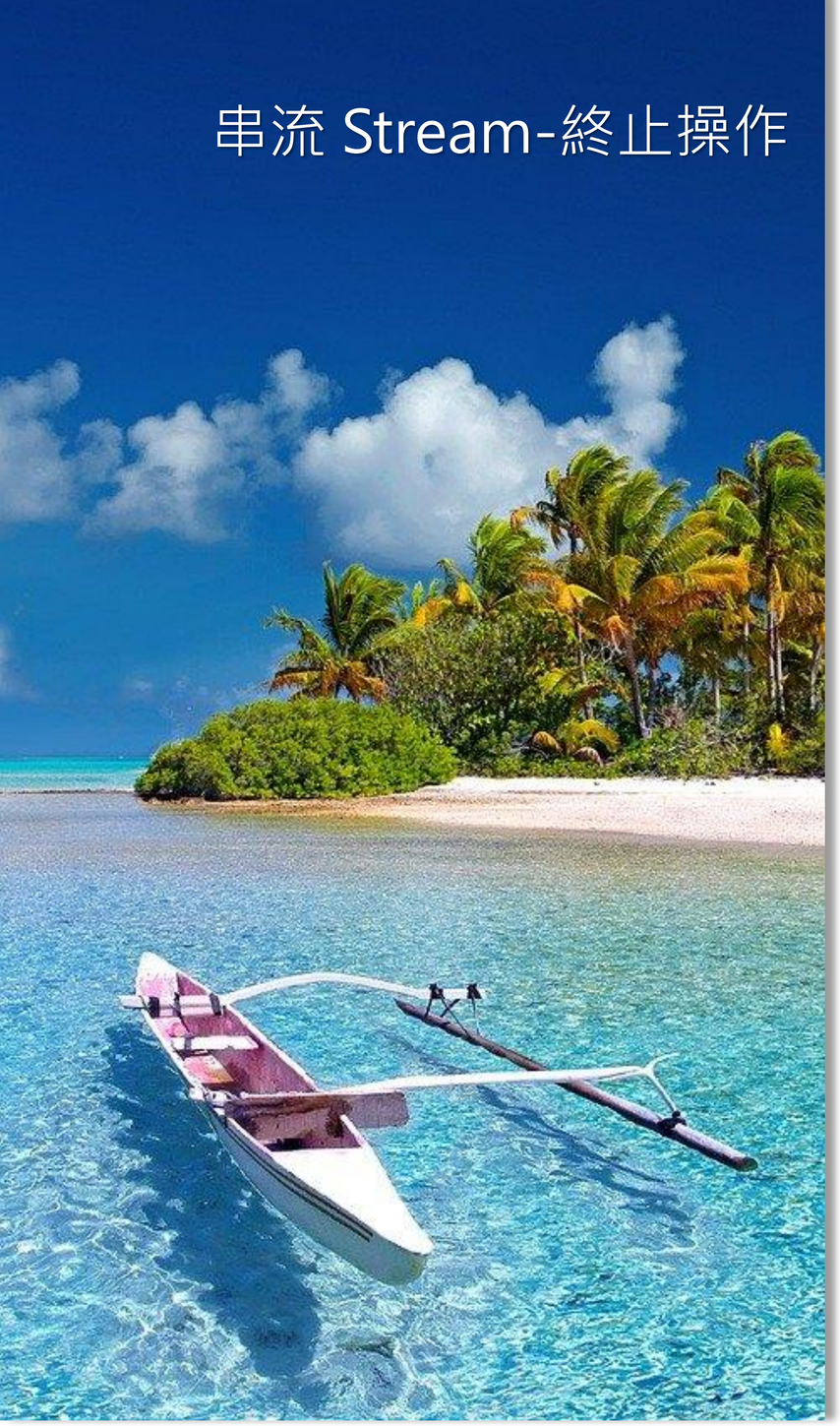
串流 Stream-終止操作



- 終止操作
  - allMatch
  - anyMatch
  - noneMatch
  - findFirst
  - findAny

```
List<Integer> list = Arrays.asList(11, 55, 23, 66, 33, 21, 99, 10, 8, 56);
        System.out.println("=========皆小於50======");
        boolean allMatch = list.stream().allMatch(s->s<50);
        System.out.println(allMatch);
        System.out.println("=========只要有小於50======");
        boolean anyMatch = list.stream().anyMatch(s->s<50);
        System.out.println(anyMatch);
        System.out.println("=========沒有小於50======");
        boolean noneMatch = list.stream().noneMatch(s->s<50);
        System.out.println(noneMatch);
        System.out.println("=========找到第一個元素======");
        Optional<Integer> findFirst = list.stream().findFirst();
        System.out.println(findFirst);
        System.out.println("=========回傳串流中數目的總數======");
        long count = list.stream().filter(s->s>=50).count();
        System.out.println(count);
        System.out.println("=========回傳串流中最大值======");
        Optional<Integer> max = list.stream().max(Integer::compare);
        System.out.println(max);
        System.out.println("=========回傳串流中最小值======");
        Optional<Integer> min = list.stream().min(Integer::compare);
        System.out.println(min);
```

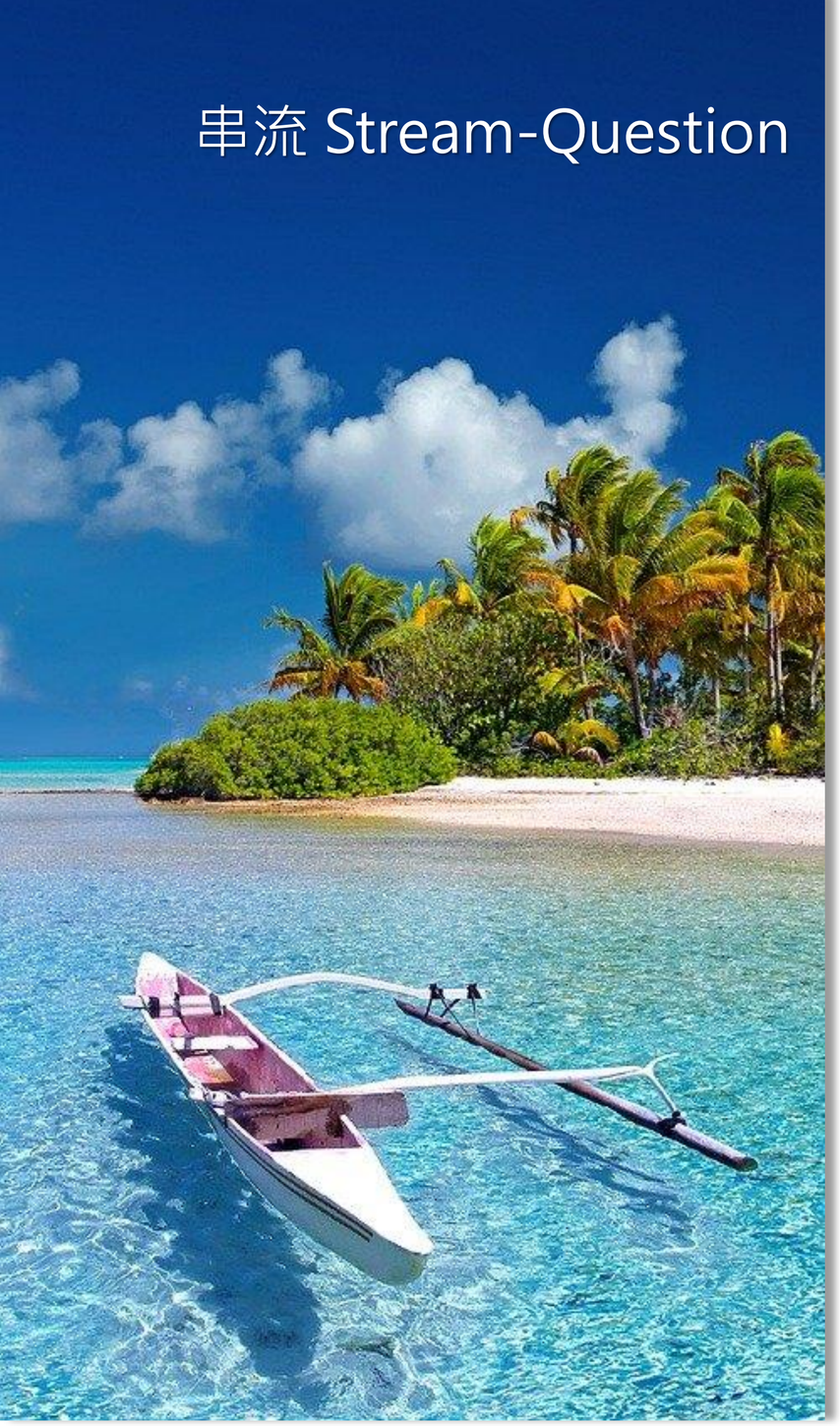
串流 Stream-終止操作



- 終止操作
  - reduce
  - collect

```
List<Integer> list = Arrays.asList(1, 2, 3, 4,5, 6, 7, 8, 9, 10);
        System.out.println("============總結================");
        Integer reduce = list.stream().reduce(0,(t1,t2)->Integer.sum(t1, t2));
        System.out.println(reduce);
        System.out.println("============收集================");
        List<Integer> list2 = list.stream().filter(t->t>=5).collect(Collectors.toList());
        list2.forEach(System.out::println);
        Set<Integer> set = list.stream().filter(t->t>=5).collect(Collectors.toSet());
        set.forEach(System.out::println);
```

串流 Stream-Question

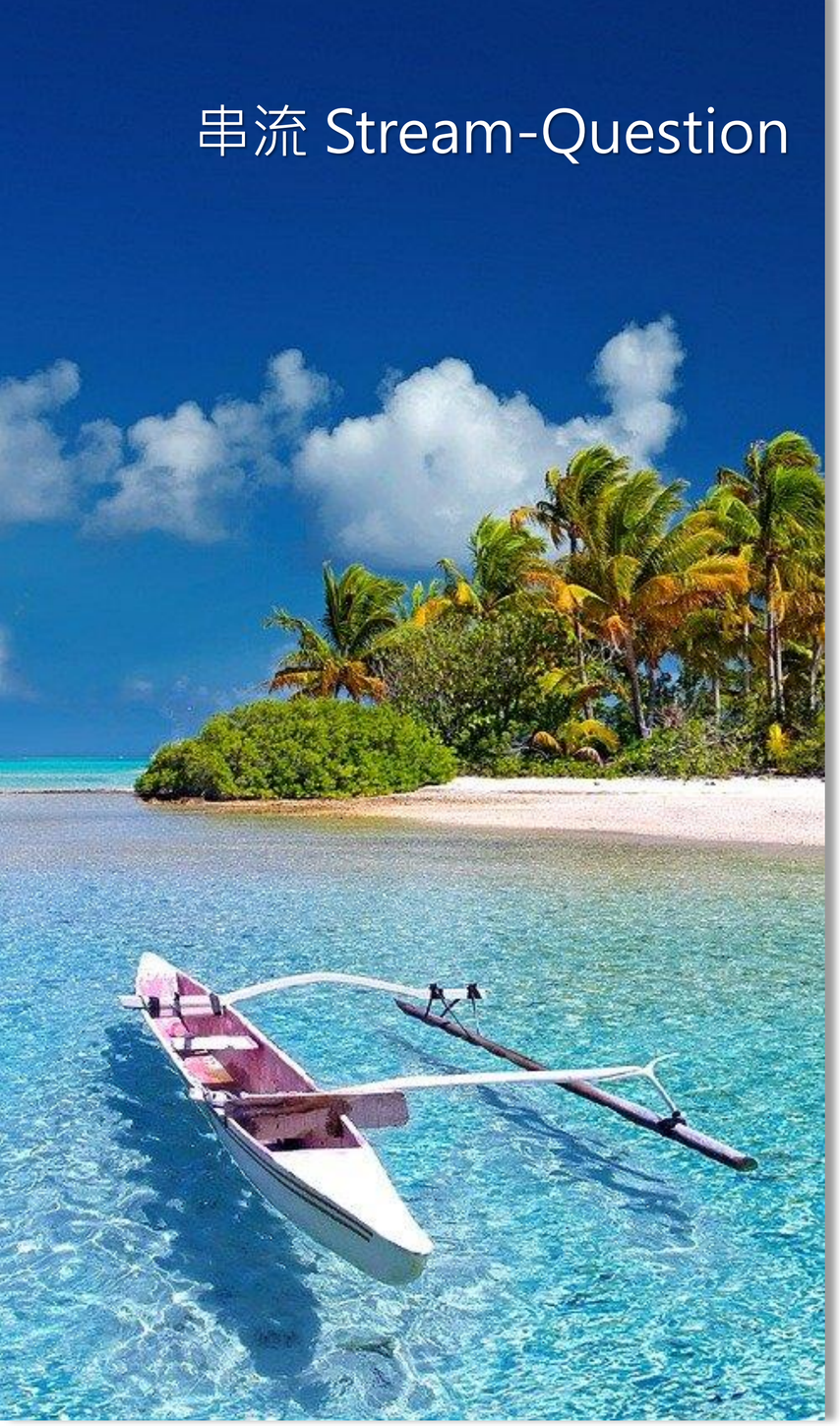


What is the result ? *

```
List<String> empDetails = Arrays.asList("100, Robin, HR",
        "200, Mary, AdminServices",
        "101, Peter, HR");
empDetails.stream()
        .filter(s -> s.contains("1"))
        .sorted()
        .forEach(System.out::println); //line n1
```

- ○ A.100,Robin,HR 換行 101,Peter,HR
- ○ B.A compilation error occurs at line n1.
- ○ C.100,Robin,HR 換行 101,Peter,HR 換行 200,Mary,AdminServices
- ○ D.100,Robin,HR 換行 200,Mary,AdminServices 換行 101,Peter,HR

串流 Stream-Question



What is the result ?

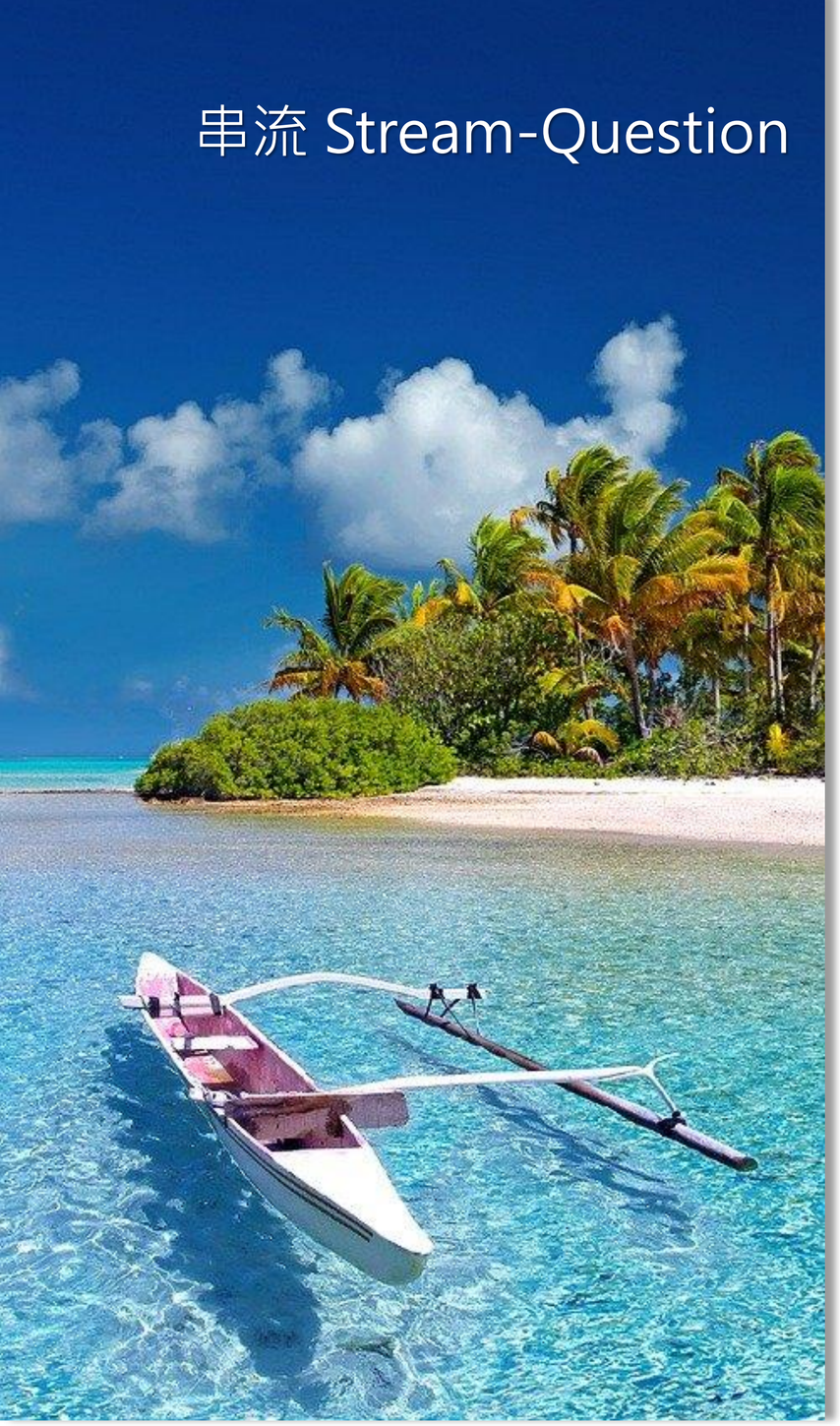
```
public class Emp {
    private String eName;
    private Integer eAge;

    Emp(String eN, Integer eA) {
        this.eName = eN;
        this.eAge = eA;
    }
    public Integer getEAge() {
        return eAge;
    }
     public String getEName() {
        return eName;
    }
}
```

and code fragment:

```
List<Emp> li = Arrays.asList(new Emp("Sam", 20), new Emp("John", 60), new
Emp("Jim", 51));
Predicate<Emp> agVal = s -> s.getEAge() > 50; //line n1
li = li.stream().filter(agVal).collect(Collectors.toList());
Stream<String> names = li.stream().map(Emp::getEName); //line n2
names.forEach(n -> System.out.print(n + " "));
```

- ○ A. Sam John Jim
- ○ B. John Jim
- ○ C. A compilation error occurs at line n1.
- ○ D. A compilation error occurs at line n2.

串流 Stream-Question

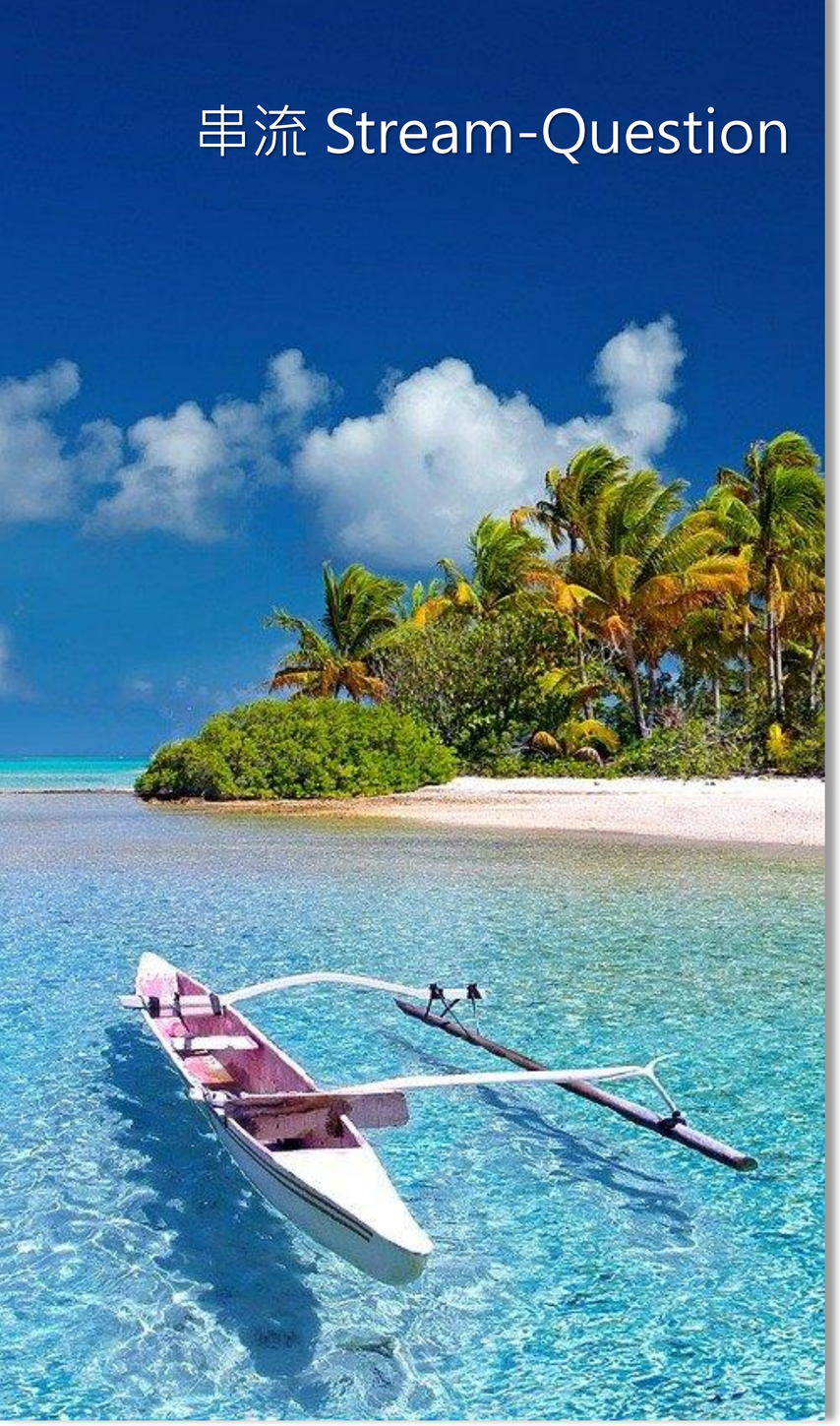


Which code fragment, when inserted at line n1, enables the code to print the count of string elements whose length is greater than three?

```
List<String> listVal = Arrays.asList("Joe", "Paul", "Alice", "Tom");

System.out.println(

    // line n1

);
```

- A. listVal.stream().filter(x -> x.length() > 3).count()
- B. listVal.stream().map(x -> x.length() > 3).count()
- C. listVal.stream().peek(x -> x.length() > 3).count().get()
- D. listVal.stream().filter(x -> x.length() > 3).mapToInt(x -> x).count()

串流 Stream-Question



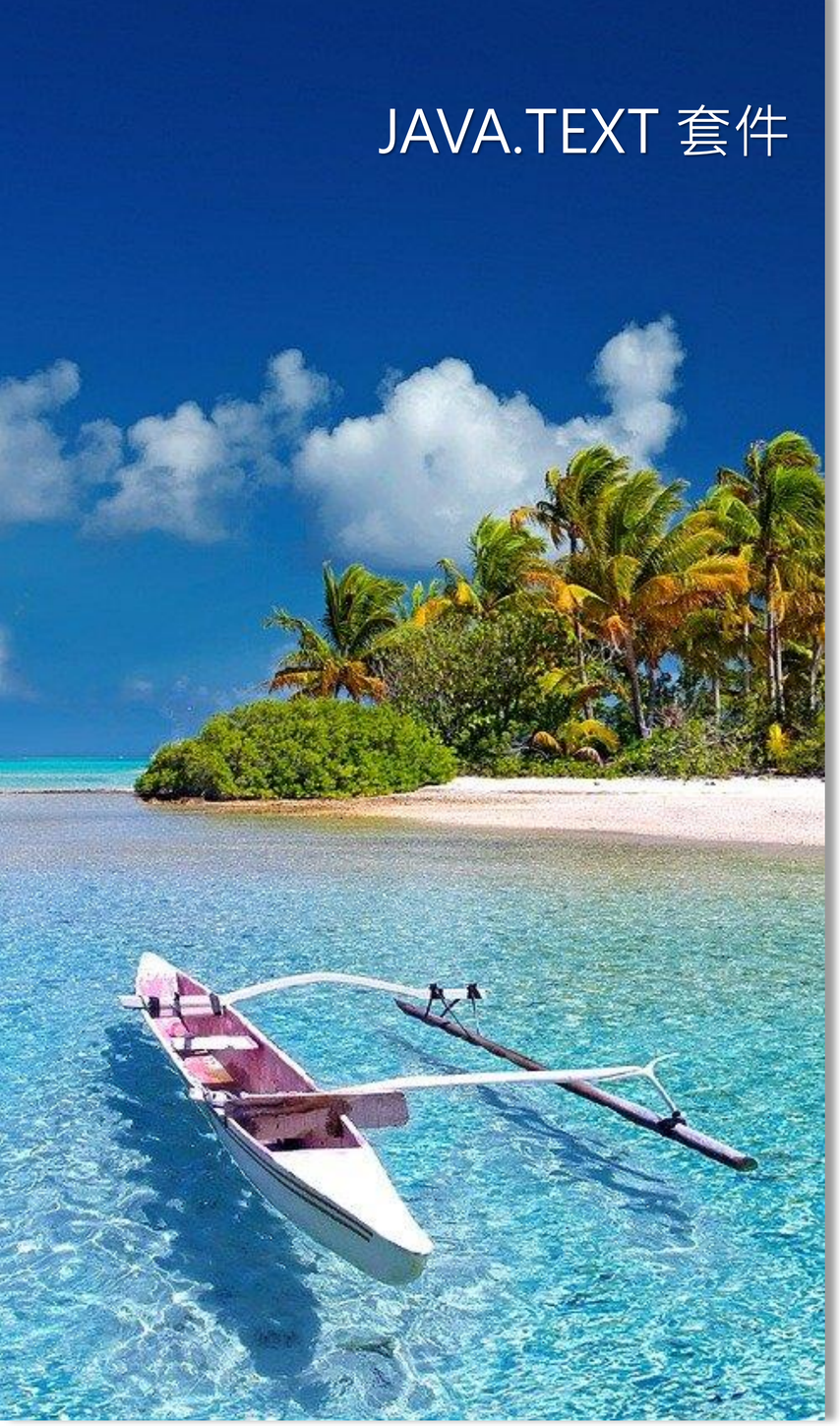
What is the result ?

```
List<String> colors = Arrays.asList("red", "green", "yellow");
Predicate<String> test = n -> {
        System.out.println("Searching...");
        return n.contains("red");
};
colors.stream()
                .filter(c -> c.length() > 3)
                .allMatch(test);
```

- ○ A. Searching...
- ○ B. Searching... 換行 Searching...
- ○ C. Searching... 換行 Searching... 換行 Searching...
- ○ D. A compilation error occurs.

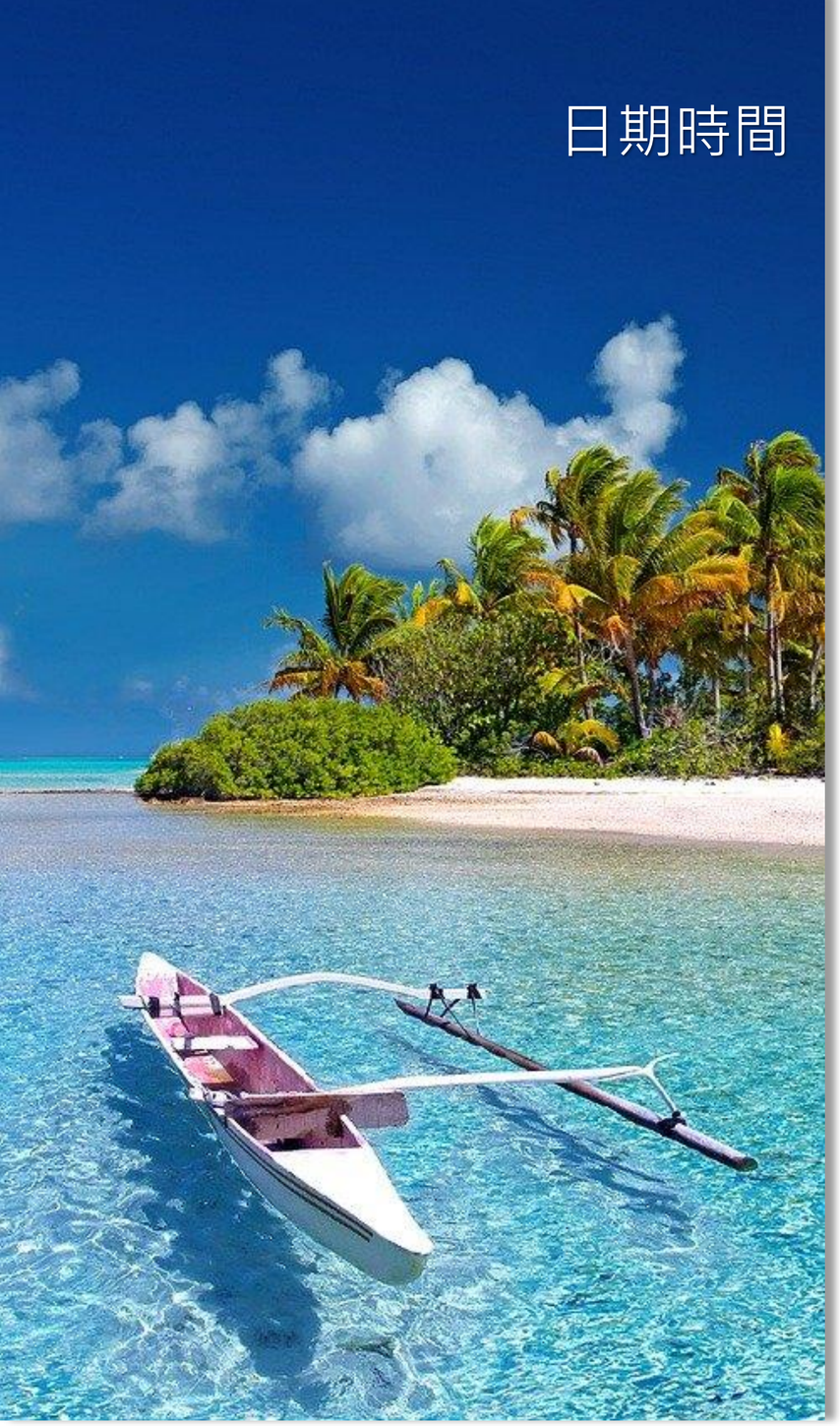
日期時間與本地化 DATETIME LOCALE

JAVA.TEXT 套件



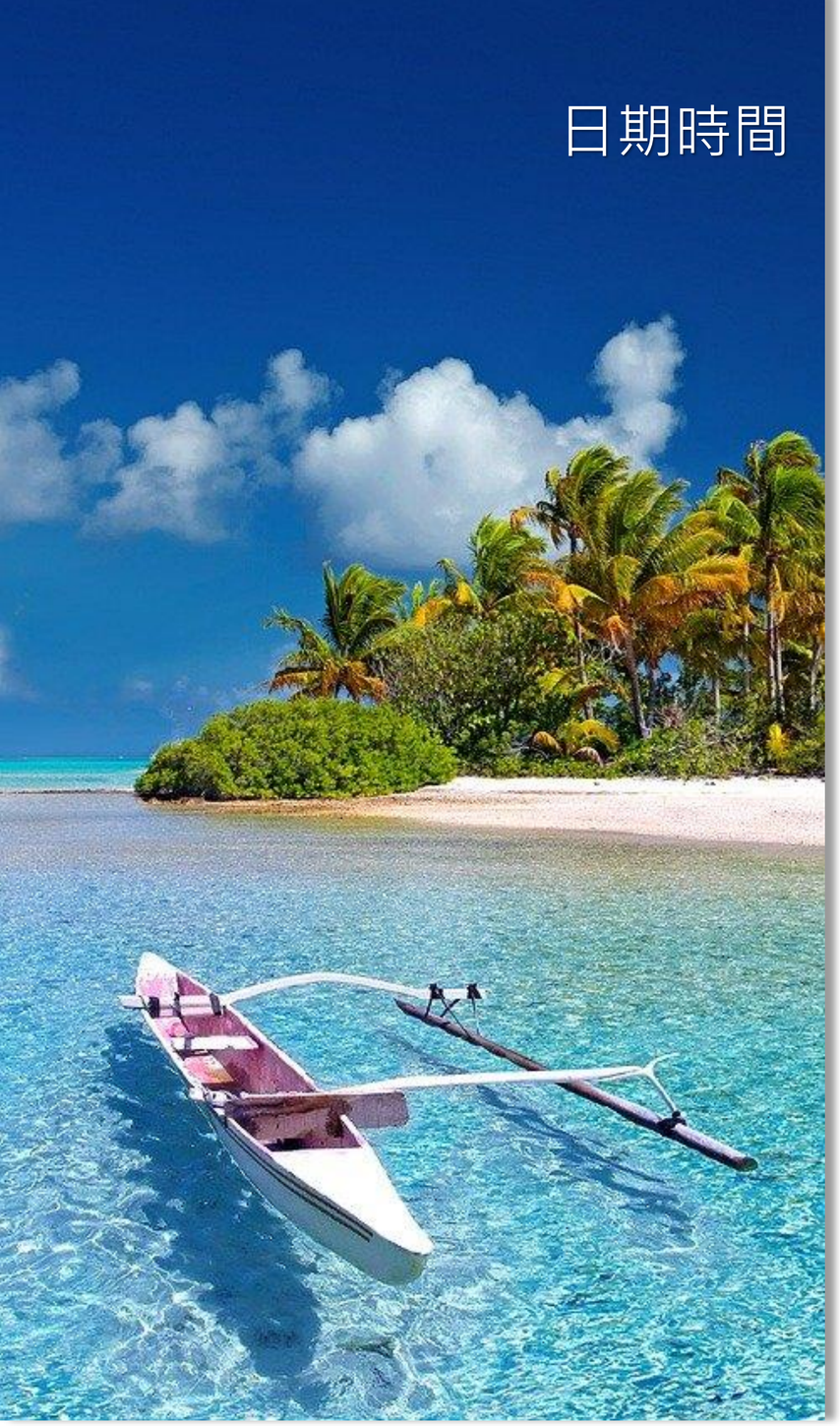
- 此套件中提供了許多關於格式的類別 常用於格式轉換有:貨幣 日期 訊息...等.如果要設定這些格式的顯示方式就會方便許多

日期時間



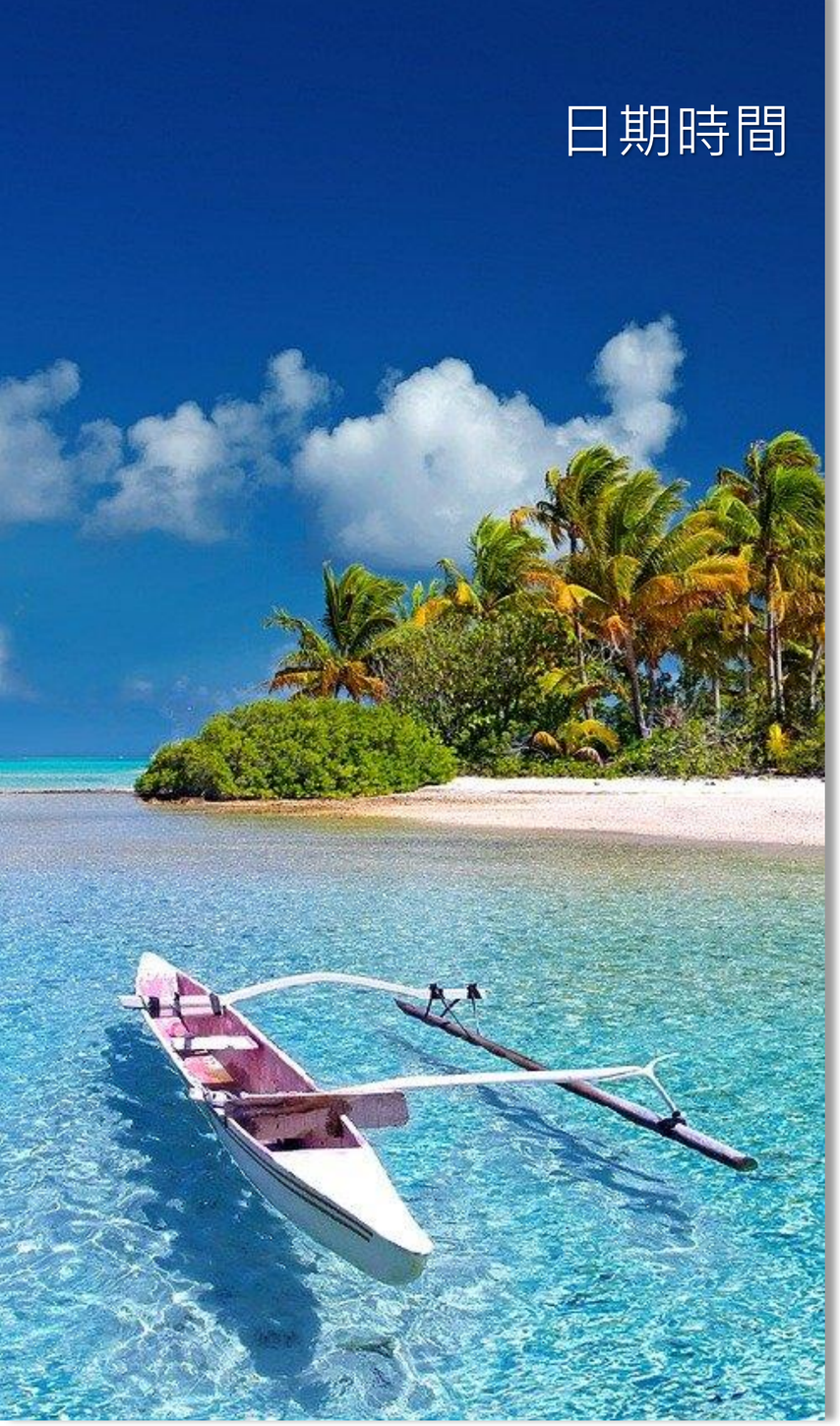
- Date類(JDK 7)
- date=new Date(2020,2,15);
- System.out.println(date);
- 同字串皆為immutable
- LocalDate(JDK 8)
  - 儲存年，月，日的資訊，且未包含時間
  - 呼叫toString()方法可回傳yyyy-MM-dd（ISO8601格式)
- LocalTime
  - 儲存時，分，秒和次於秒的更小單位的資訊，且未包含日期
  - 呼叫toString()方法可回傳HH:mm:ss.s

日期時間



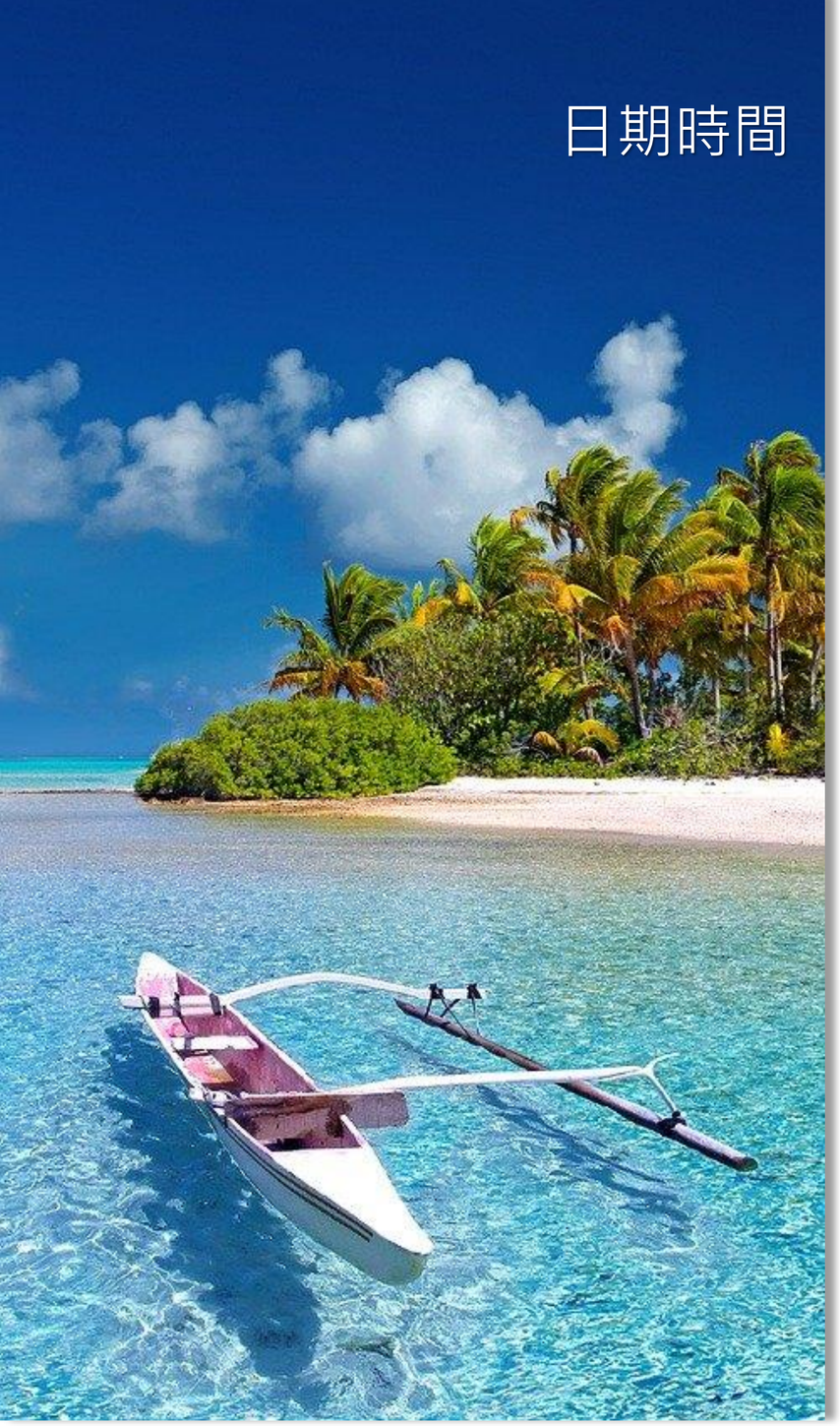
- LocalDate date=new LocalDate()//此行錯誤,因為沒有建構子
- LocalDate now = LocalDate.now();
- LocalTime time = LocalTime.now();
- LocalDateTime dateTime = LocalDateTime.now();
- System.out.println(now);
- System.out.println(time);
- System.out.println(dateTime);

日期時間

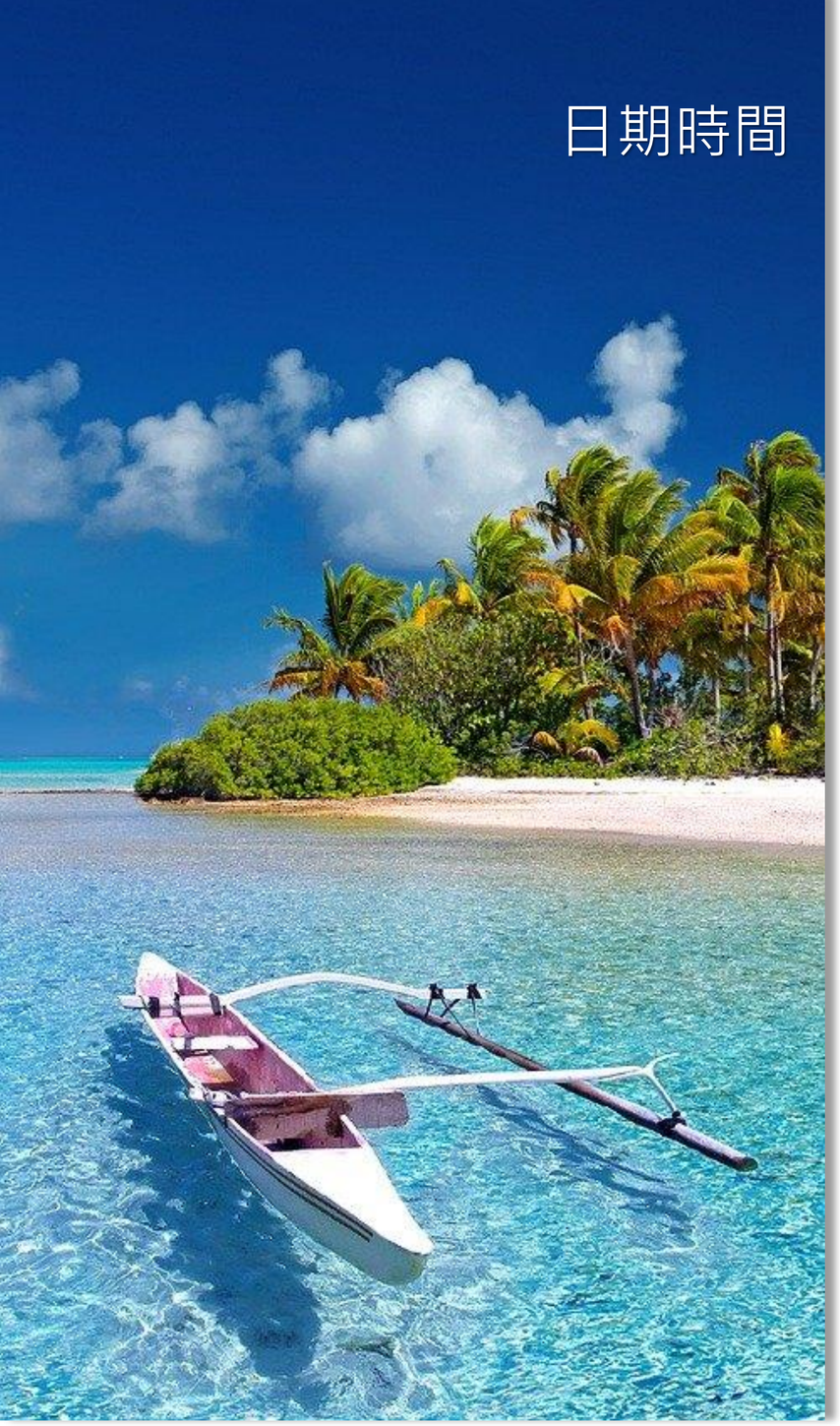


```
LocalDateTime of = LocalDateTime.of(2022,5,12,20,15,13);
        System.out.println(of);
        //取得
        System.out.println(of.getYear());
        System.out.println(of.getDayOfMonth());
        System.out.println(of.getMonthValue());
        System.out.println(of.getDayOfWeek());
        System.out.println(of.getHour());
        System.out.println(of.getMinute());
        //更改 Localxxx為immutable
        System.out.println("=======更改========");
        LocalDateTime of1 = of.withDayOfMonth(8);
        System.out.println(of);
        System.out.println(of1);
        //加減
        System.out.println("=======加減========");
        LocalDateTime of2 = of.plus(Period.ofDays(10));
        LocalDateTime of3 = of.plusDays(10);
        LocalDateTime of4 = of.plus(Period.ofMonths(2));
        LocalDateTime of5 = of.minus(Period.ofMonths(2));
        System.out.println(of2);
        System.out.println(of3);
        System.out.println(of4);
        System.out.println(of5);
        //格林威治時間
        System.out.println("====Instant======");
        Instant now2 = Instant.now();
        System.out.println(now2);
        OffsetDateTime now3 = now2.atOffset(ZoneOffset.ofHours(8));
        System.out.println(now3);
```

日期時間



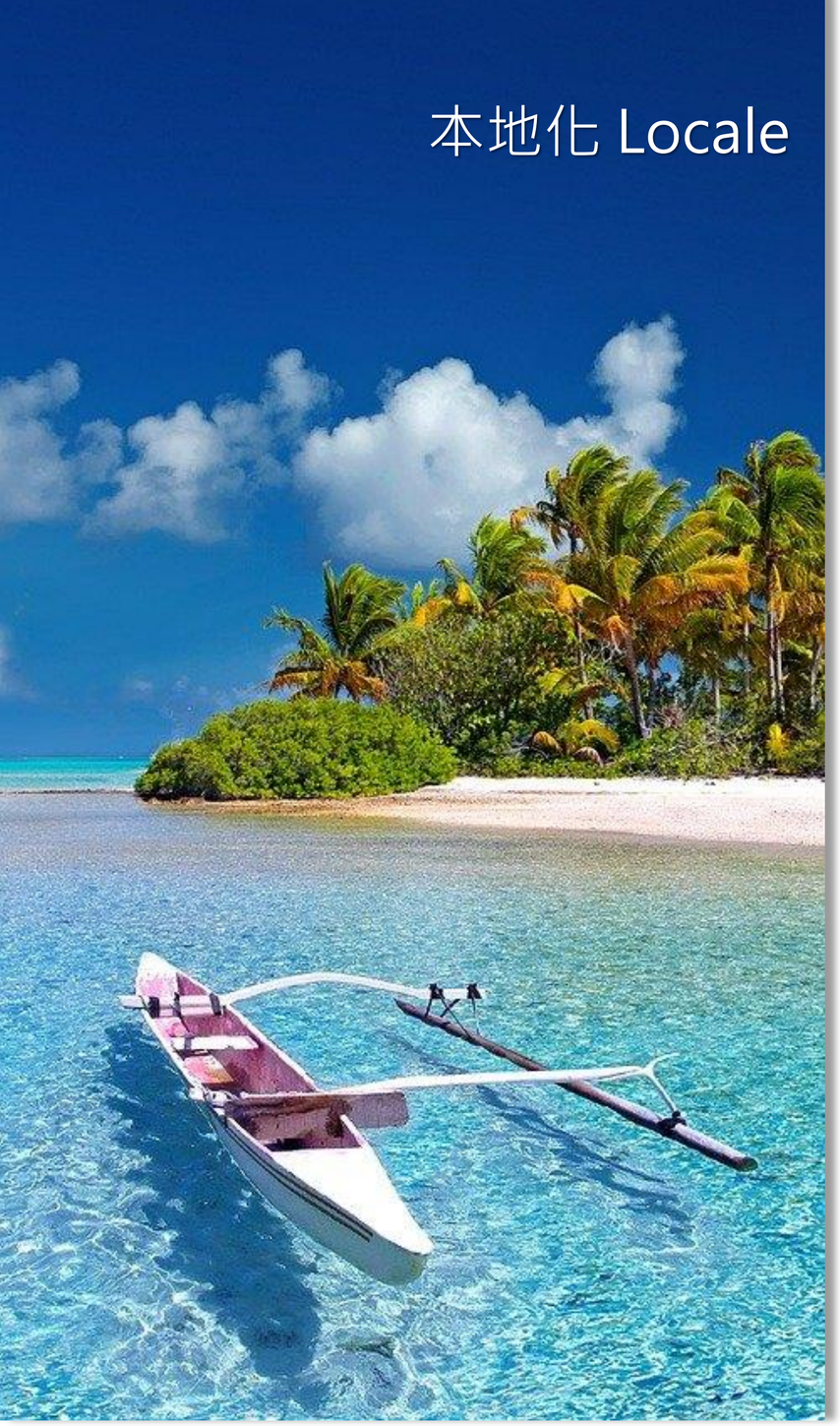
- 格式與解析
- //DateTimeFormatter格式化日期 時間與解析
- LocalDateTime now = LocalDateTime.now();
- DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
- //日期-->字串
- String str1 = formatter.format(now);
- System.out.println(now);
- System.out.println(str1);
- //解析 字串-->日期(時間存取器)
- TemporalAccessor parse = formatter.parse(str1);
- System.out.println(parse);
- //本地化相關的格式
- System.out.println("=====本地化相關的格式=====");
- //FormatStyle.FULL LONG MEDIUM SHORT
- DateTimeFormatter ofLocalizedDate = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);
- String format2 = ofLocalizedDate.format(now);
- //自定義格式 ofPattern("自訂格式")
- DateTimeFormatter ofPattern = DateTimeFormatter.ofPattern("yyyy->mm->dd hh->mm->ss");
- String format = ofPattern.format(now);
- System.out.println(format);



日期時間

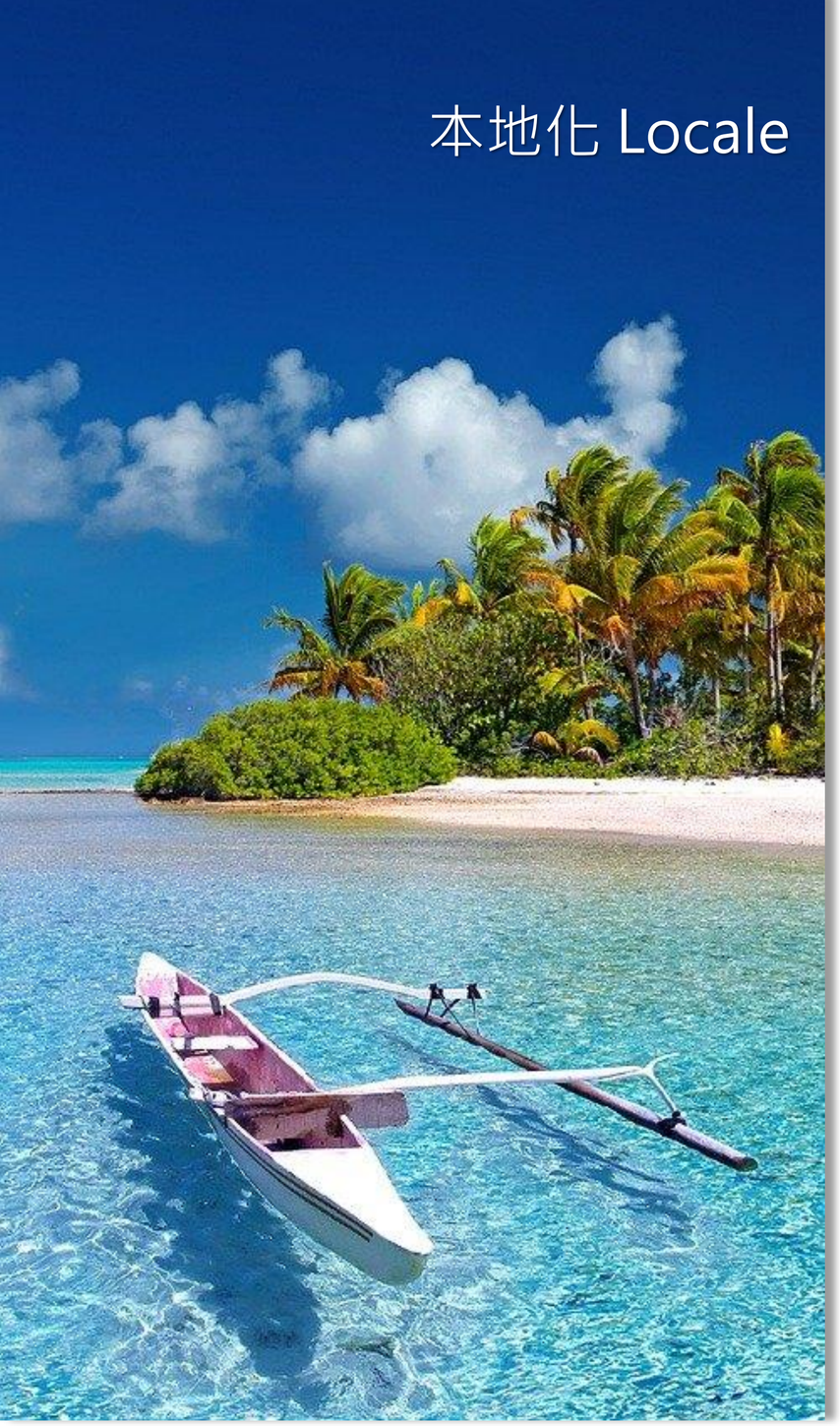
- LocalDate now=LocalDate.now();
- System.out.println("Now:"+now);
- LocalDate birthDay=LocalDate.of(1980,8,5);
- System.out.println(birthDay);
- System.out.println("是否為過去"+birthDay.isBefore(now));
- System.out.println("是否為閏年"+birthDay.isLeapYear());
- System.out.println("為星期幾"+birthDay.getDayOfWeek());
- System.out.println("為幾日"+birthDay.getDayOfMonth());
- System.out.println(“為一年中第幾天 "+birthDay.getDayOfYear());
- LocalDate nowAfterMonth=now.plusMonths(1);
- System.out.println(nowAfterMonth);

本地化 Locale



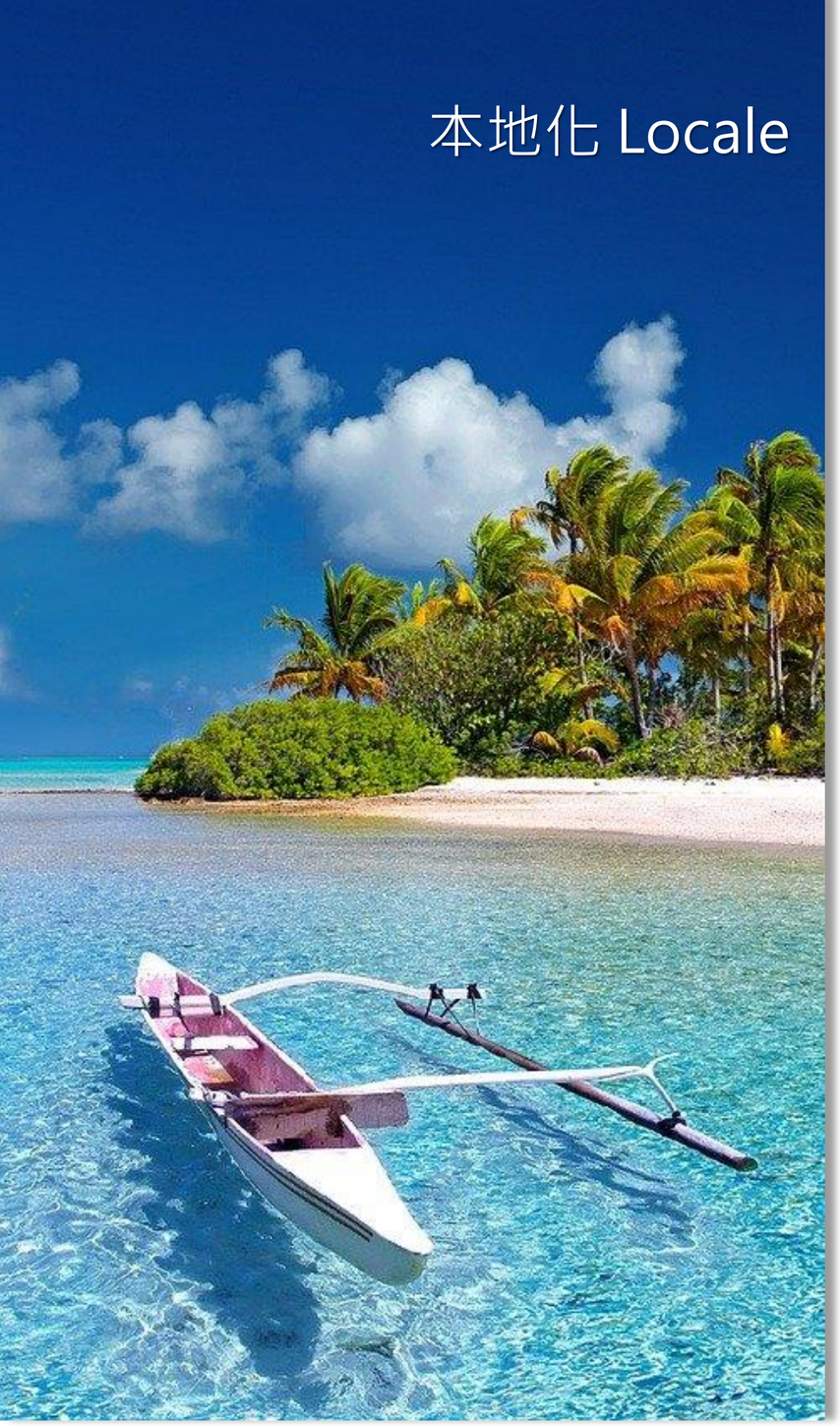
- 由於國家不同,連帶地資料的顯示習慣也不同, java.util.Locale 物件表示了特定的地理 政治 和文化地區.需要 Locale 來執行其任務的操作並且可為使用者量身定制的資訊.例如 顯示一個日期或數值應根據使用者的國家 地區 或傳統習慣來格式化該資訊

本地化 Locale



```
// 建立Locale物件方式
// 1.使用預設
Locale loc = Locale.getDefault();
System.out.println(loc);
Locale locale1 = new Locale("zh", "TW", "WIN");
System.out.println("Locale:" + locale1);
//setDefault(Locale newLocale)
//setDefault(Locale.Category category, Locale newLocale)
Locale.setDefault(new Locale("en", "US", "MAC"));
Locale locale2 = Locale.getDefault();
System.out.println("Locale::" + locale2);
// 2.使用靜態物件
System.out.println("=====使用靜態物件=======");
Locale loc1 = Locale.SIMPLIFIED_CHINESE;
System.out.println(loc1);
// 3.使用建構子
/*
* 國別一定要配合語系 Locale(String language) Locale(String language, String country)
* Locale(String language, String country, String variant)
*/
Locale loc2 = new Locale("zh", "TW");
System.out.println(loc2);
// Local支援的區域
System.out.println("=====Local支援的區域=======");
Locale[] availableLocales = Locale.getAvailableLocales();
for (Locale x : availableLocales)
    System.out.print(x + ",");
```

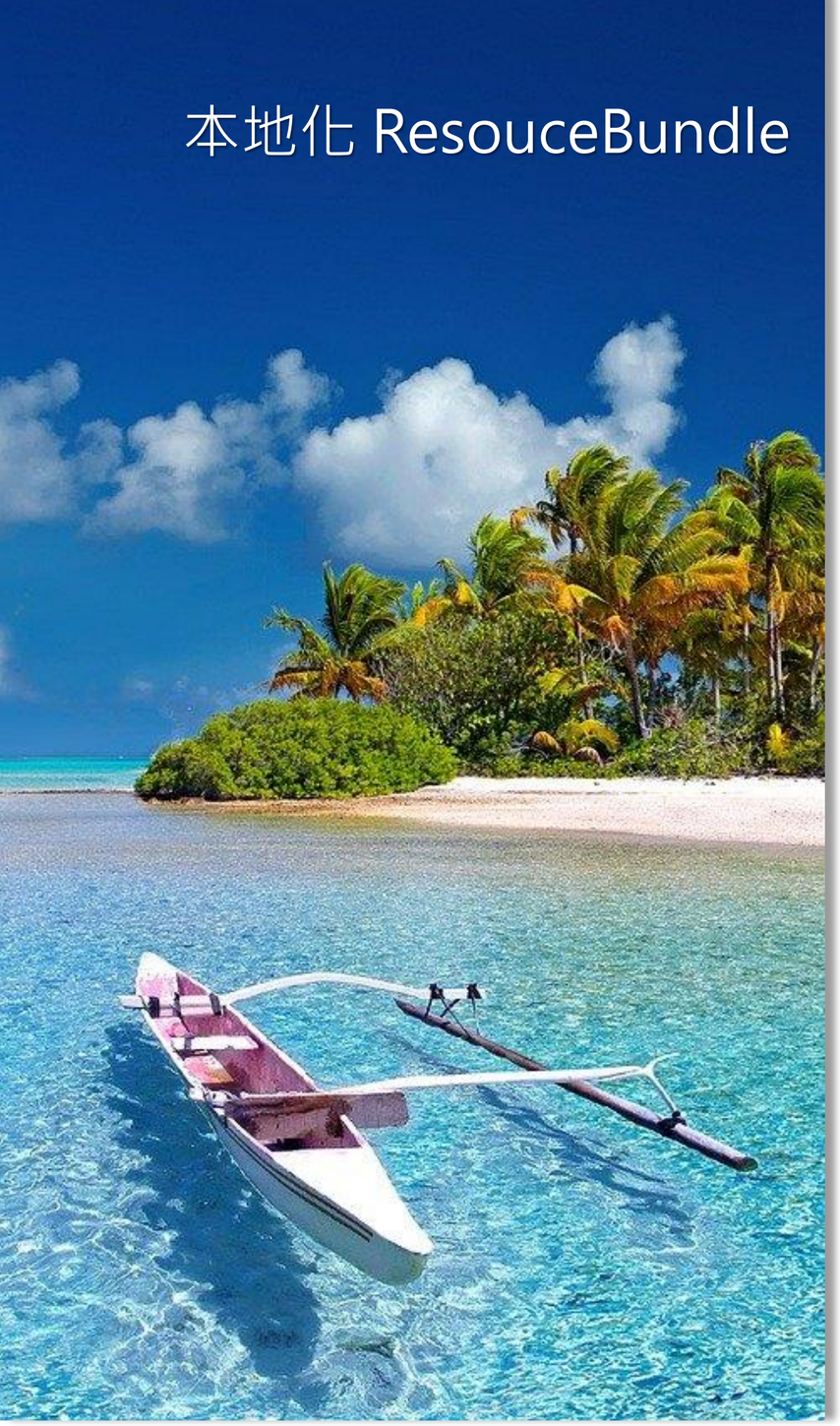
本地化 Locale



Which two statements set the default locale used for formatting numbers, currency, and percentages? (Choose two.)

A. Locale.setDefault(Locale.Category.FORMAT, "zh-CN");
B. Locale.setDefault(Locale.Category.FORMAT, Locale.CANADA_FRENCH);
C. Locale.setDefault(Locale.SIMPLIFIED_CHINESE);
D. Locale.setDefault("en_CA");
E. Locale.setDefault("es", Locale.US);

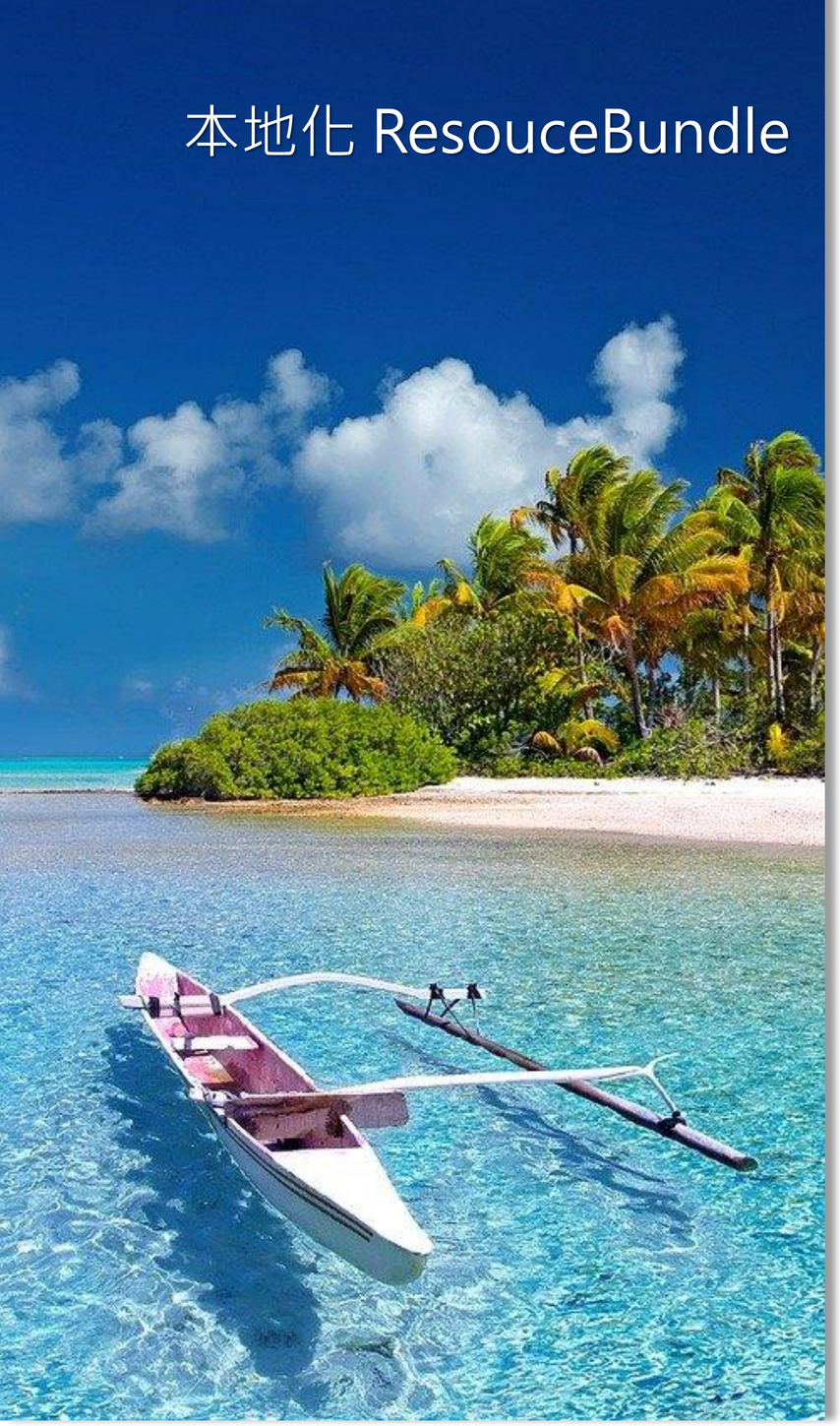
本地化 ResouceBundle



- Given the contents:
- MessageBundle.properties file:
- message=Hello
- MessageBundle_en.properties file:
- message=Hello (en)
- MessageBundle_US.properties file:
- message=Hello (US)
- MessageBundle_en_US.properties file:
- message=Hello (en_US)
- MessageBundle_fr_FR.properties file:
- message=Bonjour

```
Locale.setDefault(Locale.FRANCE);
Locale default1 = Locale.getDefault();
System.out.println(default1);
//Locale currentLocale = new Locale.Builder().setLanguage("en").setRegion("US").build();
//Locale currentLocale = new Locale.Builder.setRegion("US").build();
Locale currentLocale = new Locale.Builder().setLanguage("en").build();
System.out.println(currentLocale);
//Language為資源檔名稱
ResourceBundle messages = ResourceBundle.getBundle("Language", currentLocale);
System.out. println(messages.getString("message"));
```

本地化 ResouceBundle



```
MessagesBundle.properties file:
username = Username
password = Password
```

and

```
MessagesBundle_fr_FR.properties file:
username = Utilisateur
password = Le passe
```

and

```
MessagesBundle_ru.properties file:
username = Пользователь
password = Пароль
```

```
Locale.setDefault(Locale.FRANCE);
ResourceBundle msg=ResourceBundle.getBundle("MessagesBundle",new Locale("ru"));
System.out.println("user:"+msg.getString("username"));
System.out.println("password:"+msg.getString("password"));
```

What is the result?

A.
```
username = Пользователь
password = Пароль
```

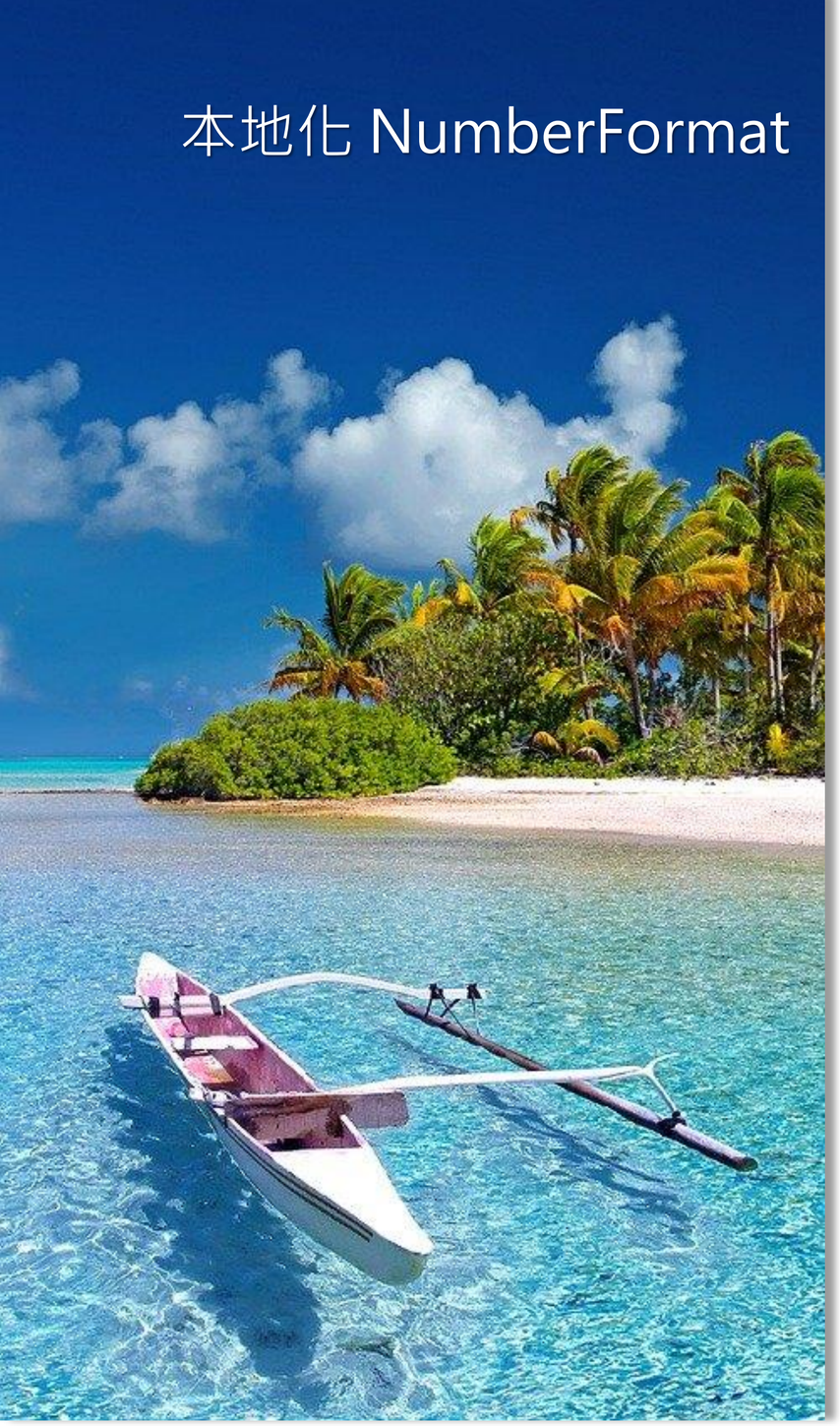
B. The compilation fails

C. MissingResourceExcetion is thrown at runtime

D. User=Utilisateur

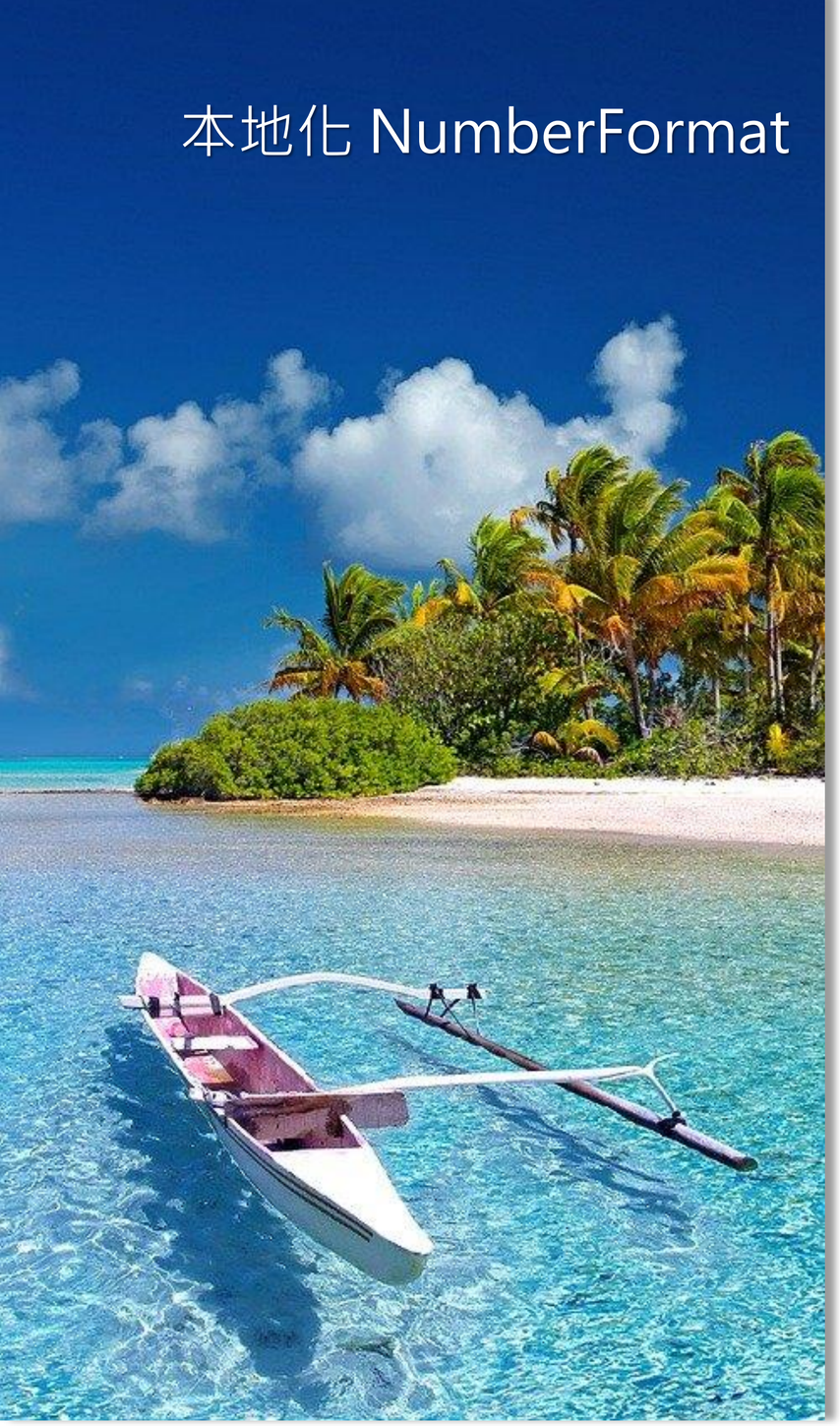
pssword=Le passes

本地化 NumberFormat



- double salary=3.15;
- Locale local=Locale.FRANCE;
- NumberFormat instance = NumberFormat.getInstance(local);
- NumberFormat currencyInstance = NumberFormat.getCurrencyInstance(local);
- String displayName = instance.format(salary);
- String displayName2 = currencyInstance.format(salary);
- System.out.println("=====數字格式========");
- System.out.println(displayName);
- System.out.println(displayName2);
- Currency currency = instance.getCurrency();
- Currency currency2 = currencyInstance.getCurrency();
- System.out.println("=====貨幣代碼========");
- System.out.println(currency);
- System.out.println(currency2);
- String displayName3 = currency.getDisplayName();
- String displayName4 = currency2.getDisplayName();
- System.out.println("=====貨幣名稱========");
- System.out.println(displayName3);
- System.out.println(displayName4);

本地化 NumberFormat



```
Locale locale=Locale.US;
double currency=1_00.00;
// line 1
System.out.println(formatter.format(currency));
```

You want to display the value of currency as $100.00

Which code inserted on line 1 will accomplish this

A. NumberFormat formatter=NumberFormat.getCurrencyInstance(locale);

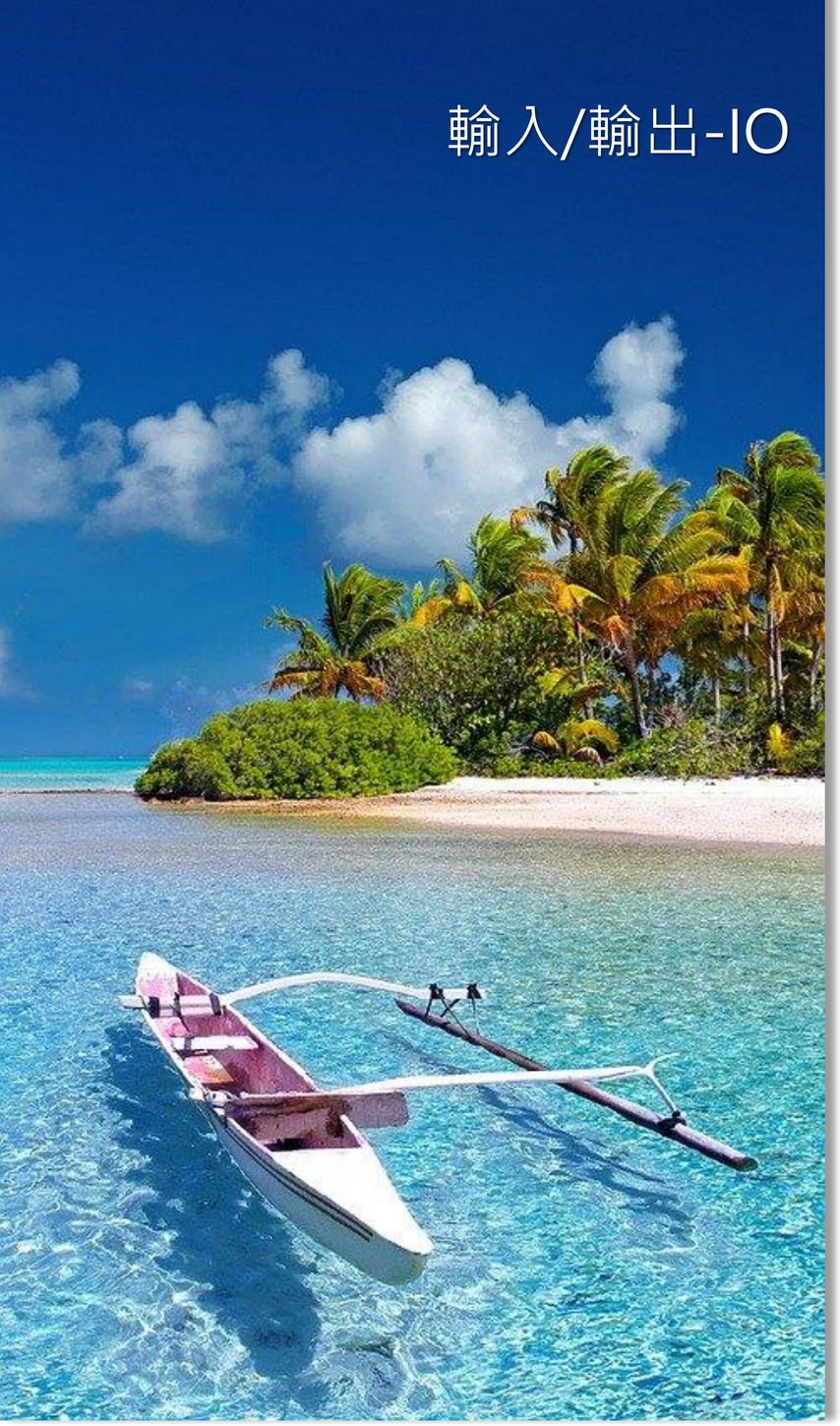
B. NumberFormat formatter=NumberFormat.getCurrency (locale);

C. NumberFormat formatter=NumberFormat.getInstance(locale).getCurrency();

D. NumberFormat formatter=NumberFormat.getInstance(locale);

輸入/輸出 I/O

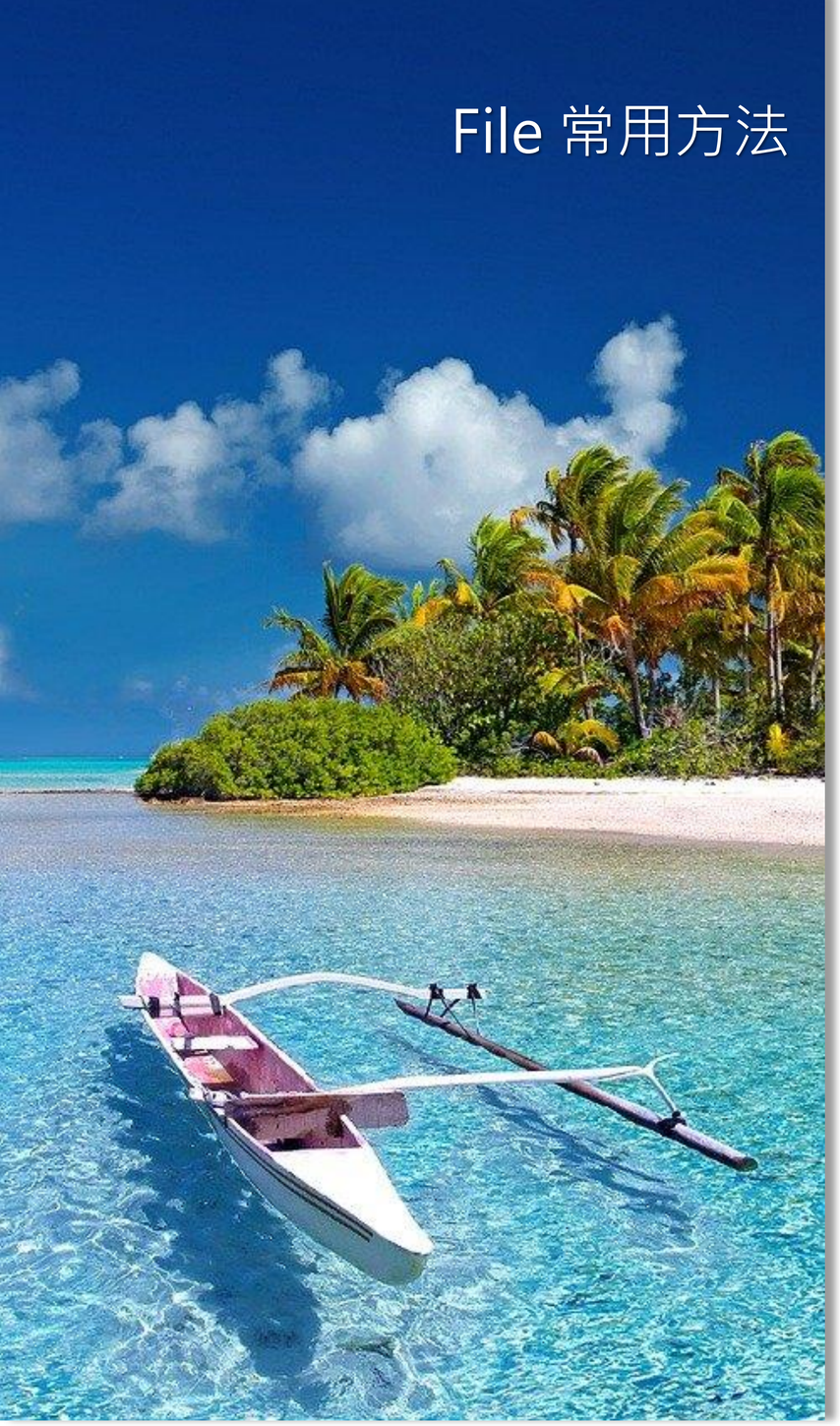
輸入/輸出-IO



一

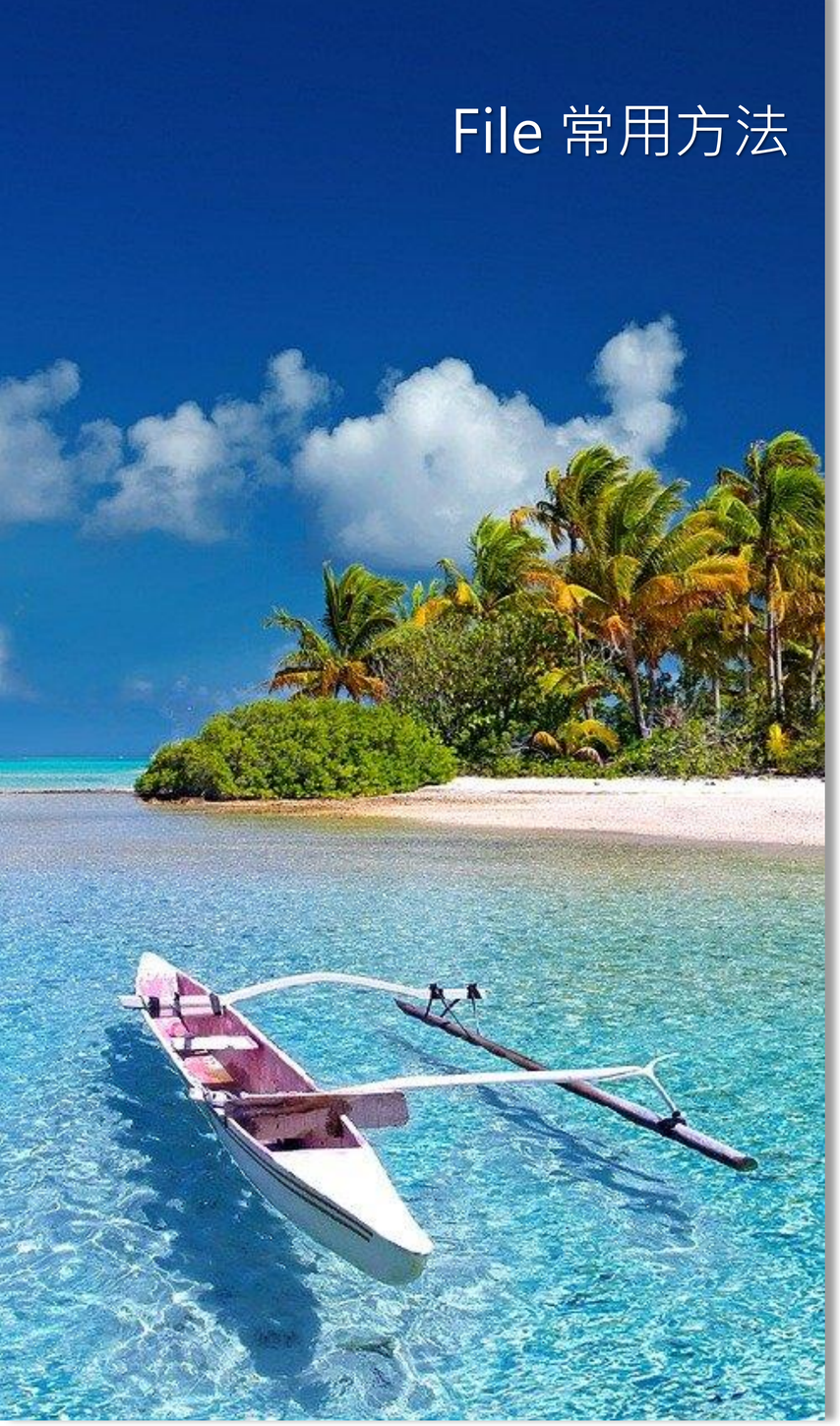
- File類別是java.io函式庫中唯一不是stream的類別，既不能讀取檔案內容，也不能改變檔案內容，主要作用是收集檔案或目錄的相關資訊，如建立檔案，修改檔名，設定檔案屬性，建立目錄，顯示目錄中一組的檔名.....等
- File類別物件則代表一個檔案或是一個資料夾
- File類別所提供的建構子
  - public File(String pathname)
    - File file=new File("abc.txt");
    - File file2=new File("d:"+File.separator+"ok"+File.separator+"abc.txt");
    - File file3=new File("d:\\ok\\abc.txt");
    - file.createNewFile();
    - file2.createNewFile();
    - file3.createNewFile();
  - public File(String parent,String child)
    - File file3=new File("d:\\ok","\\test");
  - public File(File parent,String child)//父類別為File類
    - File file4=new File(file3,"ok.txt");

File 常用方法



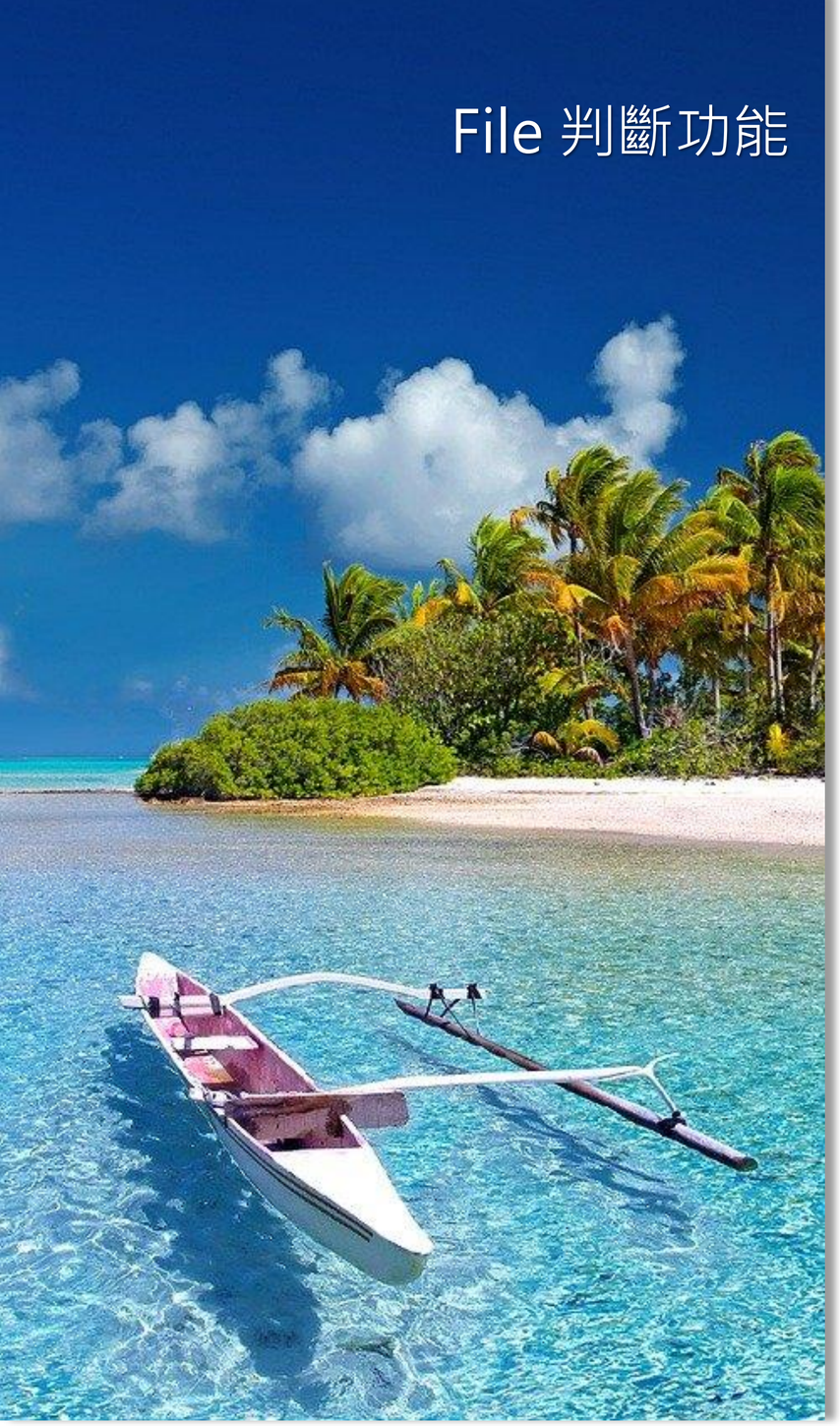
- getAbsoloutePath( )取得絕對路徑
- getPath( )取得相對路徑
- getName( )取得名稱
- getParent( )取得父層名稱
- length( )取得檔案長度
- lastModified( )取得上次修改時間
- list( )取得指定目錄下之子目錄及檔案,字串類型
- listFiles( ) ,檔案類型
- renameTo( ) 將檔案重新命名到指定的目錄,來源需存在,但目的不能存在

File 常用方法



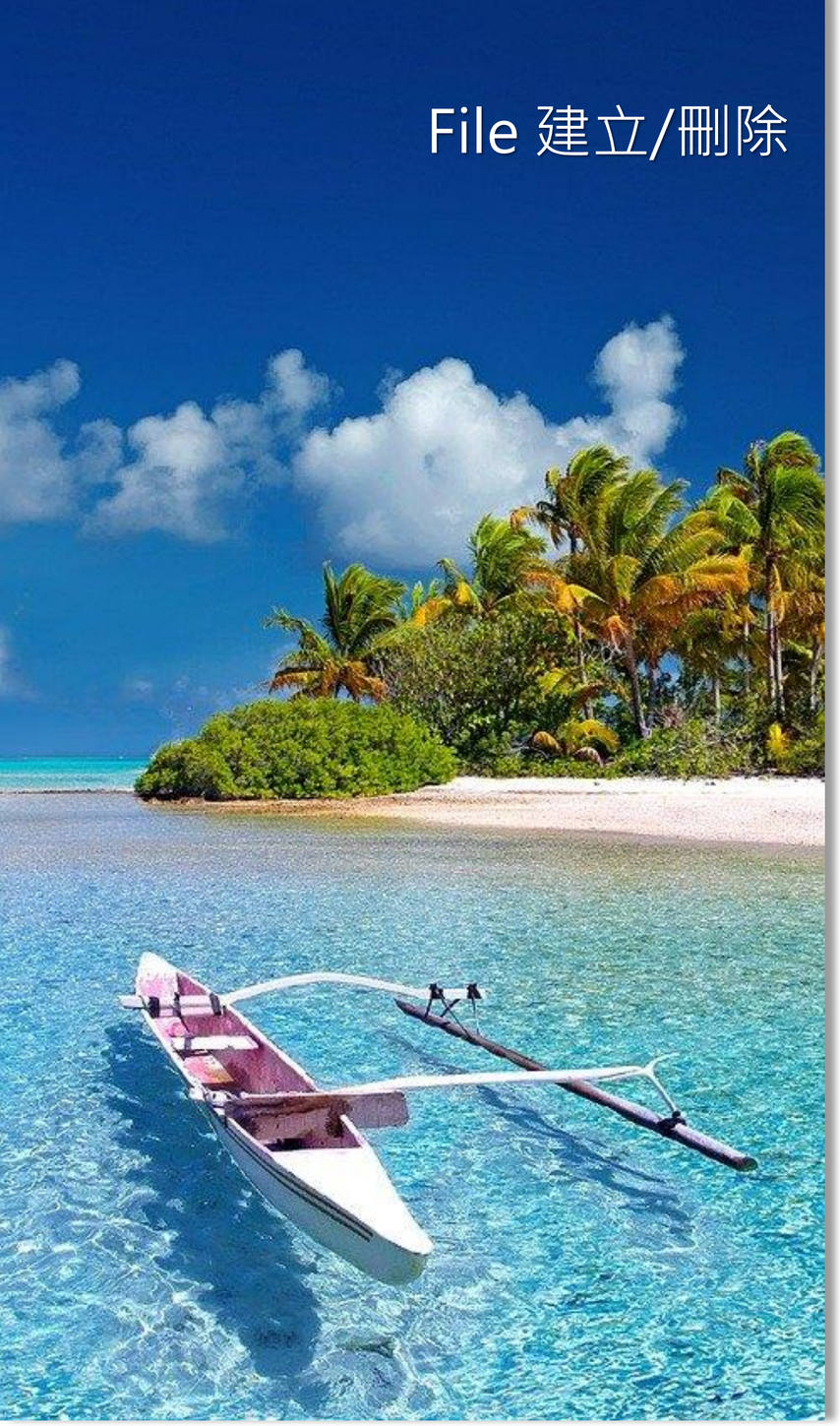
```
File f=new File("hello.txt");
    File f2=new File("d:\\ocajp\\hello2.txt");
File f3=new File("c:\\windows");
        System.out.println(f.getAbsoluteFile());
        System.out.println(f.getPath());
        System.out.println(f.getName());
        System.out.println(f.getParent());
        System.out.println(f.length());
        System.out.println(new Date(f.lastModified()));
        System.out.println("=========絕對路徑==========");
          System.out.println(f2.getAbsoluteFile());
        System.out.println(f2.getPath());
        System.out.println(f2.getName());
        System.out.println(f2.getParent());
        System.out.println(f2.length());
        System.out.println(f2.lastModified());
System.out.println("==================");
      String[] d=f3.list();
      for(String x:d)
          System.out.println(x);
        System.out.println("====================");
        File[] files=f3.listFiles();
        for(File x:files)
          System.out.println(x);
//f需存在,但f4不能存在
File f4=new File("c:\\ocpjp\\ok.txt");
        boolean b=f.renameTo(f4);
        System.out.println(b);
```

File 判斷功能



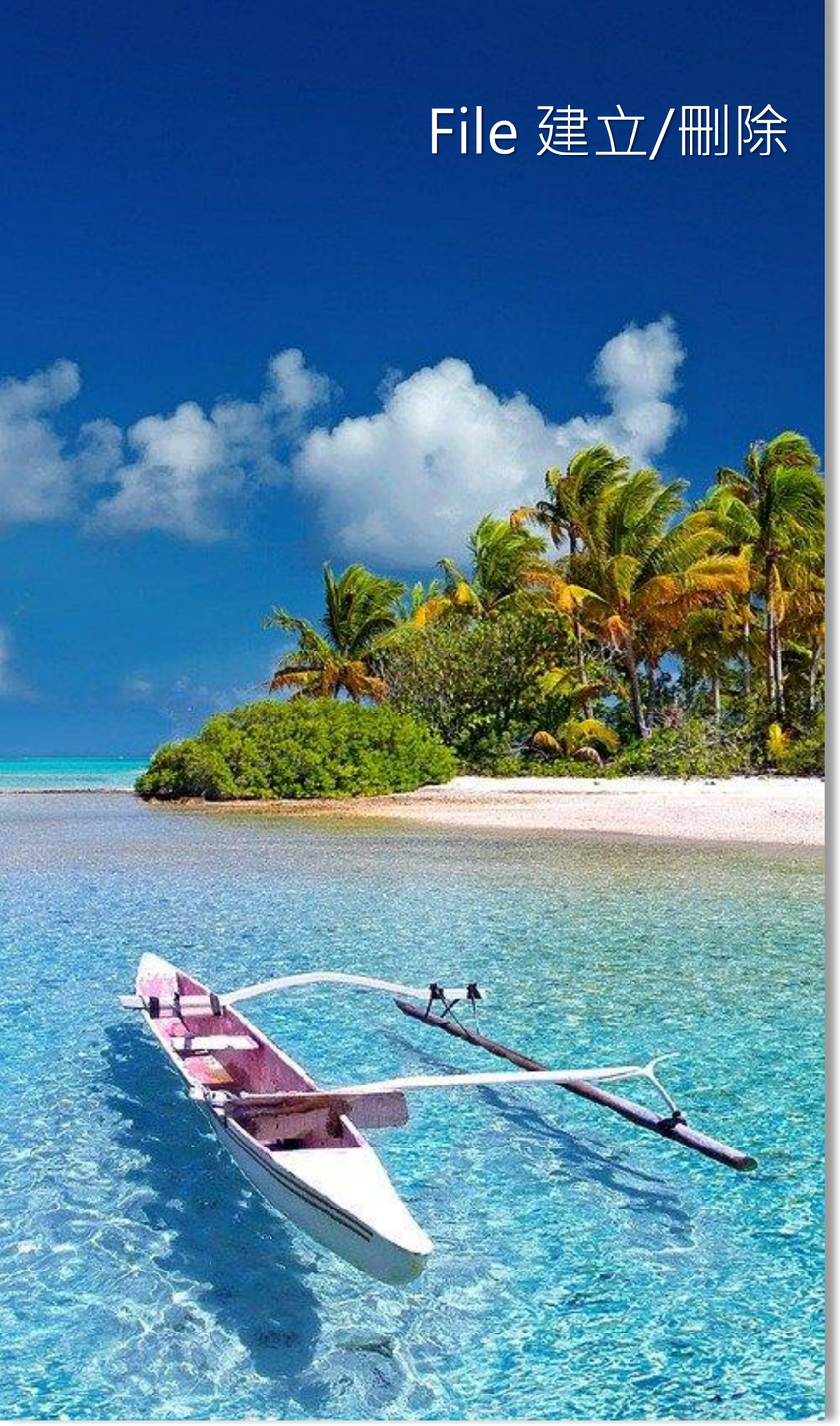
- isDirectory( ) 判斷是否為資料夾
- isFile( ) 判斷是否為檔案
- exists( ) 判斷檔案是否存在
- canRead( ) 判斷是否可讀取
- canWrite( ) 判斷是否為可寫入
- isHidden( ) 判斷是否為隱藏
- //範例
- File f = new File("abc.txt");
- System.out.println(f.isDirectory());
- System.out.println(f.isFile());
- System.out.println(f.exists());
- System.out.println(f.canRead());
- System.out.println(f.canWrite());
- System.out.println(f.isHidden());

File 建立/刪除



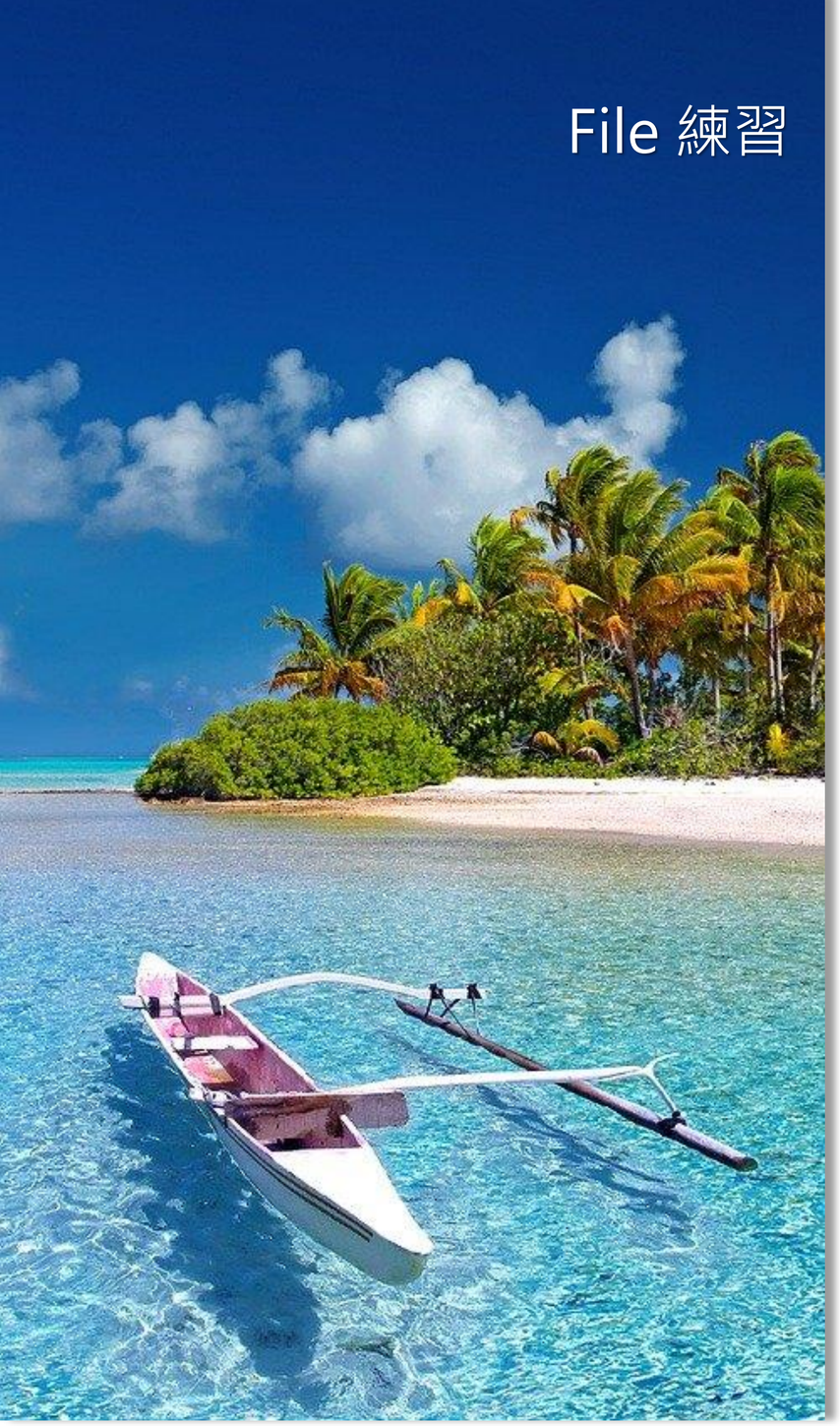
- createNewFile( ) 建立檔案,若檔案已存在是不再建立,返回false
- mkdir( ) 建立目錄,若上層目錄不存在,或是目錄已存在皆不會建立
- mkdirs( )建立目錄,若上層目錄不存在,也會一併建立
- delete( ) 刪除檔案或目錄,若是目錄內還有子目錄則無法刪除

File 建立/刪除



```
File f = new File("abc.txt");
      if(!f.exists())
      {
         f.createNewFile();
         System.out.println("檔案建立成功");
      }
      else
      {
         f.delete();
         System.out.println("檔案刪除成功");
      }

//建立資料夾
File f = new File("c:\\0703\\TestFile\\hello");
      if(!f.exists()){
         f.mkdir();//嚐試改成mkdirs
         System.out.println("資料夾建立成功");
      }
      else
      {
         f.delete();
         System.out.println("資料夾刪除成");
      }
```

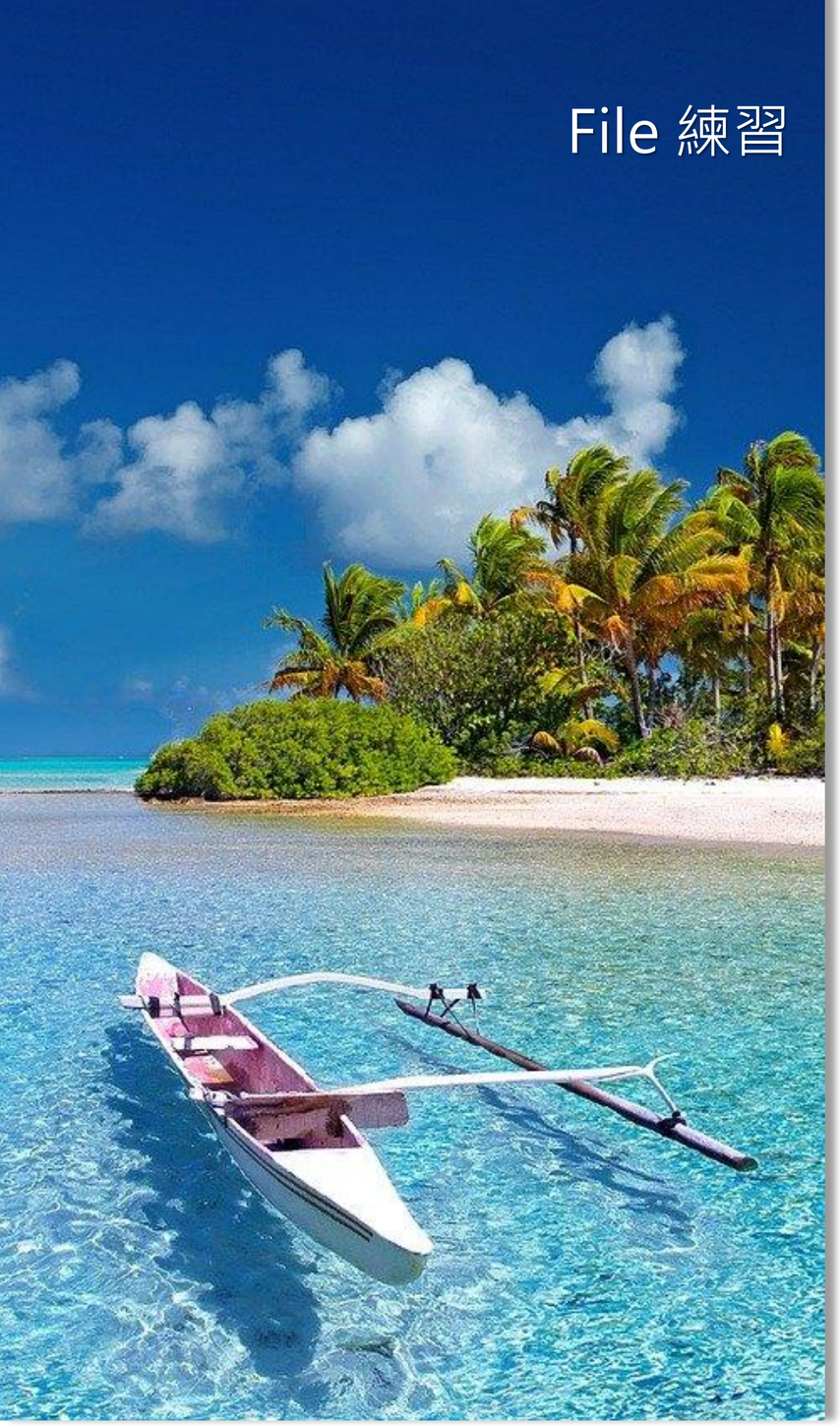
File 練習



- 判斷指定目錄下是否有副檔名為.jpg的檔案,如果有,就輸出該檔案

```
File f=new File("E:\\photoshop上課素材");
    String file[]=f.list();
    for(String x:file)
    {
        if(x.endsWith(".jpg"))
            System.out.println(x);
    }
```

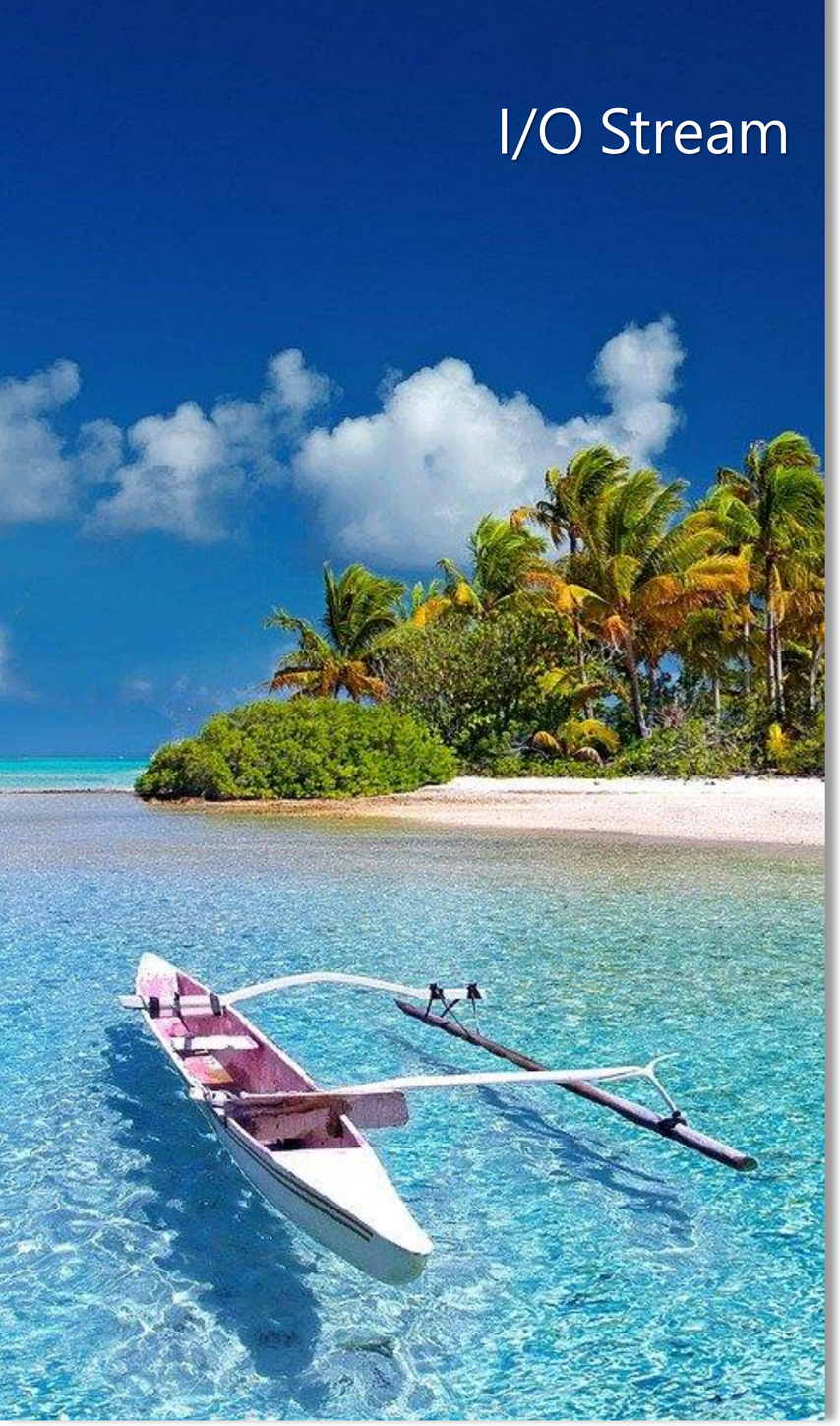
File 練習



- 撰寫方法,印出指定目錄下的所有子目錄及檔案

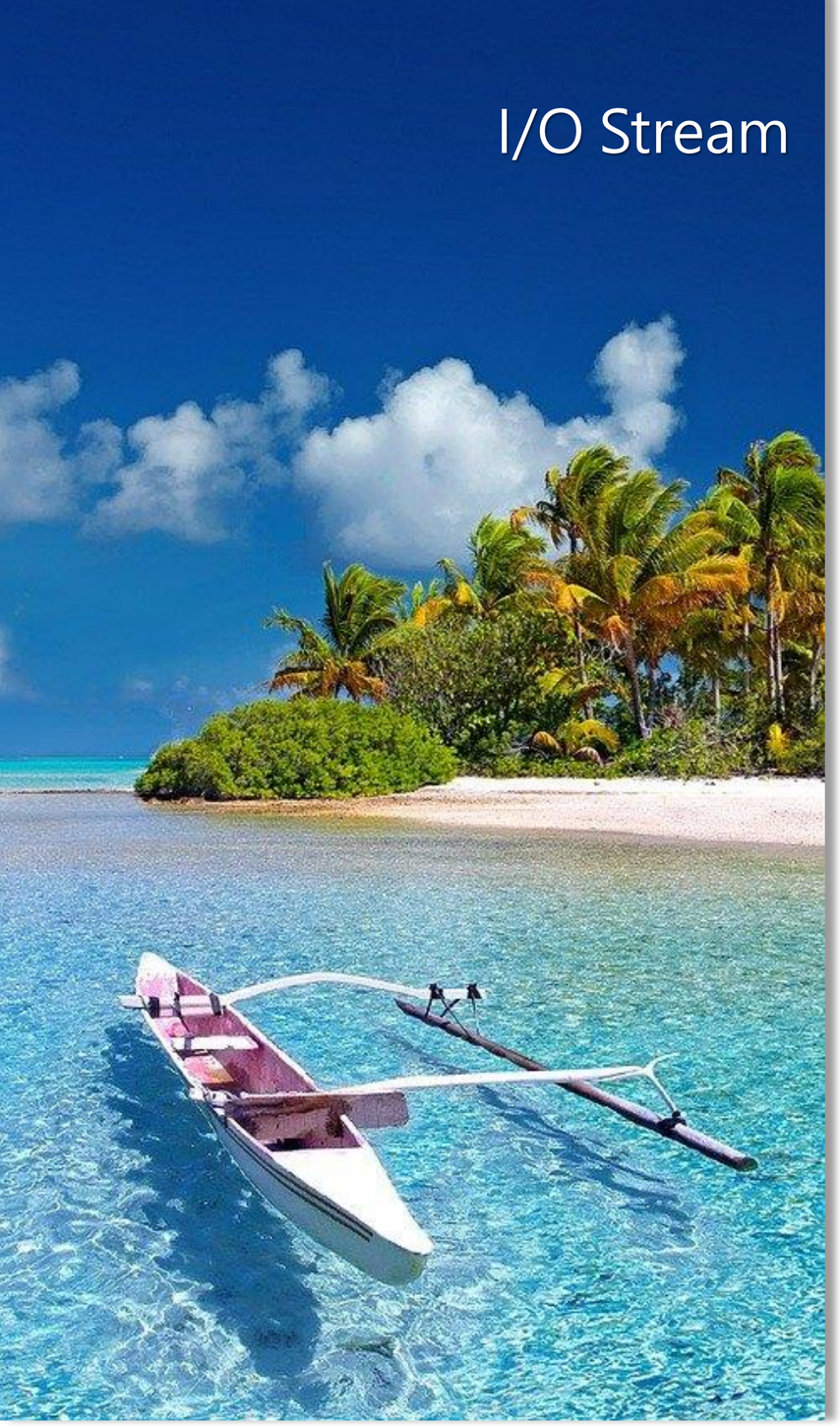
```
public static void listAllDF(String fileDir){
        List<File>fileList=new ArrayList();
        File file=new File(fileDir);
        File[] files=file.listFiles();
        if(files==null)
        {
            return;
        }
        for(File f:files)
            if(f.isFile())
                fileList.add(f);
            else if(f.isDirectory())
            {
                System.out.println(f.getAbsolutePath());
                listAllDF(f.getAbsolutePath());
            }
      for(File f1:fileList)
            System.out.println(f1.getName());

    }
```

I/O Stream



- 資料的進出像是水流/串流
  - 由來源流向目的 ，具有方向性，在JAVA中稱為stream(串流)
  - 流動的單位主要分兩類：byte，character
    - byte:圖片,視訊,音訊
- 若方向是流入java程式，則稱為input或source串流
- 若方向是流出java程式，則稱為output或sink

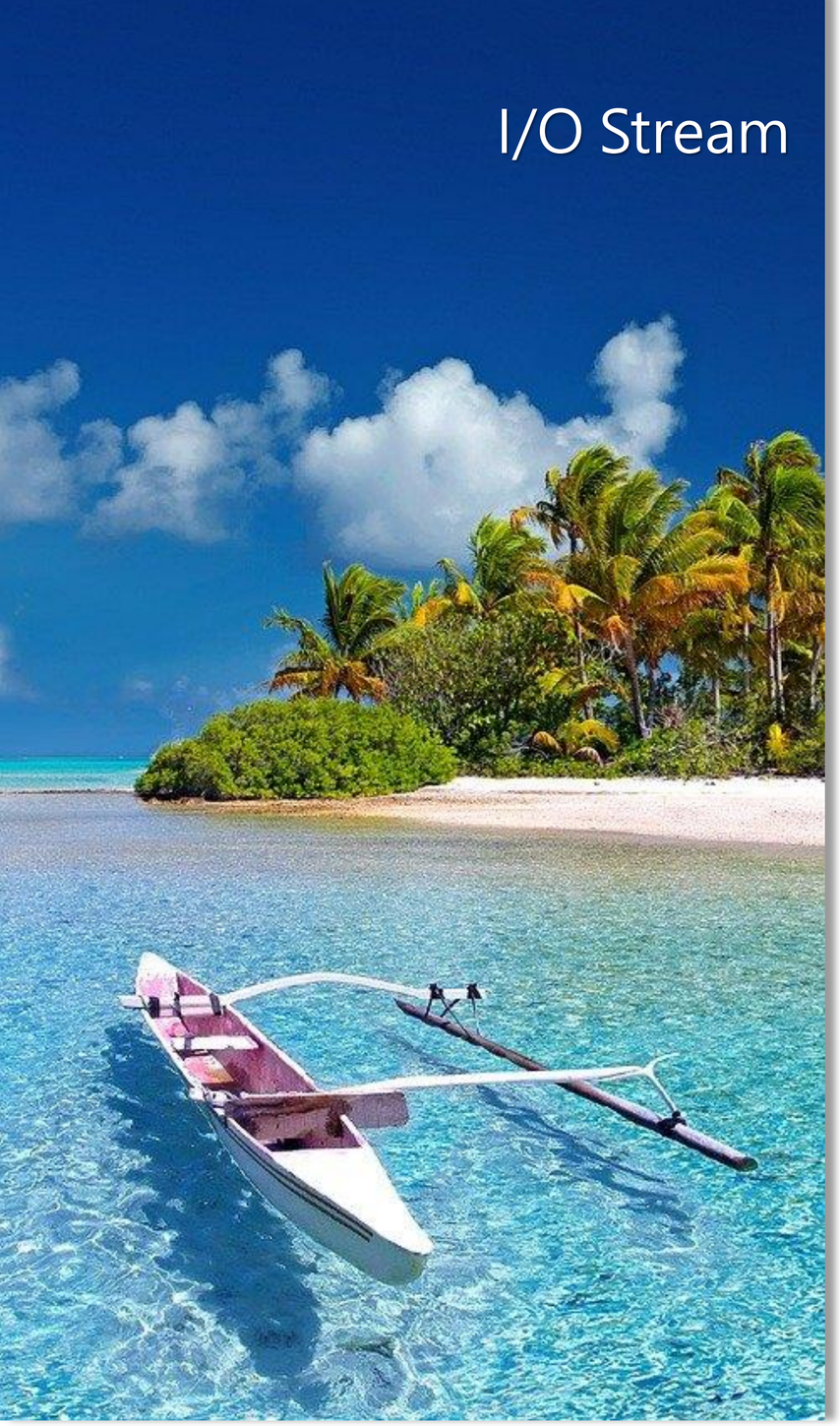
I/O Stream



一

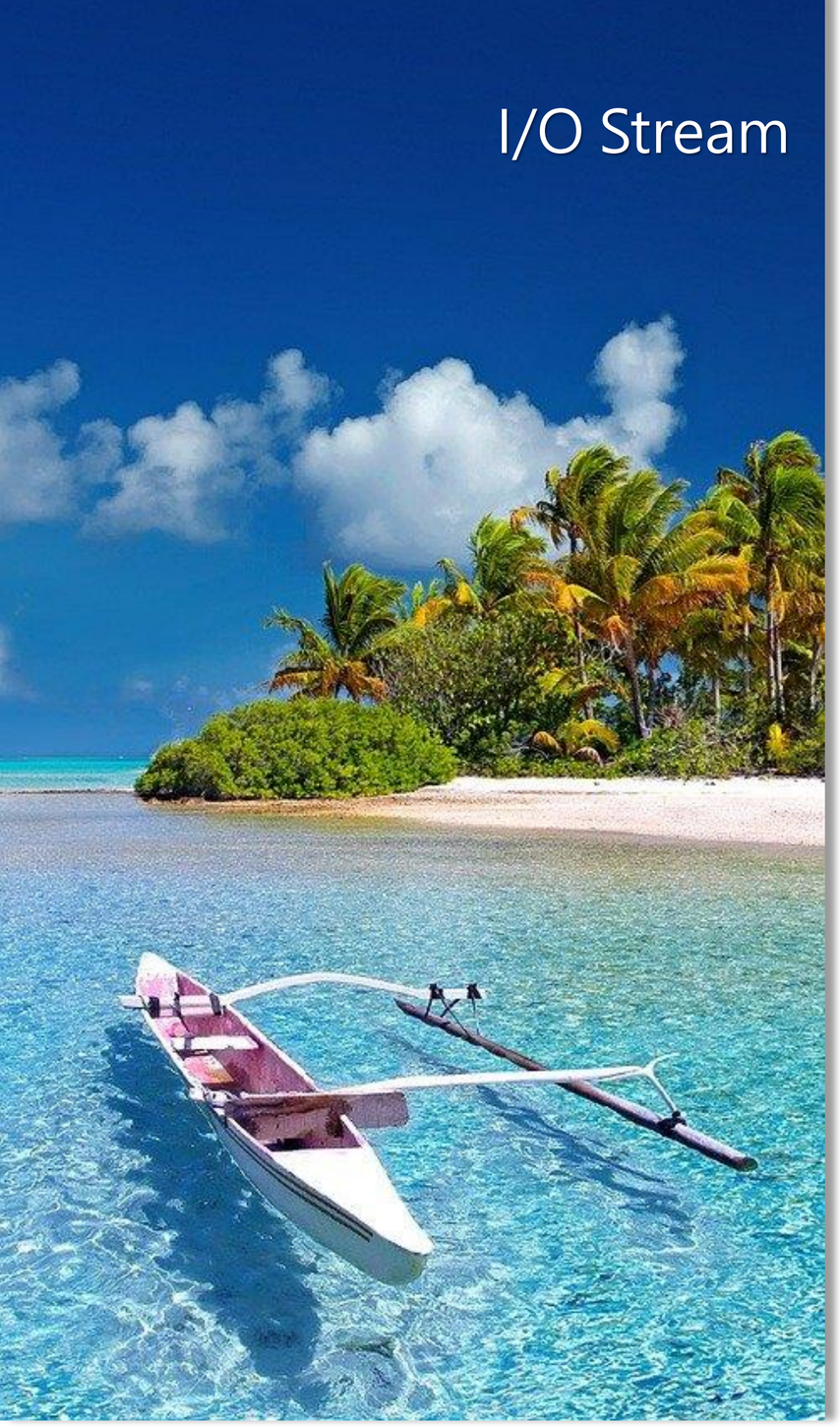
- 根據串流內的資料分類，加上流動方向，處理類別可以區分為四大類，這些都是抽象類別

方向	位元 (byte)	字元 (character)
輸入 Java	InputStream	Reader
輸出 Java	OutputStream	Writer



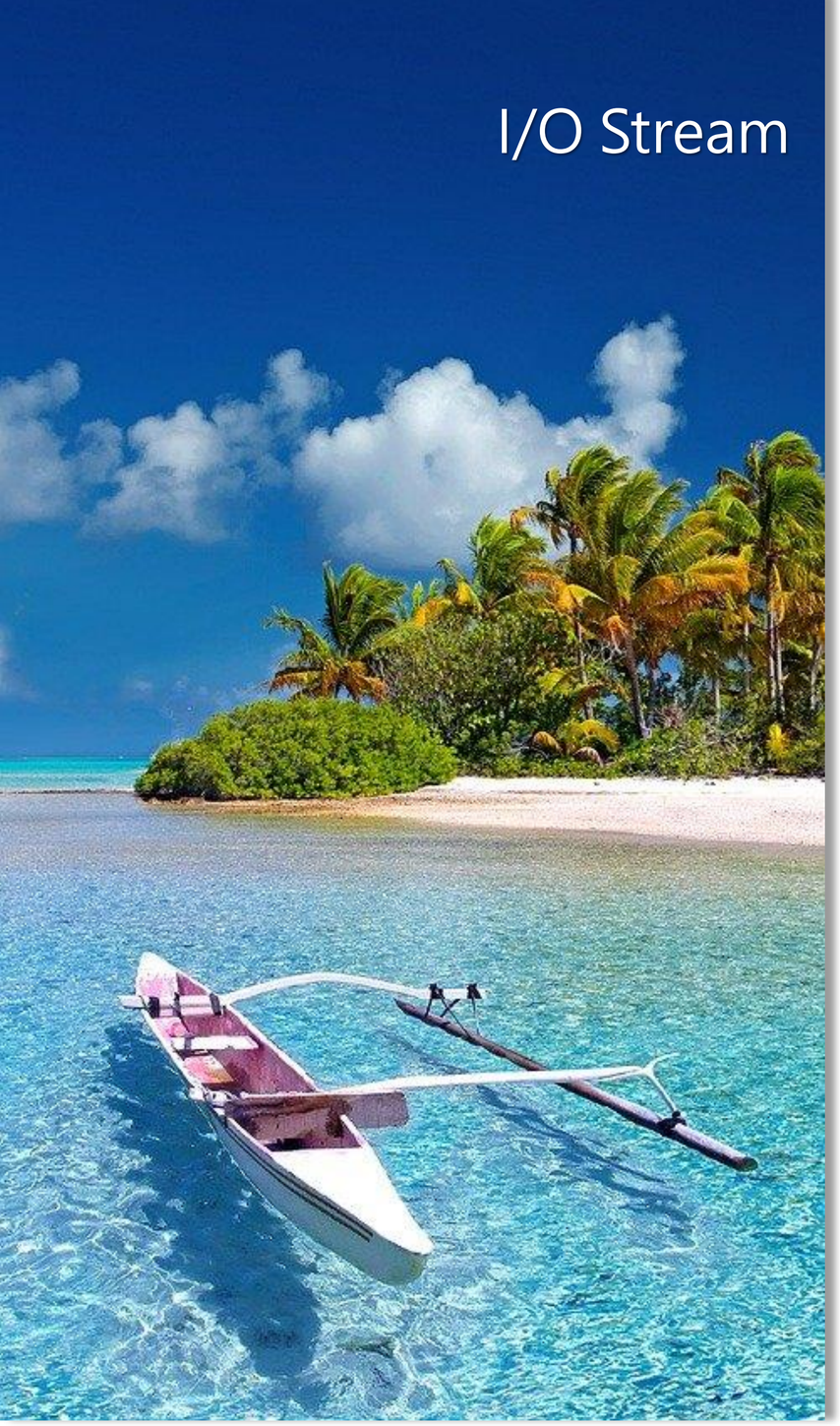
IO流架構

分類	位元輸入流	位元輸出流	字元輸入流	字元輸出流
抽象類別	InputStream	OutputStream	Reader	Writer
存取文件	FileInputStream	FileOutputStream	FileReader	FileWriter
存取陣列	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
存取管道	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
存取字串			StringReader	StringWriter
緩衝流	BufferedInputStream	BufferedOuputStream	BufferedReader	BufferedWriter
轉換流			InputStreamReader	OutputStreamWriter
物件流	ObjectInputStream	ObjectOutputStream		
	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
列印流		PrintStream		PrintWriter
推回輸入流	PushbackInputStream		PushbackReader	
特殊流	DataInpu	DataOutpu		



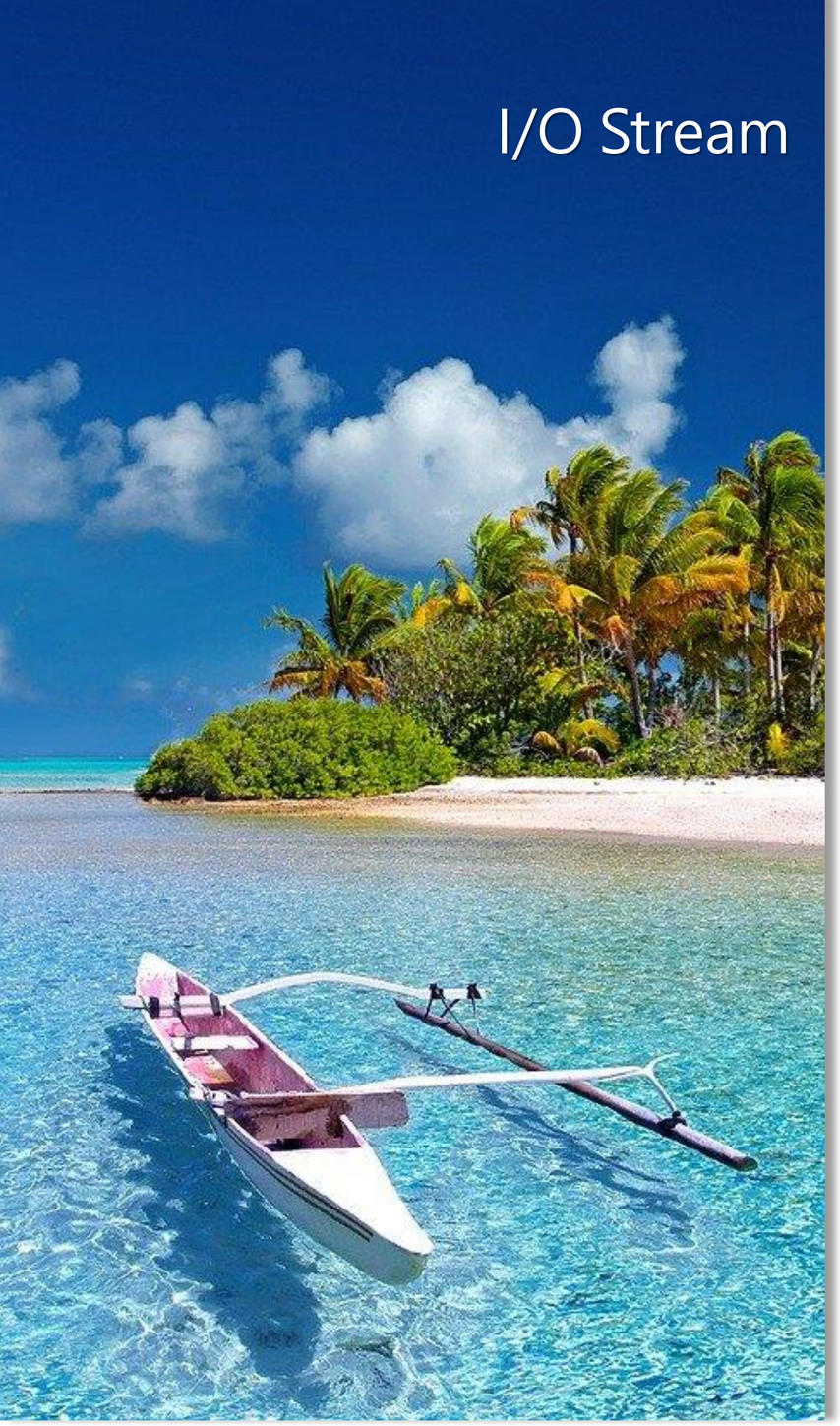
字元流

基本方法	int read() 功能：每次讀取 1 個 char。
	int read(char[] buffer) 功能：每次讀取 1 個 char[]。
	int read(char[] buffer, int offset, int length) 功能：每次讀取 1 個 char[]，且指定偏移量 (offset) 和讀取長度 (length)。
其他方法	void close() 功能：關閉 stream。
	boolean ready() 功能：確認 stream 是否已經準備好進行資料讀取。
	long skip(long n) 功能：讀取時略過 n 個 chars。
基本方法	void write(int c) 功能：將 int 寫入 Writer。
	void write(char[] buffer) 功能：將整個 char[] 寫入 Writer。
	void write(char[] buffer, int offset, int length) 功能：寫入 char[] 到 Writer，且指定長度 (length) 和偏移量 (offset)。
其他方法	void close() 功能：關閉 stream。
	void flush() 功能：強制將 Writer 中的資料寫入目的地。



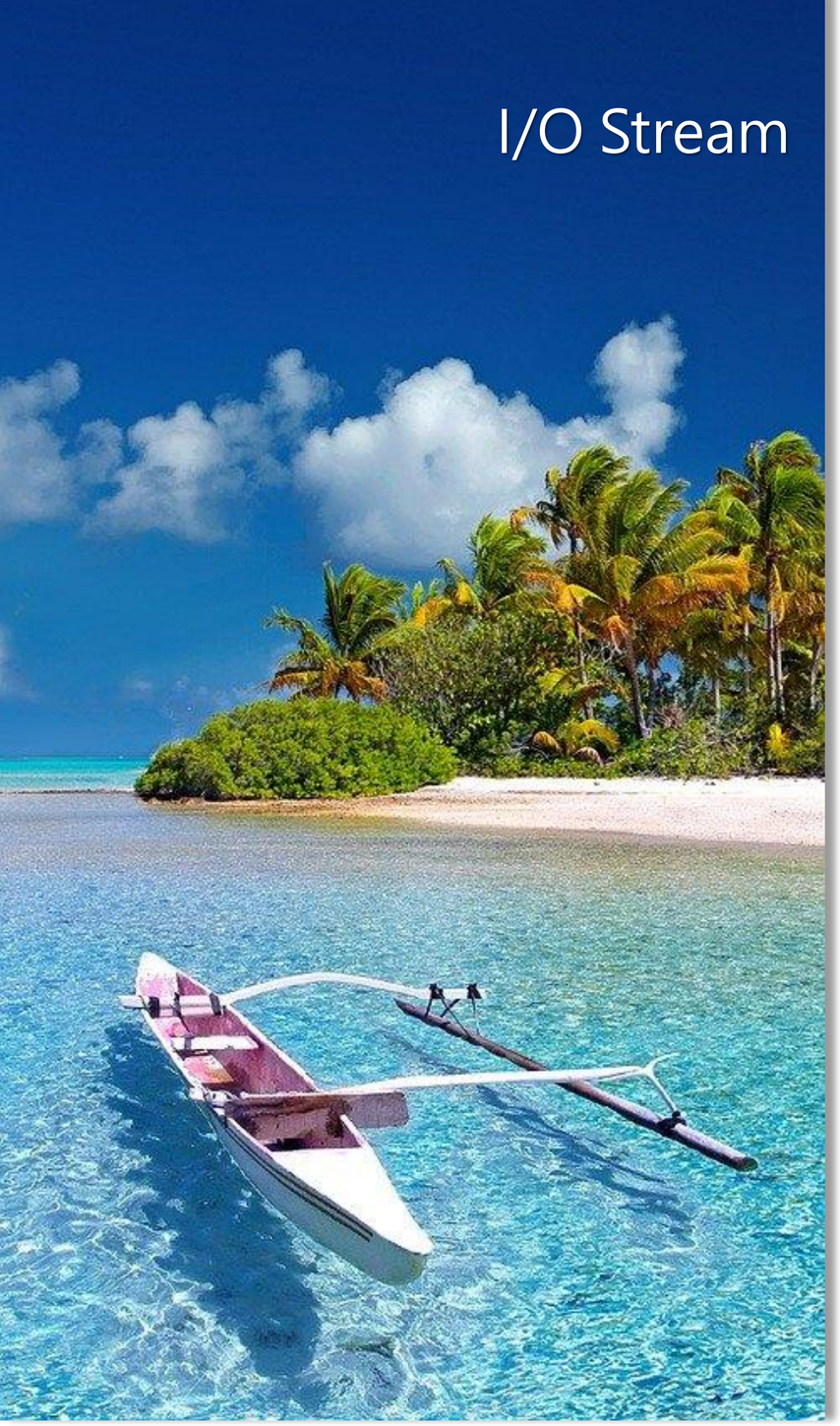
字元流-輸入

```
//1.File類的實體化
File file=new File("hello.txt");
//2.資料流的實體化
    FileReader fr=new FileReader(file);
//3.資料流的操作
    int read = fr.read();
    while (read !=-1){
        System.out.print((char)read);
        read=fr.read();
    }
    System.out.println();
//4.資料流的關閉
    fr.close();
```



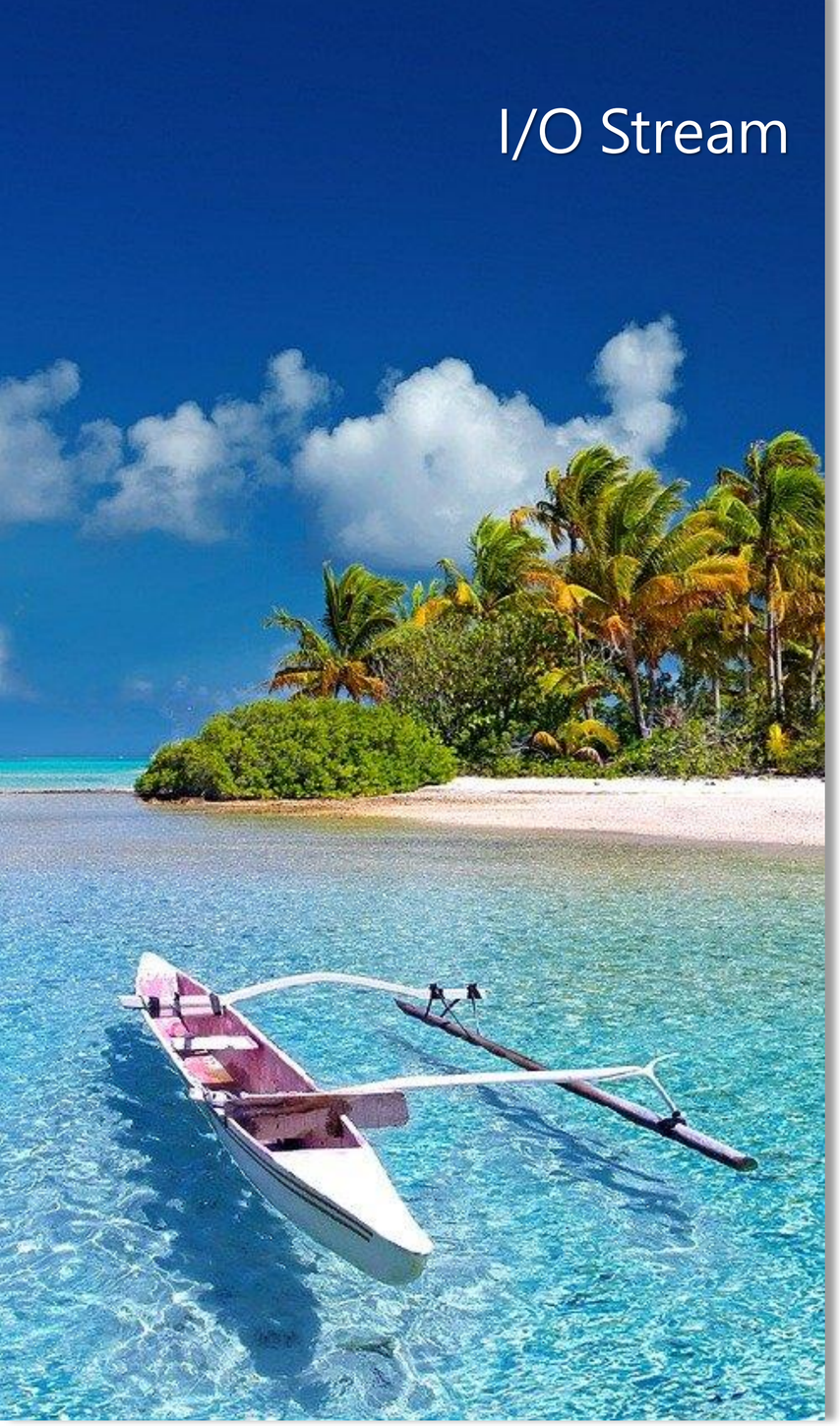
try-with-resource

```
FileReader fr=null;
        try {
                //1.File類的實體化
                File file=new File("hello.txt");
//2.資料流的實體化
                    fr=new FileReader(file);
                //3.資料流的操作
                    int read = fr.read(); //讀不到拋異常,跳出,利用finally關閉
                    while (read !=-1){
                       System.out.print((char)read);
                       read=fr.read();
                    }
                    System.out.println();
                }catch(Exception e) {}
                finally {
                //4.資料流的關閉
                    try {
                                    fr.close();
                            } catch (Exception e) {
                                    // TODO Auto-generated catch block
                                    e.printStackTrace();
                            }
                }
```



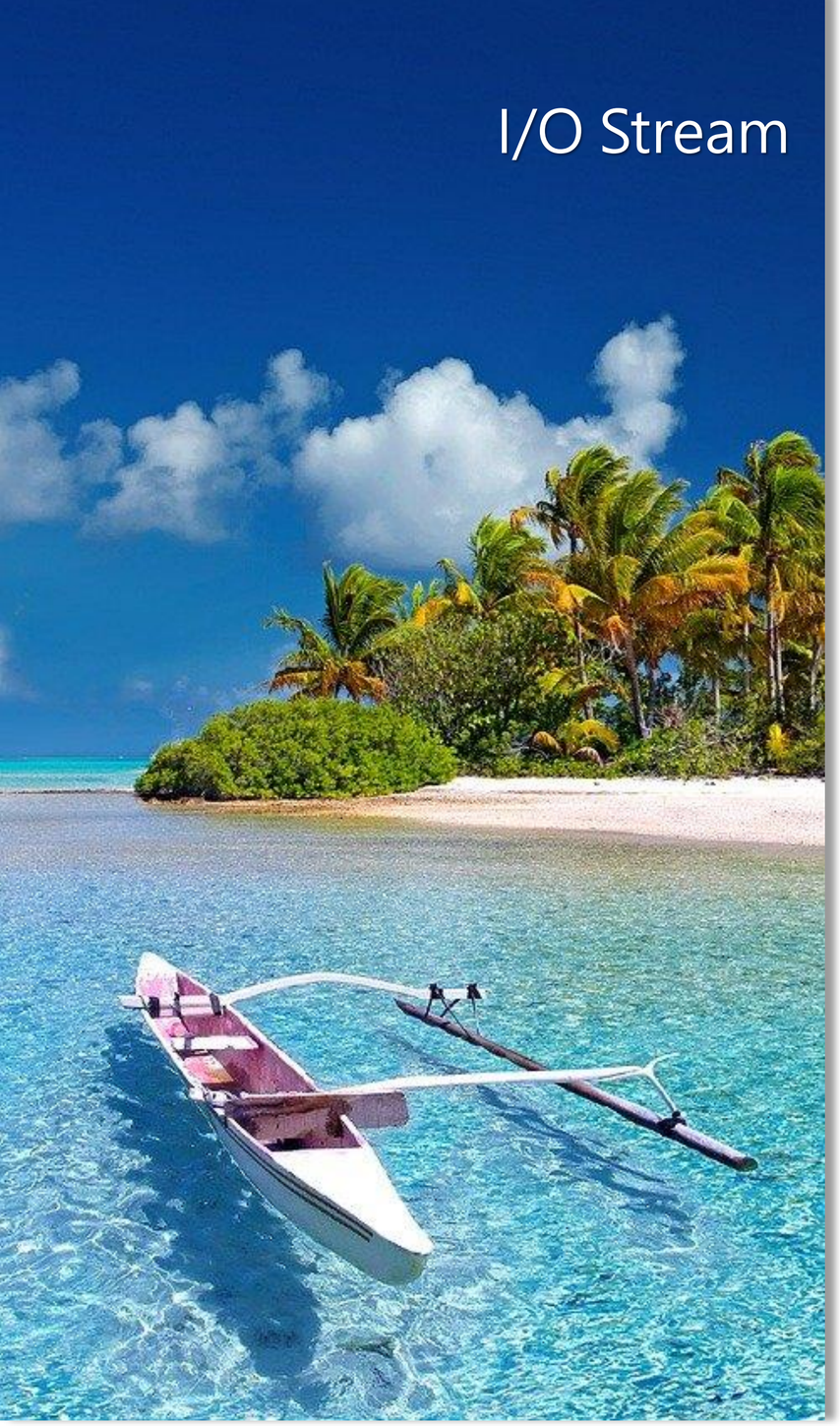
try-with-resource

```
//建立hello.txt檔案時需在files面板中建立,路徑為相對於專案
try (FileReader fr=new FileReader("hello.txt")){
        //3.資料流的操作
            int read = fr.read();
            while (read !=-1){
               System.out.print((char)read);
               read=fr.read();
            }
            System.out.println();
        }catch(Exception e) {}
```



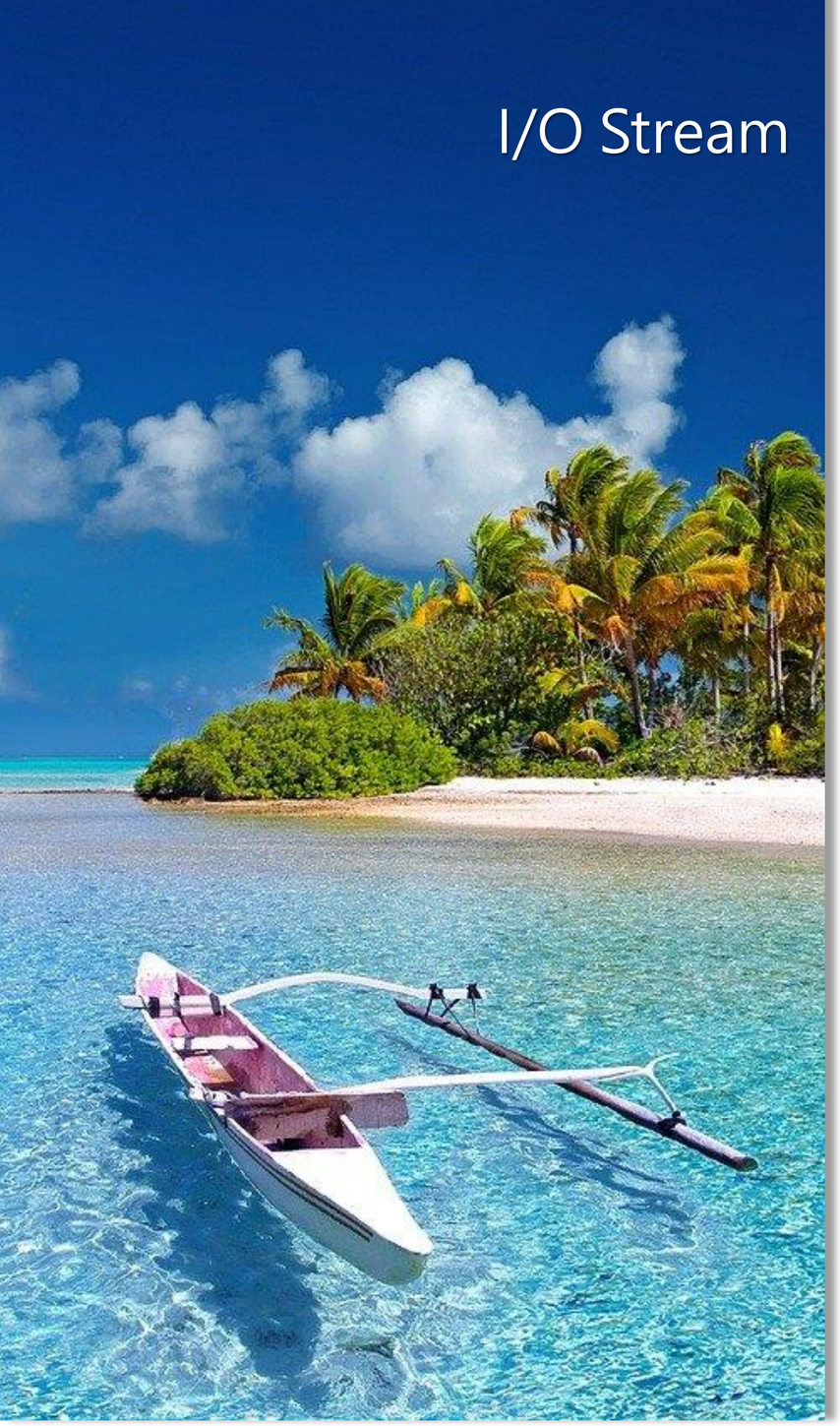
try-with-resource

- 實作 AutoCloseable介面
- 所有支援資源自動關閉的物件資源,都必須實作AutoCloseable介面



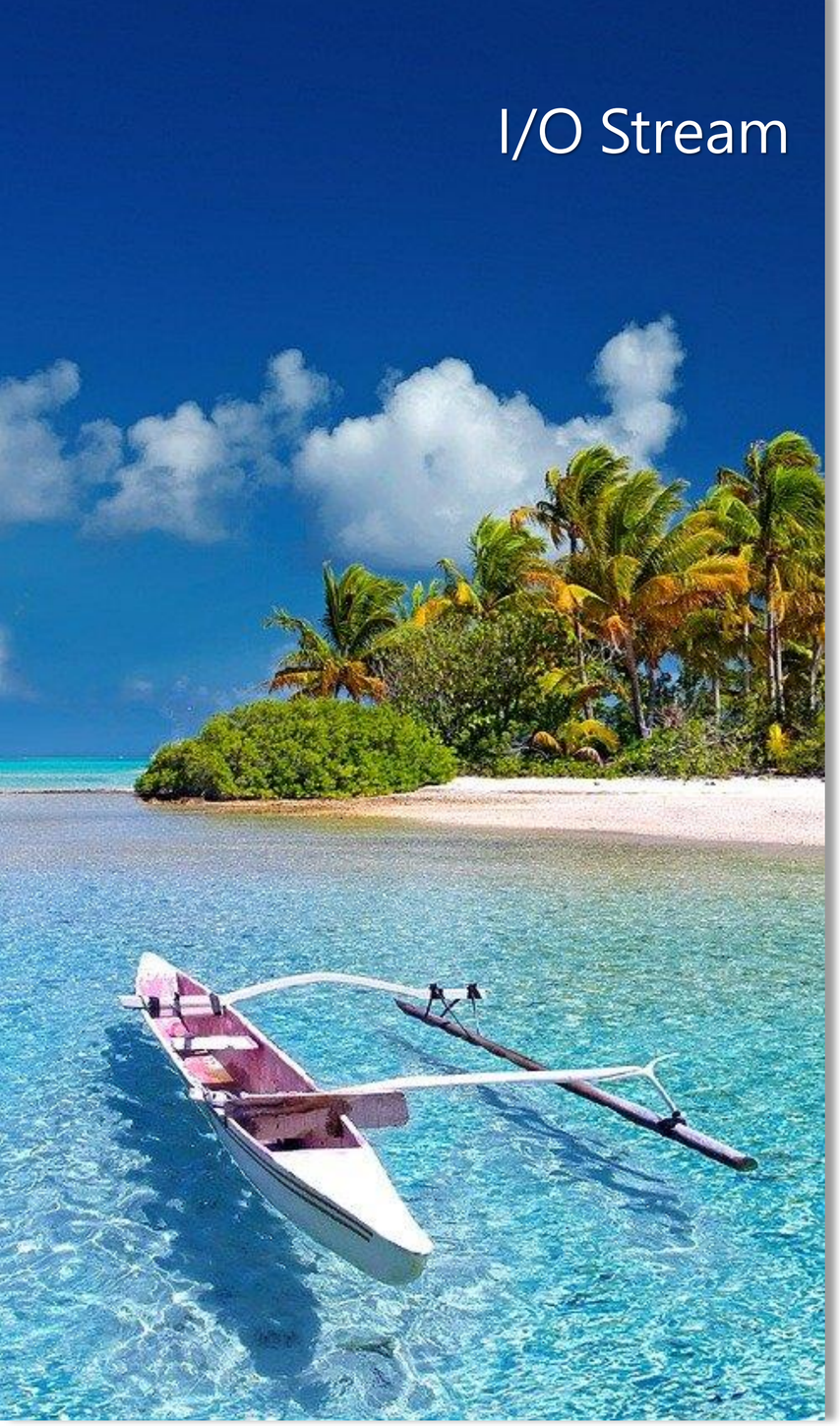
字元流- -輸入

```
File file=new File("hello.txt");
      FileReader fr=new FileReader(file);
      char charBuffer[]=new char[2];
      int len=0;
      while ((len=fr.read(charBuffer))!=-1){
           for(int m=0;m<charBuffer.length;m++)
             System.out.print(charBuffer[m]);
//for(int m=0;m<len;m++)
            //  System.out.print(charBuffer[m]);
  }
      System.out.println();
      fr.close();
```



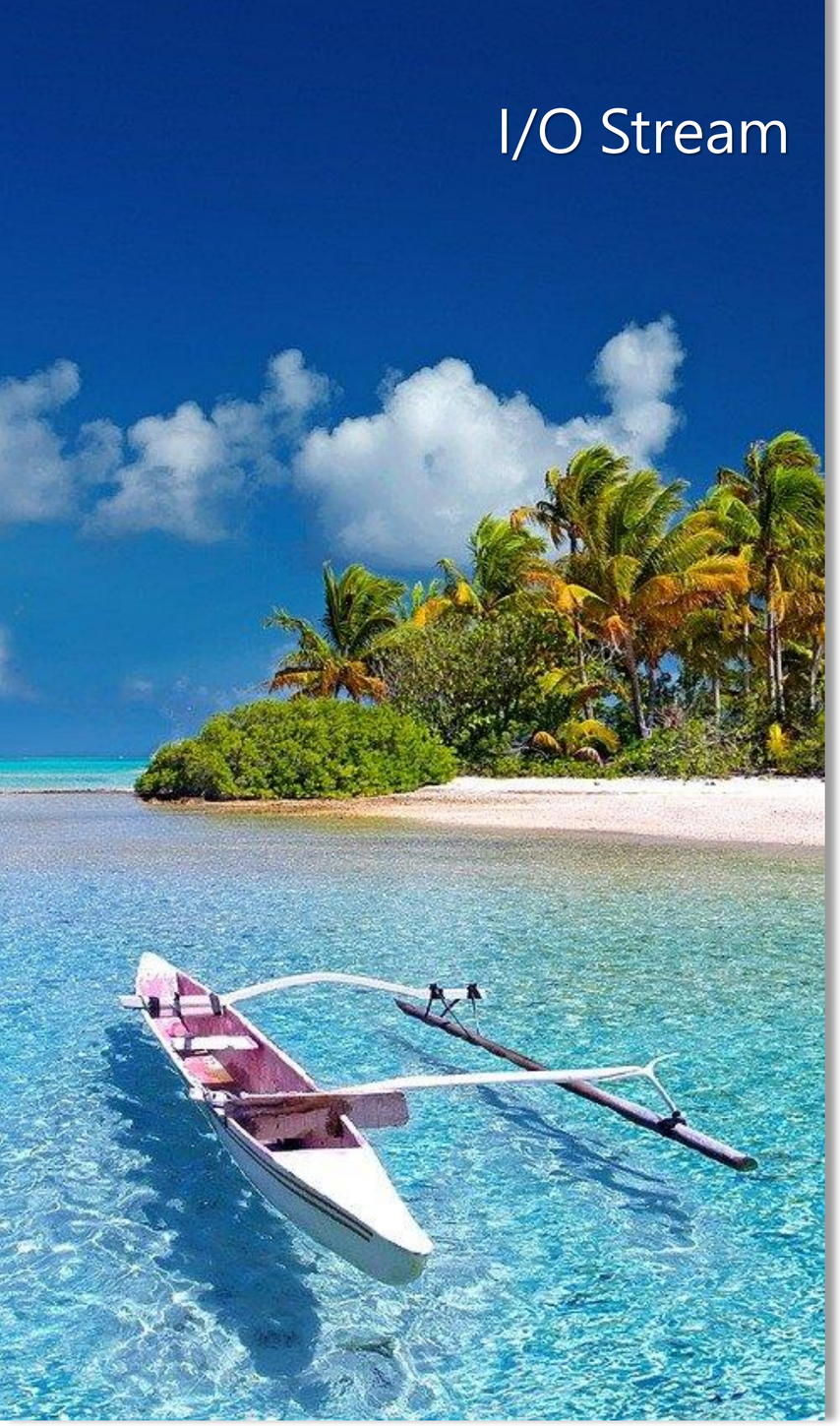
字元流- -輸出

- File file=new File("test.txt");
- //FileWrite(套用不同建構子即可完成擴充,預設為覆蓋)
- FileWriter fr=new FileWriter(file);
- fr.write("今天天氣很好\n");
- fr.write("OCA OCP PASS");
- fr.close();



字元流-輸入/輸出

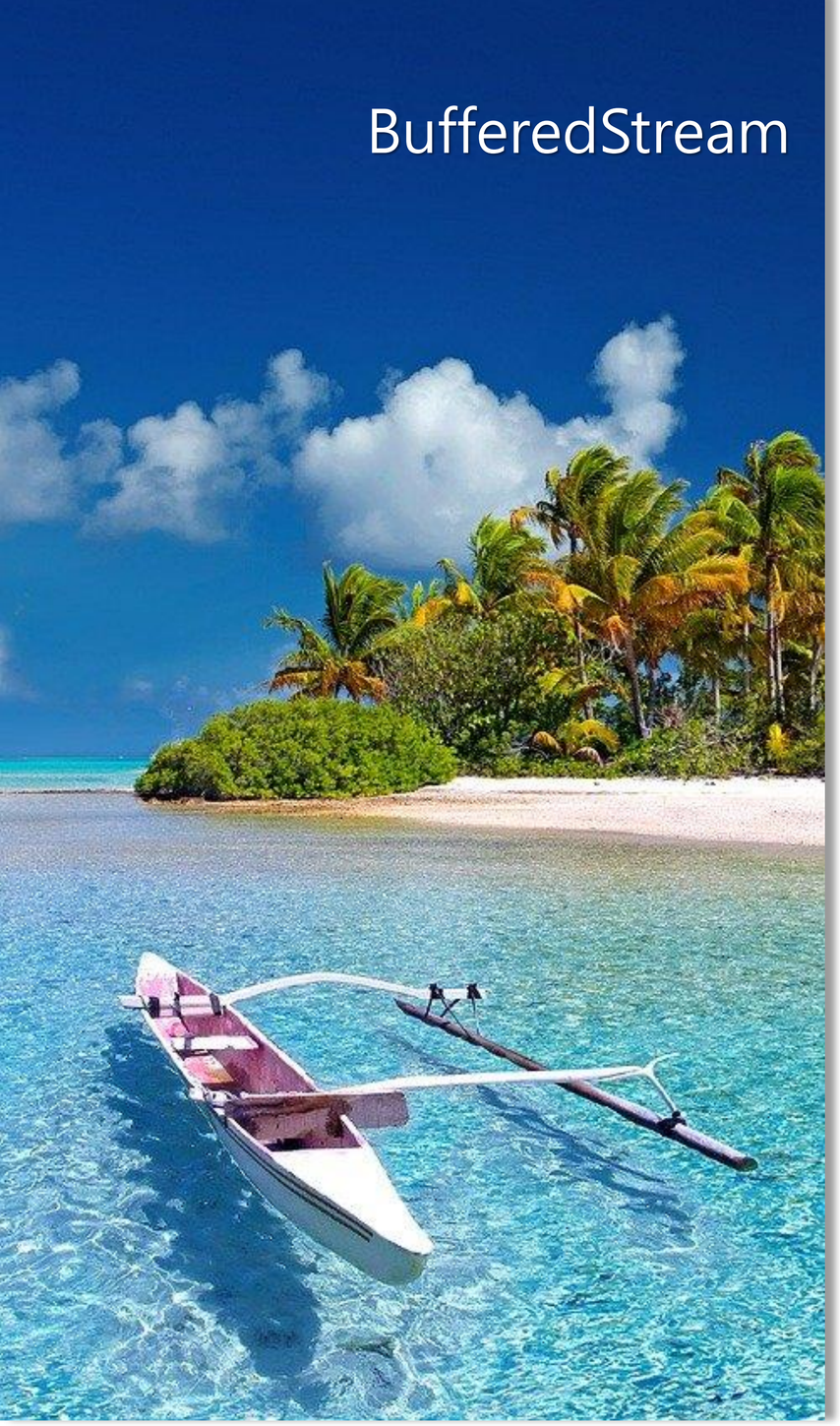
```
File fileSrc=new File("source.txt");
    File fileDest=new File("destination.txt");
    FileReader fr=new FileReader(fileSrc);
    FileWriter fw=new FileWriter(fileDest);
    char[] charBuffer=new char[3];
    int len;
    while((len=fr.read(charBuffer))!=-1)
    {
       String str=new String(charBuffer,0,len);
       fw.write(str);

    }
    fr.close();
    fw.close();
```



位元流-輸入/輸出

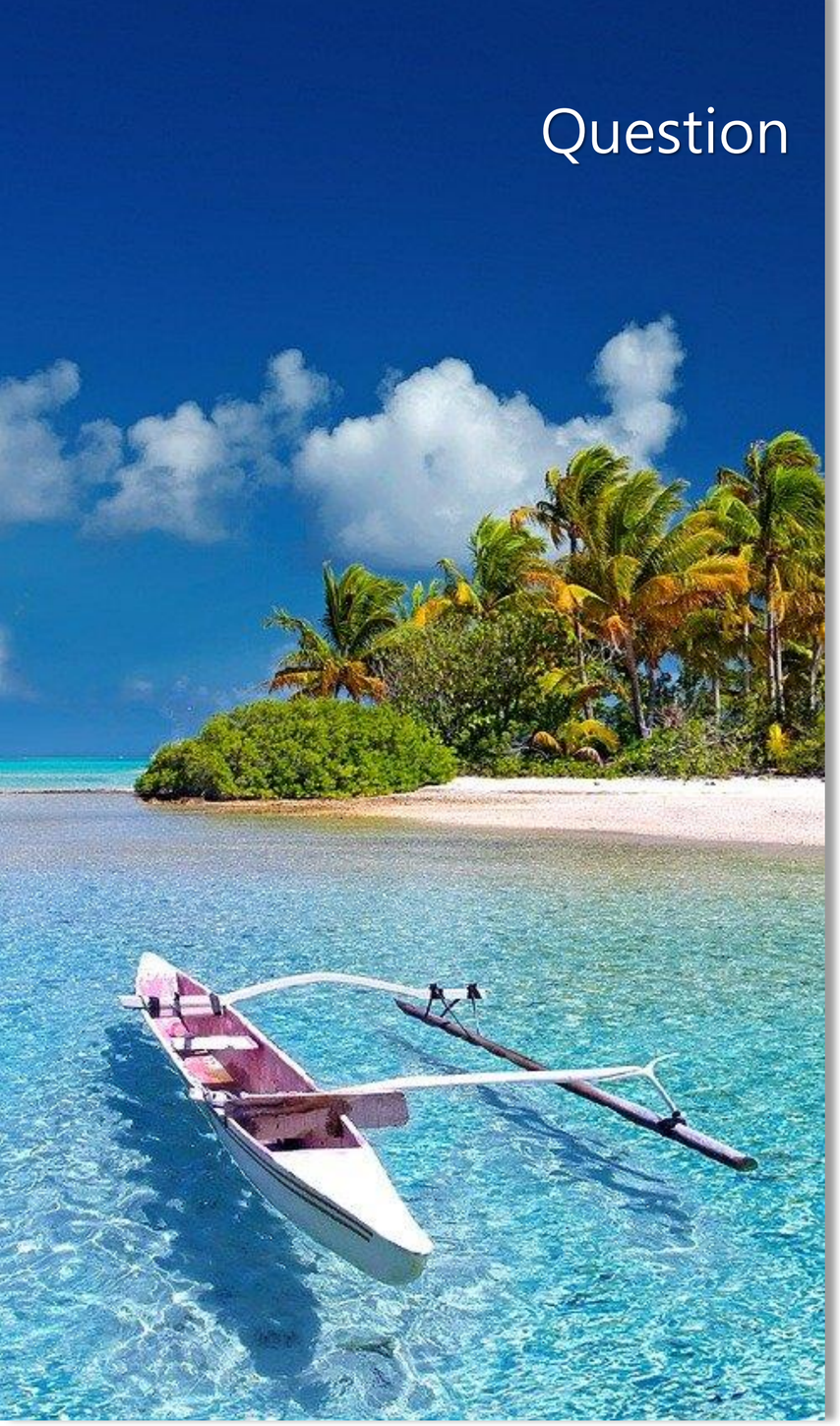
```
File fs=new File("a5.jpg");
    File fd=new File("a6.jpg");
    FileInputStream fis=new FileInputStream(fs);
    FileOutputStream fos=new FileOutputStream(fd);
    byte[] bsBuffer=new byte[1024];
    int len;
    while((len=fis.read(bsBuffer))!=-1)
    {
       fos.write(bsBuffer,0,len);
    }
    fis.close();
    fos.close();
```

BufferedStream



- 處理流 BufferedStream-用來提升流的存取速度
  - BufferedInputStream/BufferedOutputStream
  - 要先建立節點流後才可再建立處理流
  - 關閉時需先關閉處理流再關閉節點流
  - File fileOpen=new File(srcPath);
  - File fileWrite=new File(destPath);
  - FileInputStream in=new FileInputStream(fileOpen);
  - FileOutputStream out=new FileOutputStream(fileWrite);
  - BufferedInputStream bin=new BufferedInputStream(in);
  - BufferedOutputStream bout=new BufferedOutputStream(out);
  - int data;
  - byte buff[]=new byte[1024];
  - while((data=bin.read(buff))!=-1) {
  - bout.write(buff,0,data);
  - }
  - bout.close();
  - bin.close();

Question

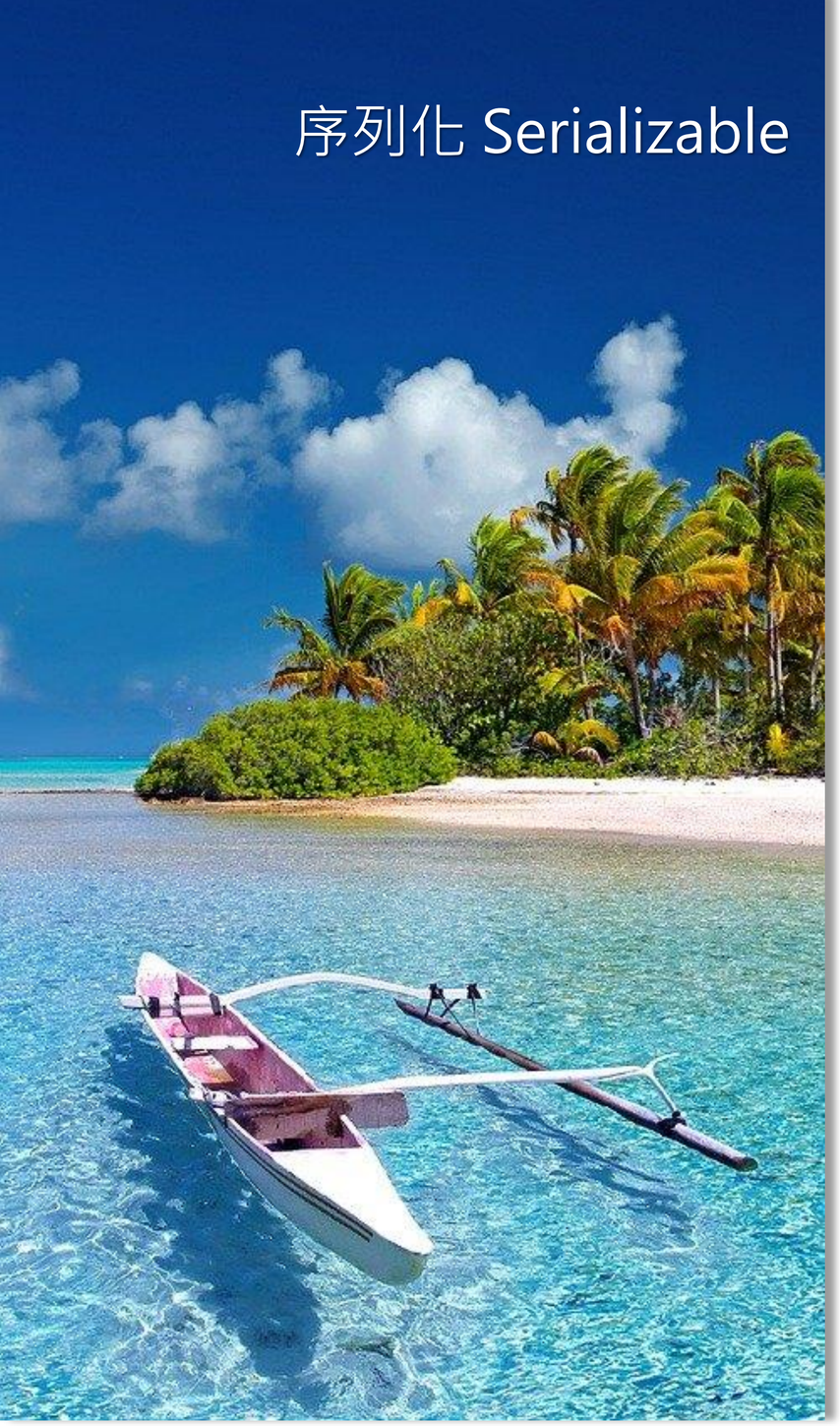


You want to read data through the reader Object. Which statement inserted on line 1 will accomplish this ?

```
char[] characters=new char[100];
try(FileReader reader=new FileReader("file_to_path");) {
    // line 1
    System.out.println(String.valueOf(characters));
}catch(IOException e) {
    e.printStackTrace();
}
```

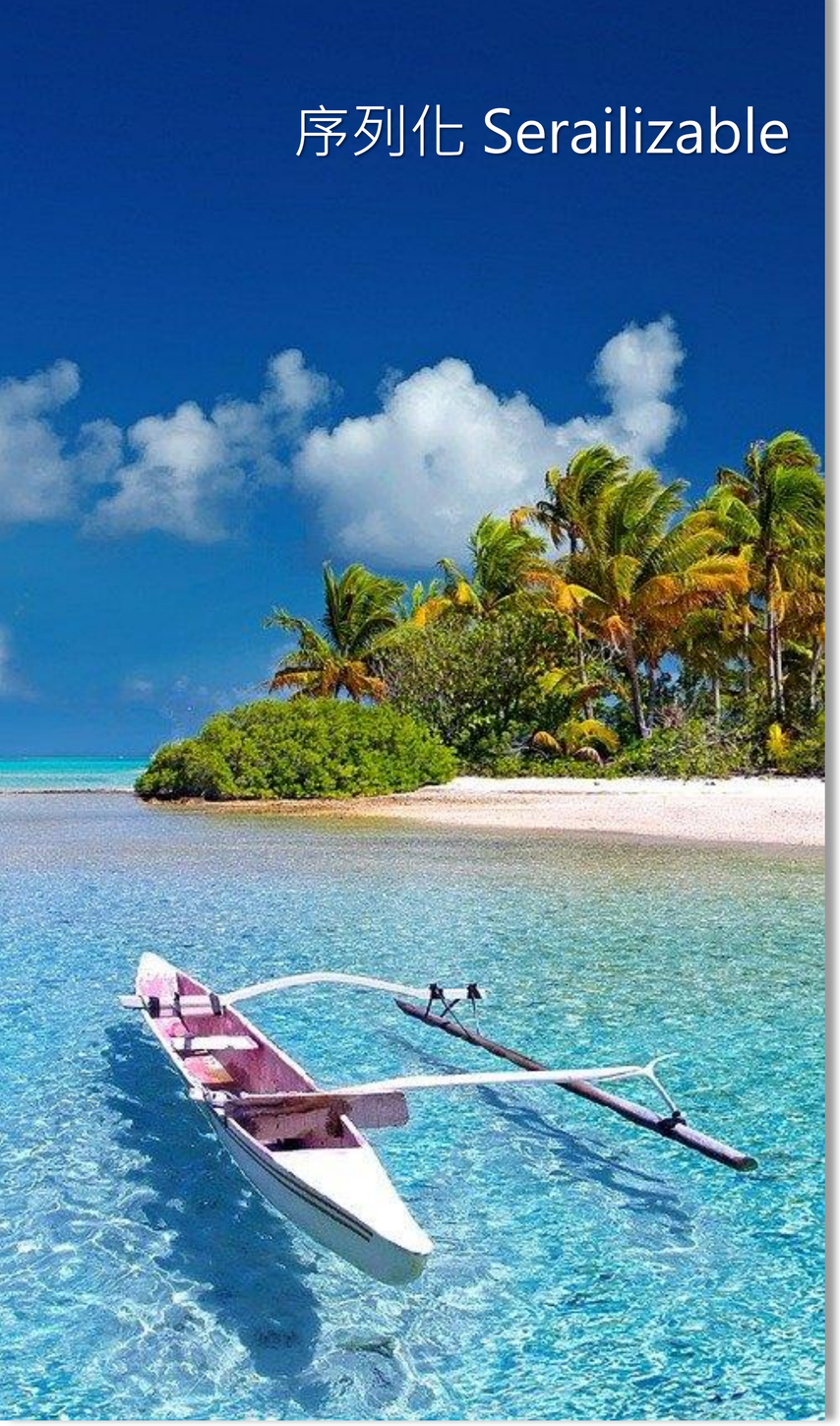
- ◯ A. characters.read( );
- ◯ B. characters = reader.read();
- ◯ C. reader.readLine( );
- ◯ D. reader.read(characters);

序列化 Serializable



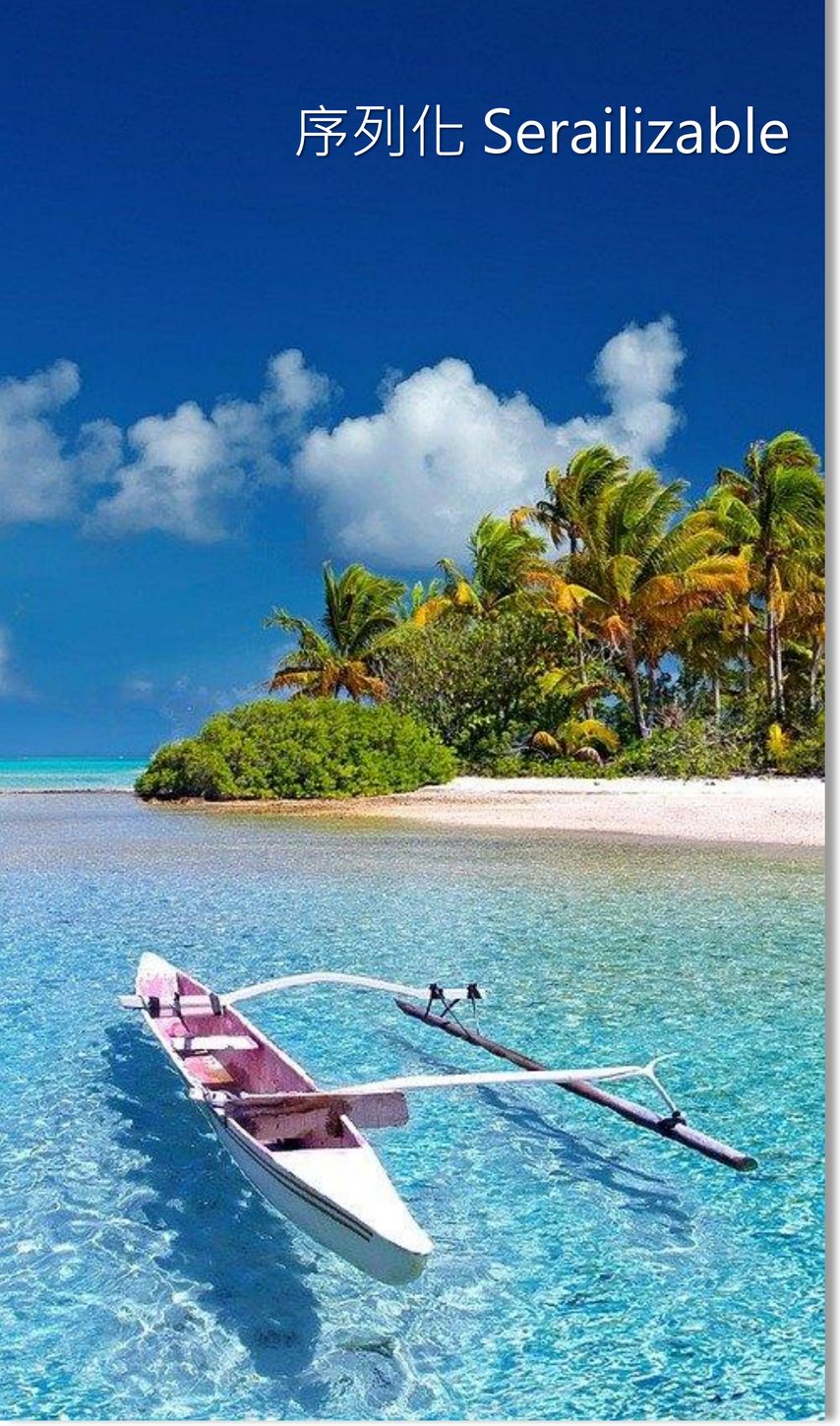
- 物件序列化機制:把java物件轉換成與平台無關的位元流(Byte),進而允許把這種二進制的資料流持久地保存在硬碟上,或通過網路將這種二進制的資料流傳輸到另一個網路節點,當其它程序取了這個資料流,就可以反序列化為原來物件
- 序列化的好處在於可將任何實現Serializable介面的物件轉化為位元數據,使其保存和傳輸時可被還原
- 序列化是RMI(Remote Method Invoke 遠程方法調用) 過程的參數和返回值都必須實現的機制,而RMI是java EE的基礎
- 如果需要讓某個物件支持序列化機制,則必需讓物件所屬的類別及其屬性是可序列化的,為了讓類別是可序列化的,則該類別必須實現二個介面其中一個
- Serializable或Externalizable

序列化 Serailizable



- //序列化 用ObjectOutputStream 類保存基本資料型態或物件的機制
- ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("info.dat"));
- oos.writeObject(new String("巨匠電腦"));
- oos.flush();
- oos.close();
- //反序列化 用ObjectInputStream 類讀取基本資料型態或物件的機制
- ObjectInputStream ois=new ObjectInputStream(new FileInputStream("info.dat"));
- Object obj = ois.readObject();
- String str=(String)obj;
- System.out.println(str);
- ois.close();
- PS:不能序列化 transient 修飾的成員變數

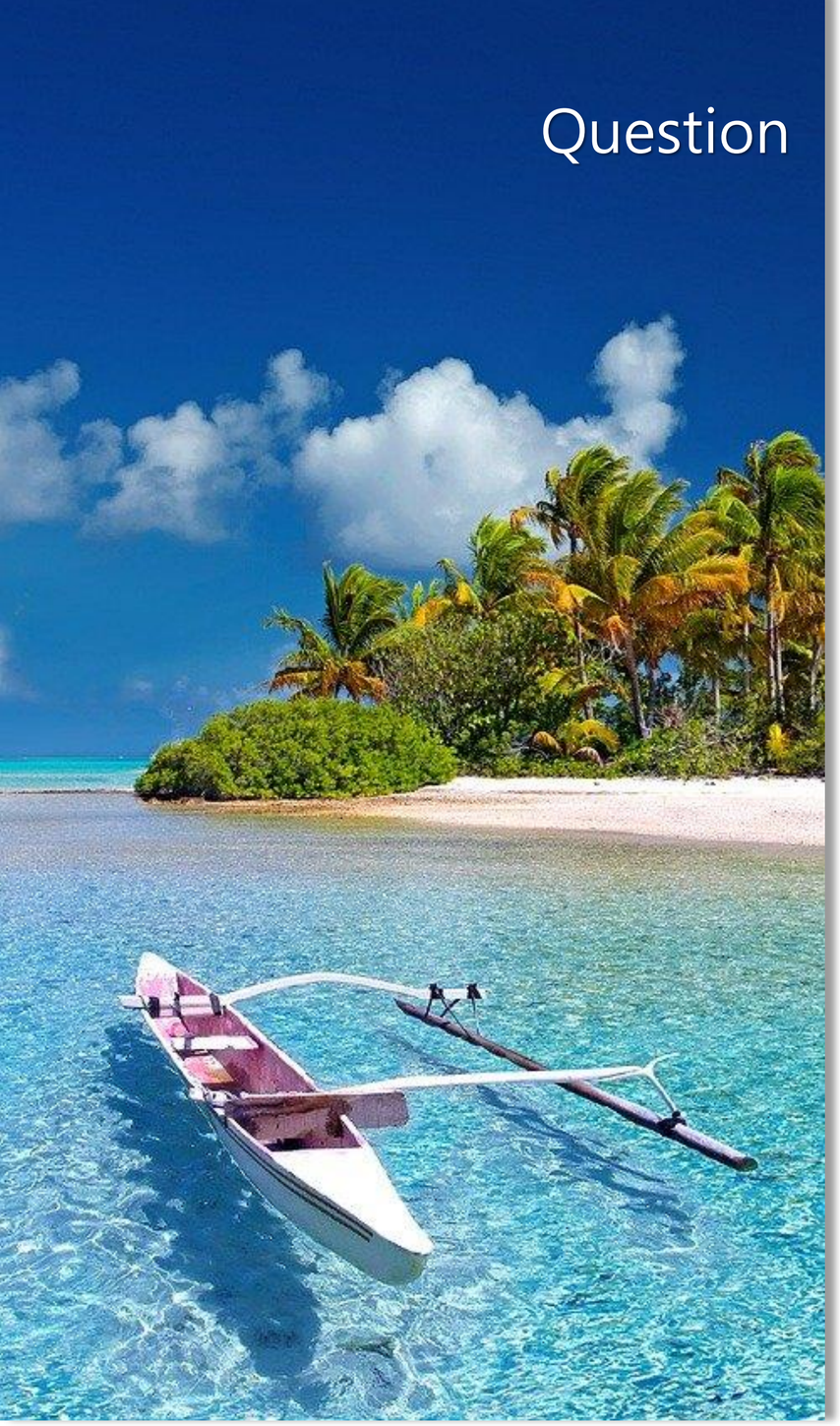
序列化 Serailizable



```
//若沒有實作Serializable是無法序列化
class Person implements Serializable{
        static final long serialVersionUID = 1234567842L;
        //不想序列化的欄位前加上 transient
         private String name;
        private int age;
        public Person(String name, int age) {
                super();
                this.name = name;
                this.age = age;
        }
        public Person() {super();}
        public String getName() {       return name;}
        public void setName(String name) {
                this.name = name;
        }
        public int getAge() {return age;}
        public void setAge(int age) {
                this.age = age;
        }
        @Override
        public String toString() {
                return "Person [name=" + name + ", age=" + age + "]";
        }
}
```

```
//序列化
        ObjectOutputStream oos=new ObjectOutputStream(new FileOutputStream("info.dat"));
        oos.writeObject(new String("巨匠電腦"));
        oos.flush();
        oos.writeObject(new Person("巨匠美語",20));
        oos.flush();
        oos.close();
        //反序列化
        ObjectInputStream ois=new ObjectInputStream(new FileInputStream("info.dat"));
        Object obj = ois.readObject();
        String str=(String)obj;
        System.out.println(str);
        Person p = (Person)ois.readObject();
        System.out.println(p);
        ois.close();
```

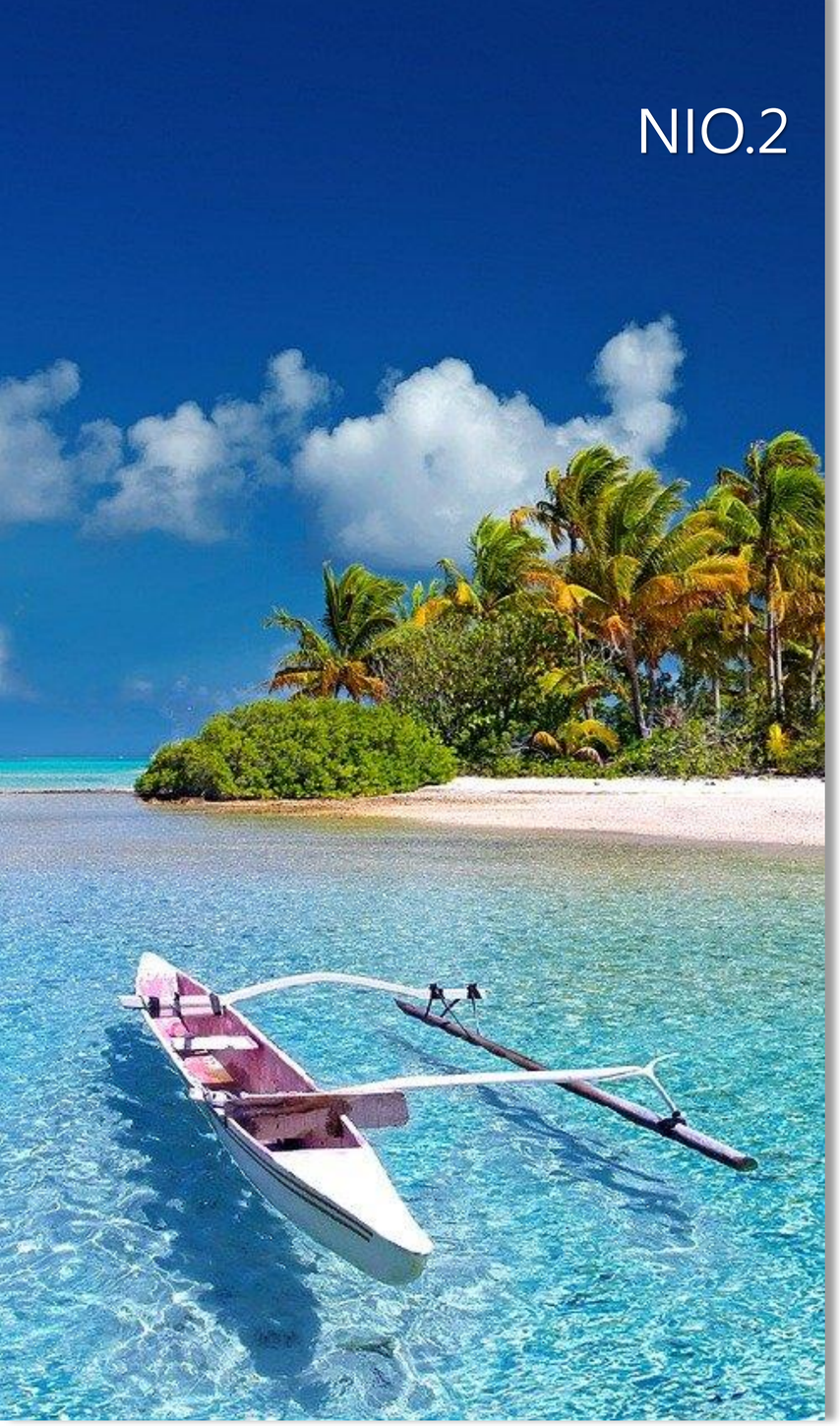
Question



You want to implement the java.io.Serializable interface to the MyPersistenceData class. Which method should be overridden ?

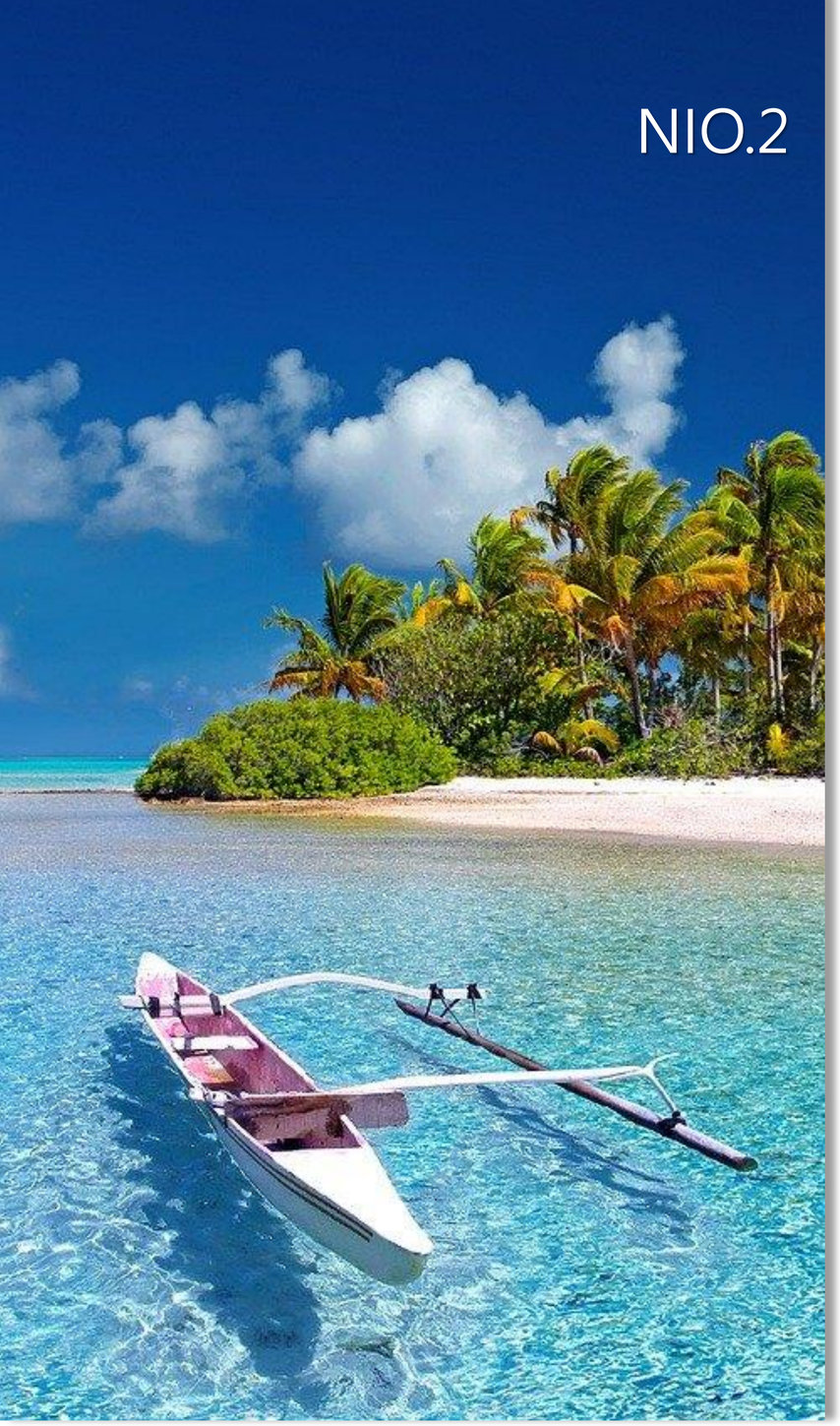
```
class MyPersistenceData{
    String str;
    private void methodA() {
        System.out.println("methodA");
    }
}
```

- ◯ A. Nothing.
- ◯ B. the readExternal method
- ◯ C. the readExternal and writeExternal method
- ◯ D. the writeExternal method



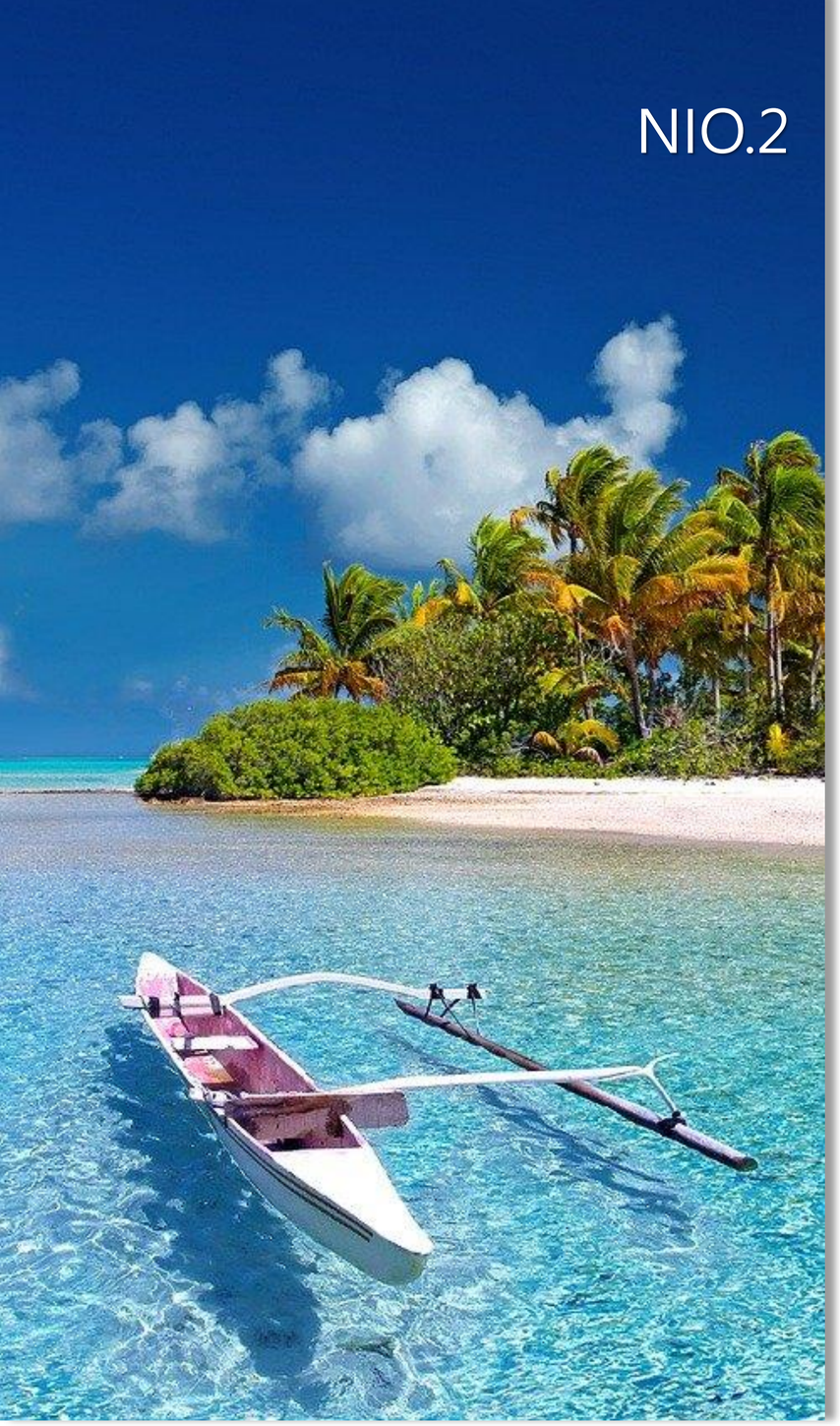
Path/Paths類別-操作路徑

- Path path=Paths.get("c:\\temp\\company\\Finance\\salary.dat");
- Path path1=Paths.get("salary.dat");
- System.out.println("getFileName:"+path.getFileName());
- System.out.println("getParent:"+path.getParent());
- System.out.println("getNameCount:"+path.getNameCount());
- System.out.println("subPath(0,2):"+path.subpath(0,2));
- System.out.println("isAbsolute:"+path.isAbsolute());
- System.out.println("toAbsolutePath:"+path.toAbsolutePath());
- System.out.println("toAbsolutePath:"+path1.toAbsolutePath());
- System.out.println("toUri:"+path.toUri());



Path/Paths類別-操作路徑

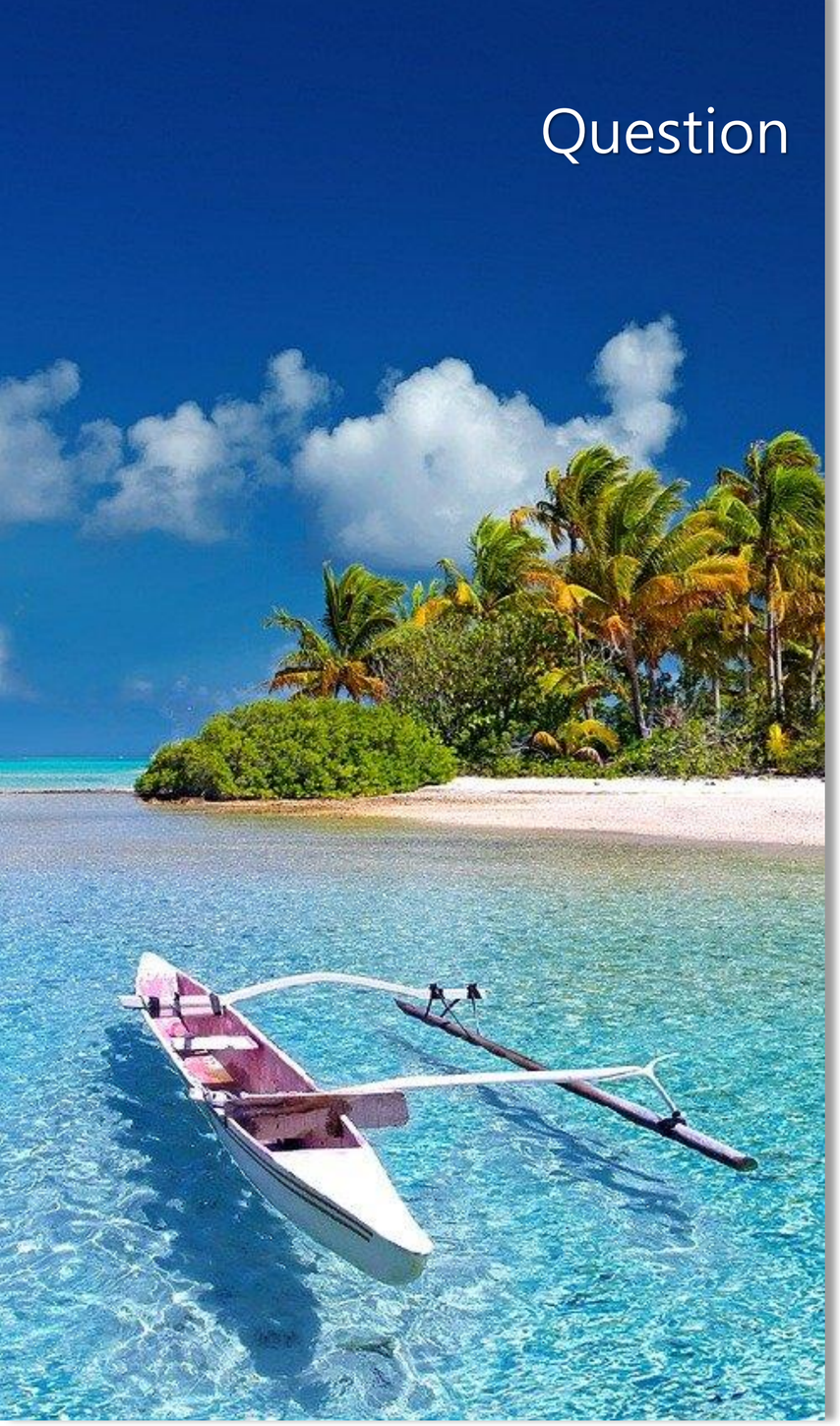
- Path path=Paths.get("c:\\temp\\company\\Finance\\salary.dat");
- Path path1=Paths.get("c:\\temp\\company\\..\\Finance\\salary.dat").normalize();
- System.out.println(path1);
- //路徑結合 resolve()
- Path path2=Paths.get("c:\\temp\\company\\");
- Path path3=Paths.get("salary.dat");
- Path path4=Paths.get("\\salary.txt");
- System.out.println(path2.resolve(path3));
- System.out.println(path2.resolve(path4));
- //比較頭尾是否相同startsWith/endsWith
- Path path5=Paths.get("c:\\temp");
- System.out.println(path.startsWith(path5));
- System.out.println(path.endsWith(path3));



Files類別-操作檔案

- Path src=Paths.get("d:\\temp\\company\\Finance\\salary.dat");
- Path dest=Paths.get("d:\\temp\\company\\salary.dat");
- Files.copy(src, dest,StandardCopyOption.REPLACE_EXISTING);

Question



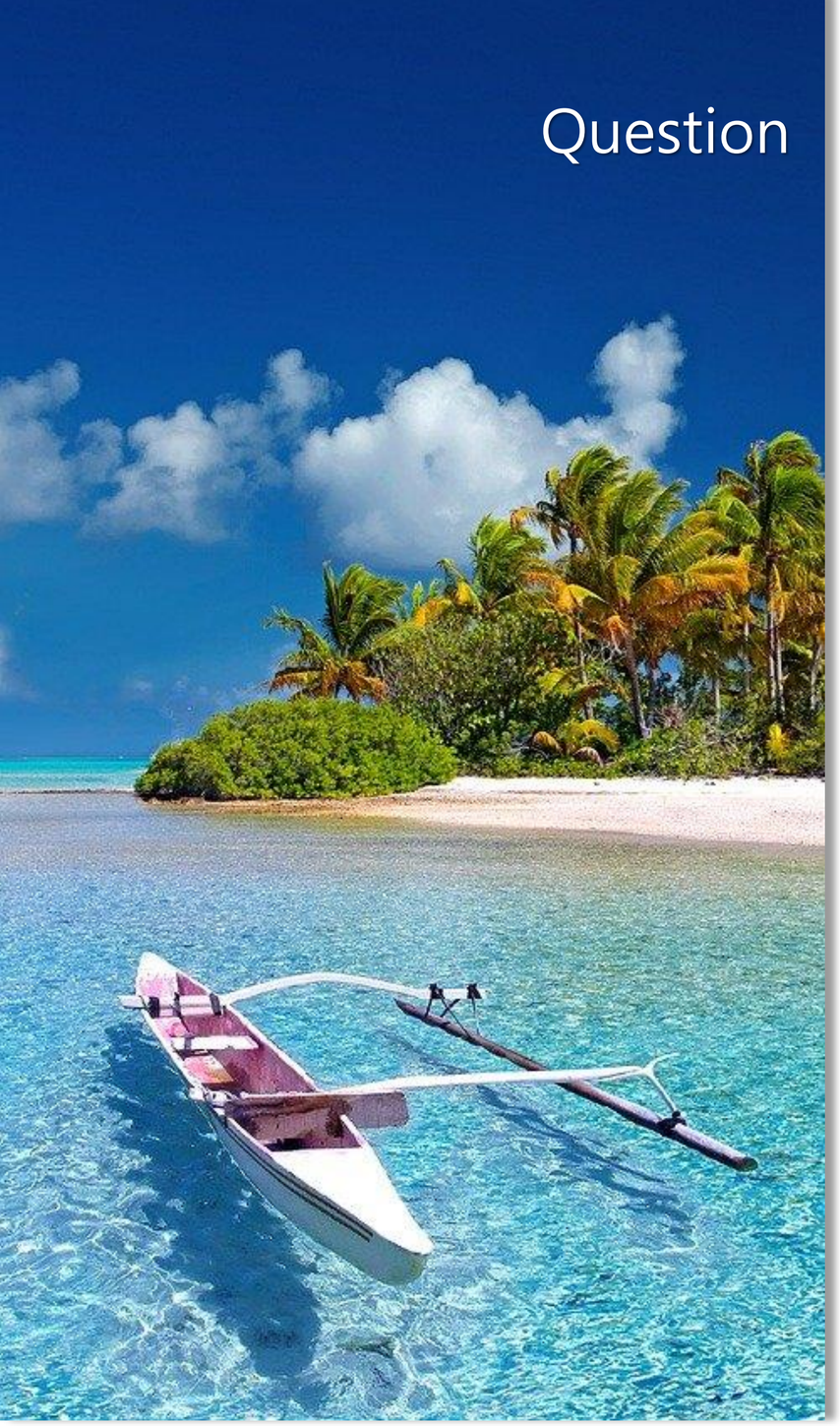
What is the output?

Given:

String originalPath = "data\\projects\\a-project\\..\\..\\another-project"; Path path = Paths.get(originalPath);

System.out.print(path.normalize());

- A. data\another-project
- B. data\projects\a-project\another-project
- C. data\\projects\\a-project\\..\\..\\another-project
- D. data\projects\a-project\..\..\another-project



Question

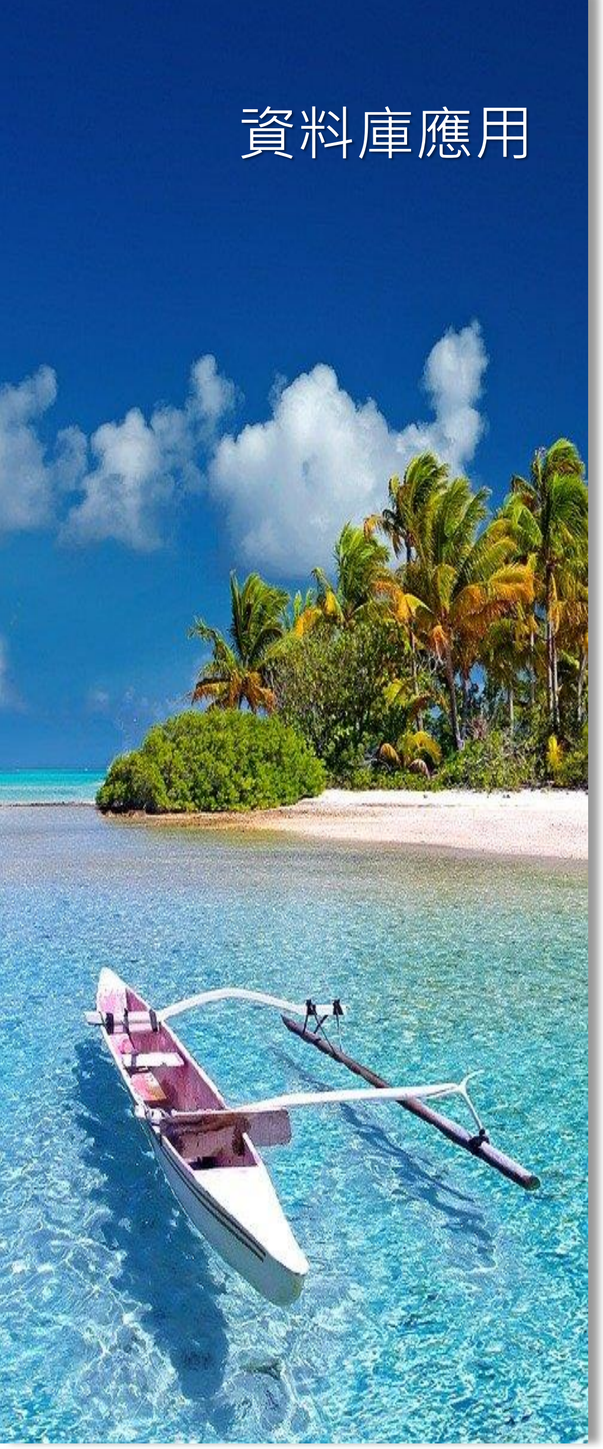
Assume the file on path does not exist. What is the result?

```
public class Test52 {
    public static void main(String[] args) {
        try {
            Path path = Paths.get("/u01/work/filestore.txt");
            boolean result = Files.deleteIfExists(path);
            if (result)
                System.out.println(path + "is deleted.");
            else
                System.out.println(path + "is not deleted.");
        } catch (IOException e) {
            System.out.println("Exception");
        }
    }
}
```

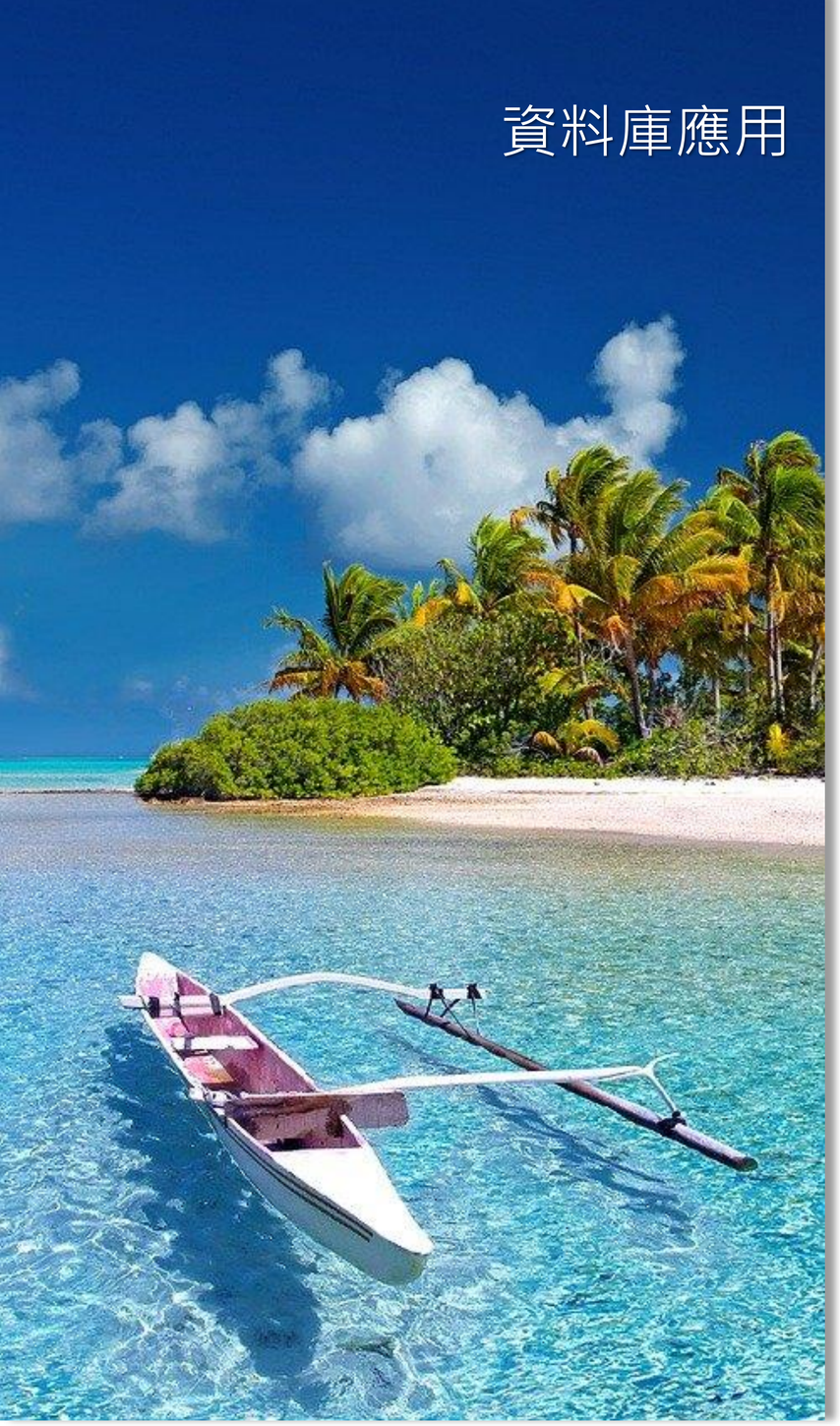
- ◯ A. The compilation fails.
- ◯ B. /u01/work/filestore.txt is not deleted.
- ◯ C. Exception
- ◯ D. /u01/work/filestore.txt is deleted.

資料庫 DATABASE

資料庫應用



Oracle
Oracle Driver
MS SQL
MS SQL Driver
Others
JDBC ODBC Bridge
DriverManager
Connection
PreparedStatement
Statement
CallableStatement
Result
Result
Result

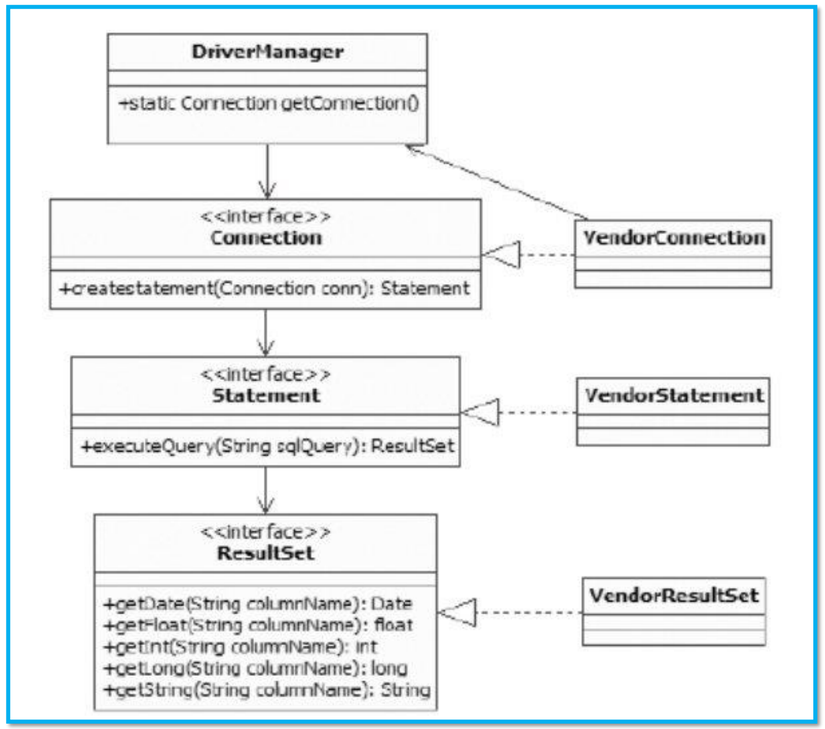


資料庫應用

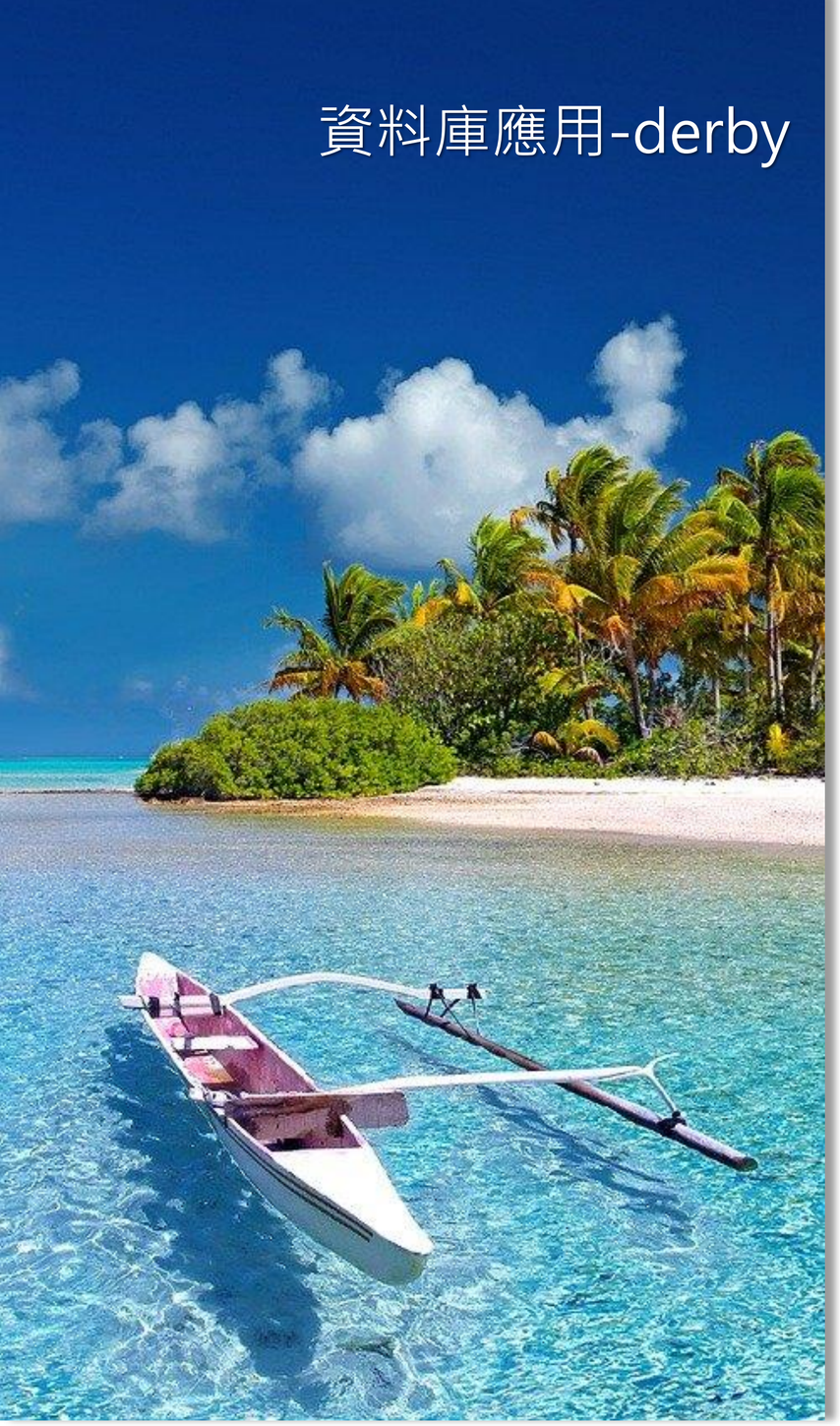
JDBC API 主要由一個CLASS和三個interface組成，除了DriverManager外都是介面，關係是

- 使用DriverManage取得Connection
- 使用Connection取得Statement
- 使用Statement取得ResultSet

因為市場上有多種資料庫且由不同廠商負責，連線資料庫的機制該由廠商提供，因此JDBC只定義抽象介面，由各家資料庫廠商提供實作類別

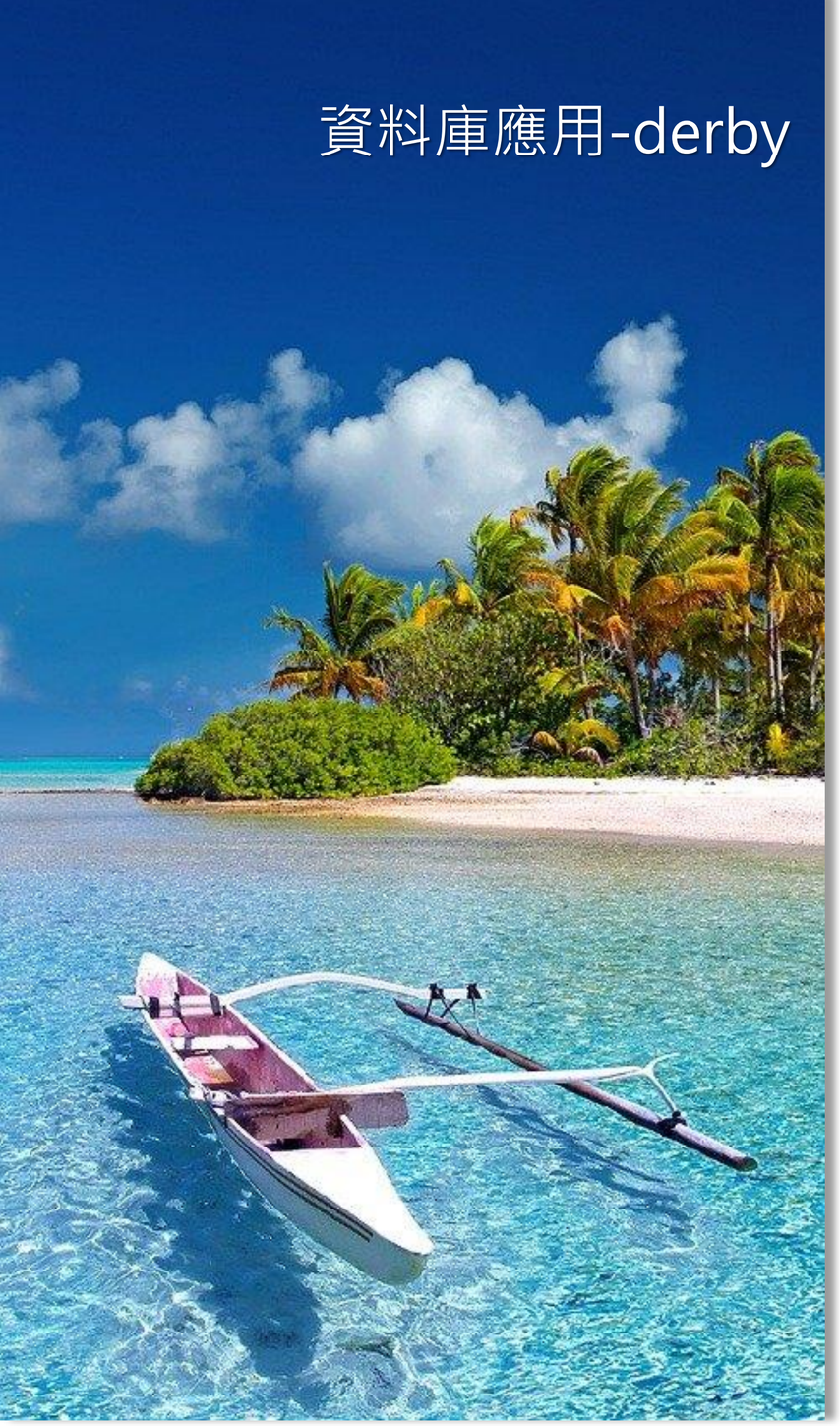


資料庫應用-derby



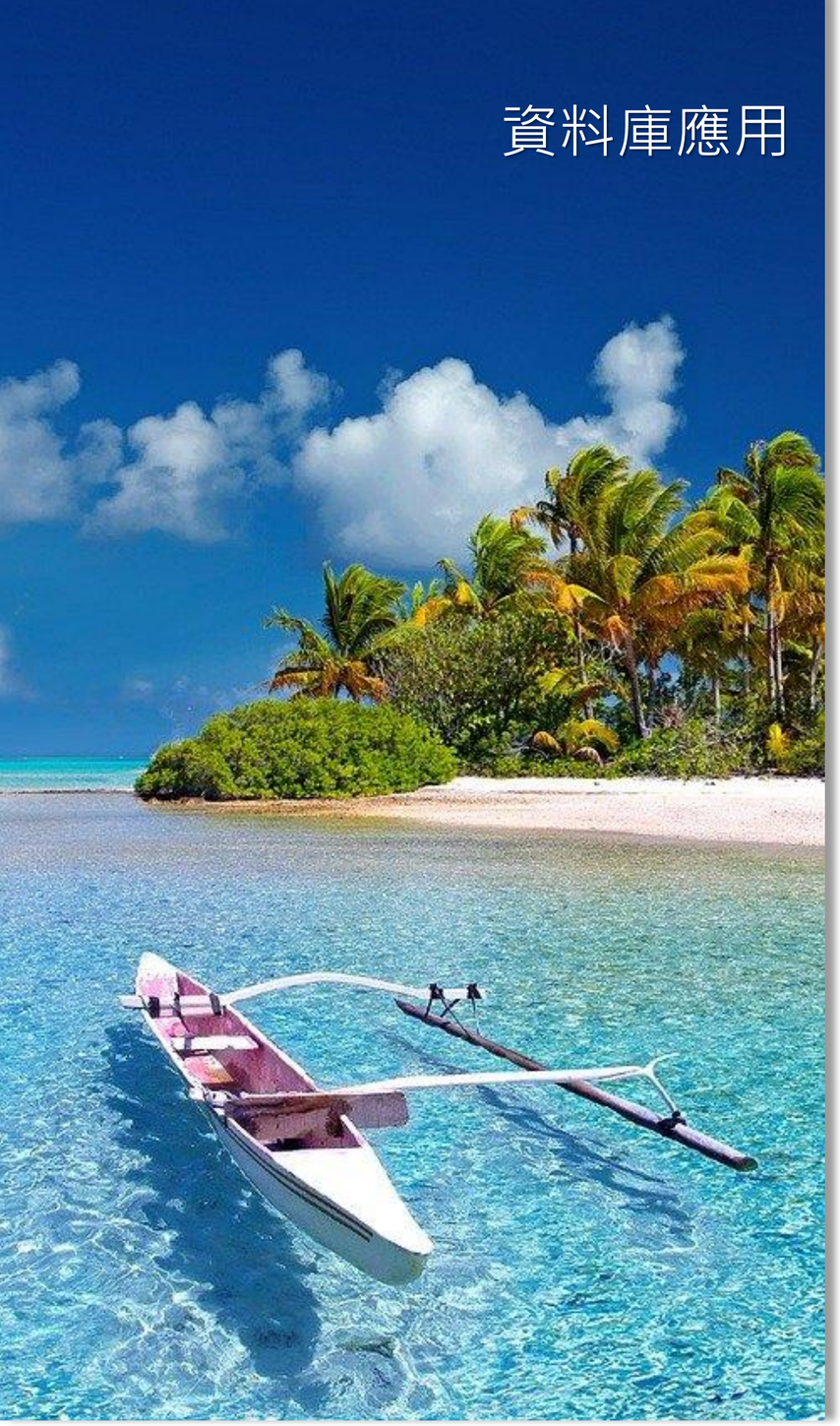
- 到網址下載 壓縮檔 https://db.apache.org/derby/derby_downloads.html2.
- 解壓縮後 在eclipse專案內新增一資料夾,名稱可叫做libs
- 將步驟2解壓縮後的 lib資料夾內的
  - derby.jar
  - derbyrun.jar
  - derbyshared.jar
  - derbytools.jar
  - 等幾個檔案複製到eclipse的libs資料夾內
- 在libs資料夾內的所有 .jar 檔上點右鍵 > Build path > add to build path 即會存於Referece Libraries資料夾內
- 主方法內撰寫如下程式 Connection conn = DriverManager.getConnection ("jdbc:derby:hello;create=true"); create=true 代表若是找不到資料庫,則自動建立資料庫

資料庫應用-derby



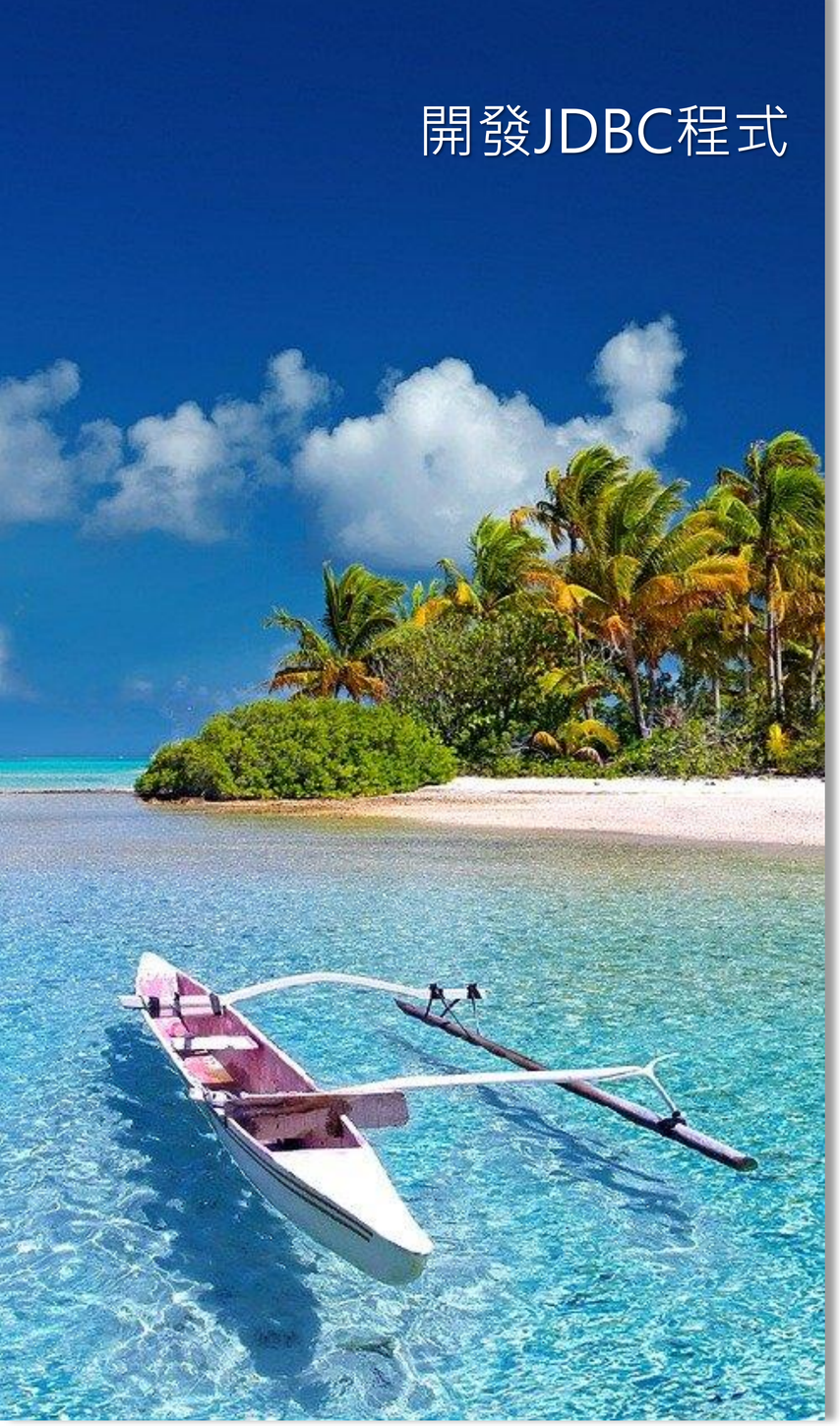
```
String url = "jdbc:derby:amashop;create=true";
Connection conn = null;
Statement st = null;
conn = DriverManager.getConnection(url);
System.out.println("資料庫連結成功");
st = conn.createStatement();
// 建立資料表
String sql = "CREATE TABLE frp(" + "ID INTEGER not null primary key," + "NAME VARCHAR(10),"
                                + "GENDER VARCHAR(2)," + "EMAIL VARCHAR(50)" + ")";
st.executeUpdate(sql);
// 插入記錄
String sql2 = "INSERT INTO frp(id,name,gender,email)" + "VALUES(2,\'巨匠電腦\',\'先生\',\'pc@gmail.com\')";
st.executeUpdate(sql2);
// 查看記錄
String sql3 = "select * from frp";
ResultSet rs = st.executeQuery(sql3);
while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                String gender = rs.getString("gender");
                String email = rs.getString("email");
                System.out.printf("%d\t%s\t%s\t%s\n", id, name, gender, email);
        }
```

資料庫應用



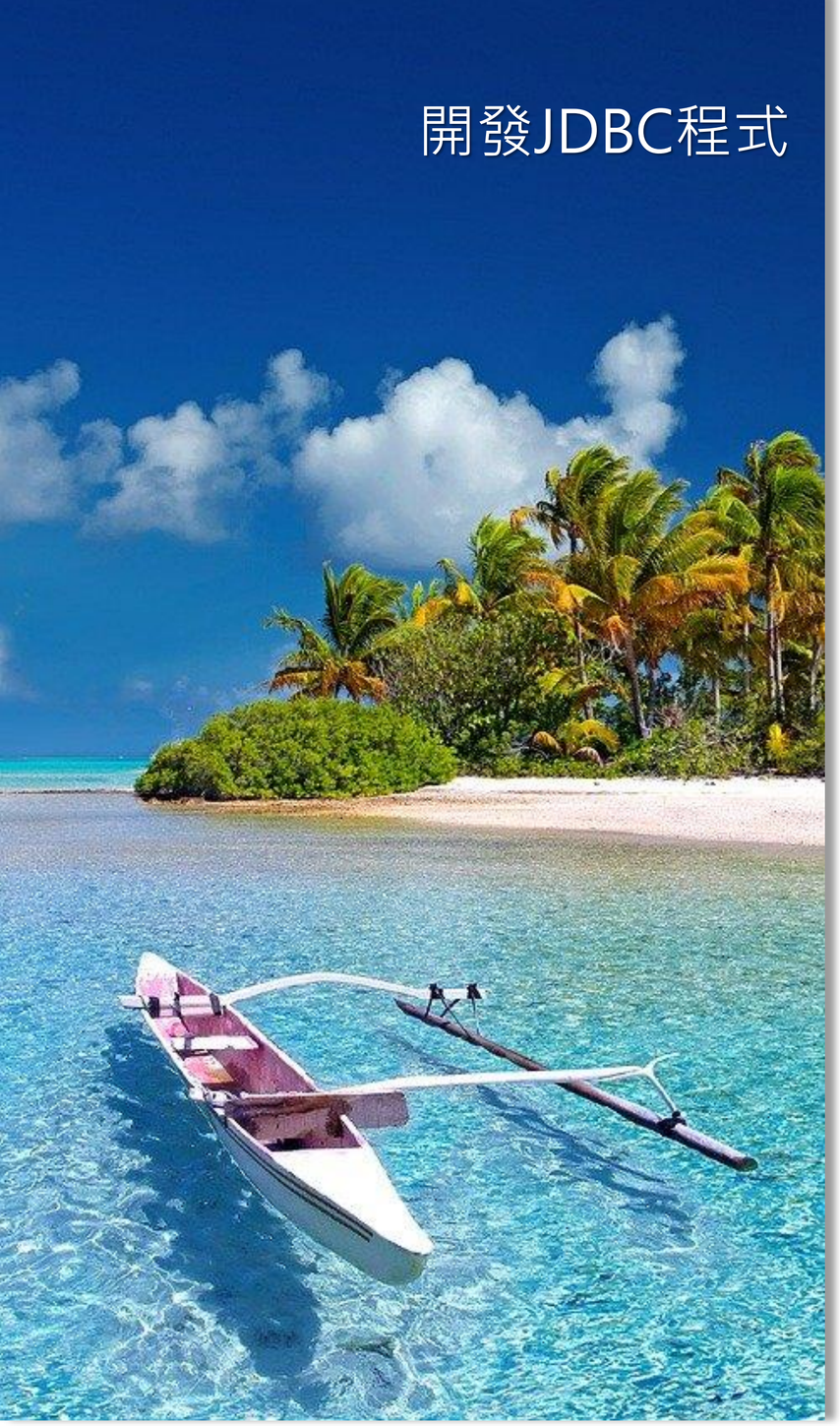
- 安裝mysql server,mysql workbench
- 進入mysql建立資料庫及資料表
- 下載安裝jdbc驅動程式以連線資料庫
  - 下載mysql jdbc j驅動程式.jar 下載網址 https://dev.mysql.com/downloads/connector/j/
  - 在eclipse環境之projects上點右鍵>Build path>add External Archives
  - 而後選擇下載之mysql connector的jar檔即可

開發JDBC程式



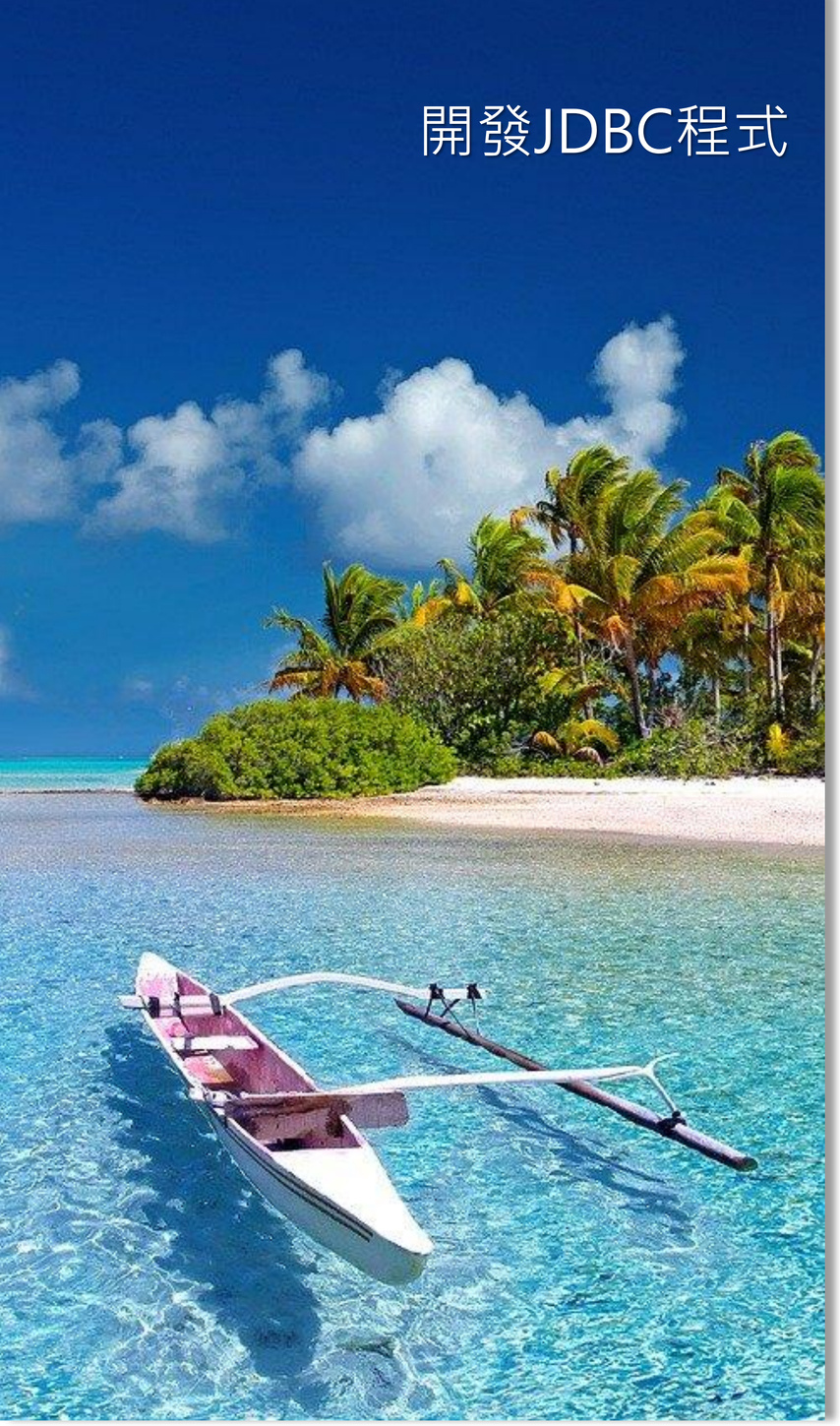
- 開發JDBC程式的幾個主要步驟
  1. 指定URL:URL是用來指出連線資料庫時使用的DRIVER名稱 / 種類，資料庫位置，或是其他建立連線時需要的一併提供的屬性名稱與值，語法為
     - jdbc:<driver>:[sub_protocol:][databaseName][;attribe=value]
  2. 使用DriverManager取得java.sql.Connection的物件，該物件將建立與資料庫的連線（Session)
     - Connection con=DriverMager.getConnection(url,username,password);
  3. 使用java.sql.Connection取得java.sql.Statement物件
     - Statement stmt=con.createStatement();
  4. 使用java.sql.ResultSet取得java.sql.Statement執行SQL後的查詢結果
     - String query= “SELECT * FROM EMPLOYEE”;
     - ResetSet rs=stmt.executeQuery(query);

開發JDBC程式



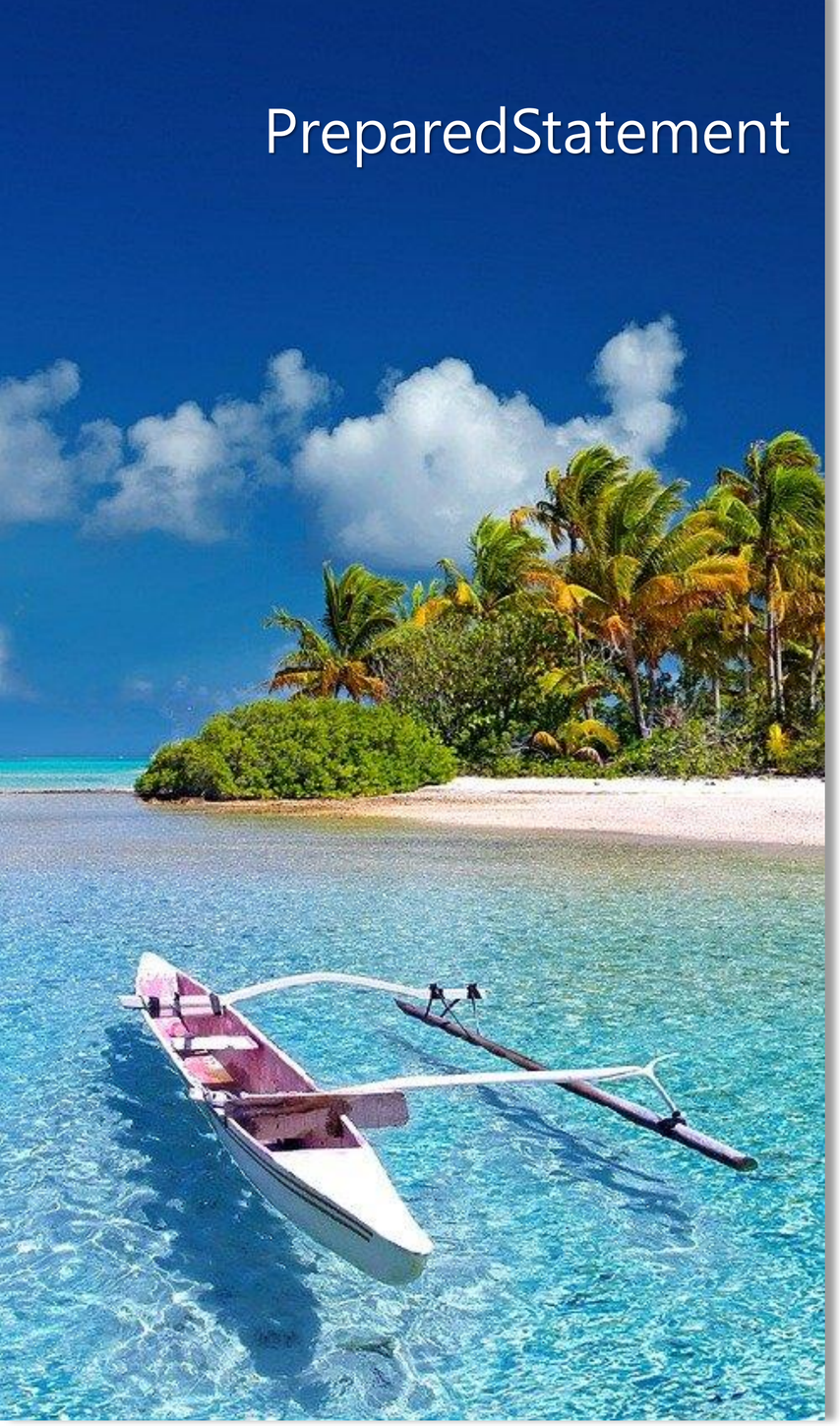
- 要執行SQL敘述(statement)時，必須要有Statement的物件，也可以說Statement介面是SQL statement的包覆類別
  - Statement stmt=con.createStatement();
  - String sql= “select * from table”;
  - ResultSet rs=stmt.executeQuery(sql);

開發JDBC程式



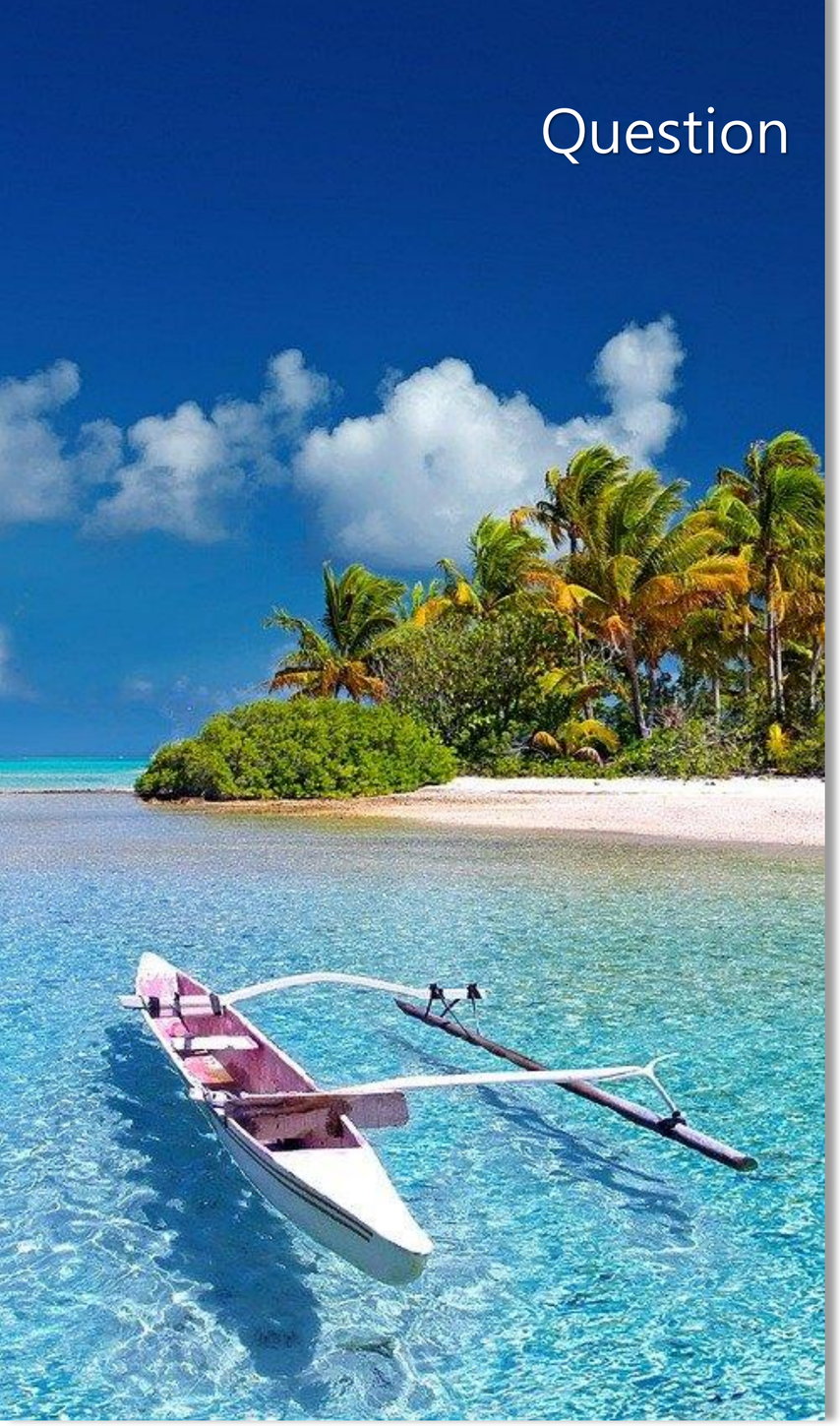
```
String url="jdbc:mysql://localhost:3306/amashop?useSSL=false&useUnicode=true&characterEncoding=utf8";
    String user="root";
    String pw="52680614";
    Connection conn=null;
    Statement st=null;
      try {
          conn=DriverManager.getConnection(url, user, pw);
          System.out.println("資料庫連結成功");
          st=conn.createStatement();
          String sql="select * from member";
          ResultSet rs=st.executeQuery(sql);
          while(rs.next()){
//注意資料表的資料型態是否相同
              int id=rs.getInt("mid");
              String name=rs.getString("mname");
              int age=rs.getInt("mage");
              String gender=rs.getString("msex");
//注意資料表的資料型態是否相同
              System.out.printf("%d\t%s\t%d\t%s\n",id,name,age,gender);
          }
      } catch (SQLException ex) {
          System.out.println(ex);
      }
```

PreparedStatement



```
String url =
"jdbc:mysql://localhost:3306/amashop?useSSL=false&useUnicode=true&characterEncoding=utf8";
        String usr = "root";
        String password = "52680614";
        Connection conn = DriverManager.getConnection(url, usr, password);
        String sql = "select * from member";
        String sql2 = "insert into member(mname,mage,msex) values(?,?,?)";
        PreparedStatement prepare = conn.prepareStatement(sql2);
        prepare.setString(1, "周杰倫");
        prepare.setInt(2, 100);
        prepare.setNull(3, java.sql.Types.VARCHAR);
        int executeUpdate = prepare.executeUpdate();
        PreparedStatement prepare2 = conn.prepareStatement(sql);
        ResultSet rst = prepare2.executeQuery();
        while (rst.next()) {
                int id = rst.getInt("mid");
                String name = rst.getString("mname");
                int age = rst.getInt("mage");
                String email = rst.getString("msex");
                System.out.println(id + " " + name + " " + age + " " + email);


        }
```

Question



Which statement inserted on line 14 sets NAME column to a NULL value?

Given the Customer table structure:

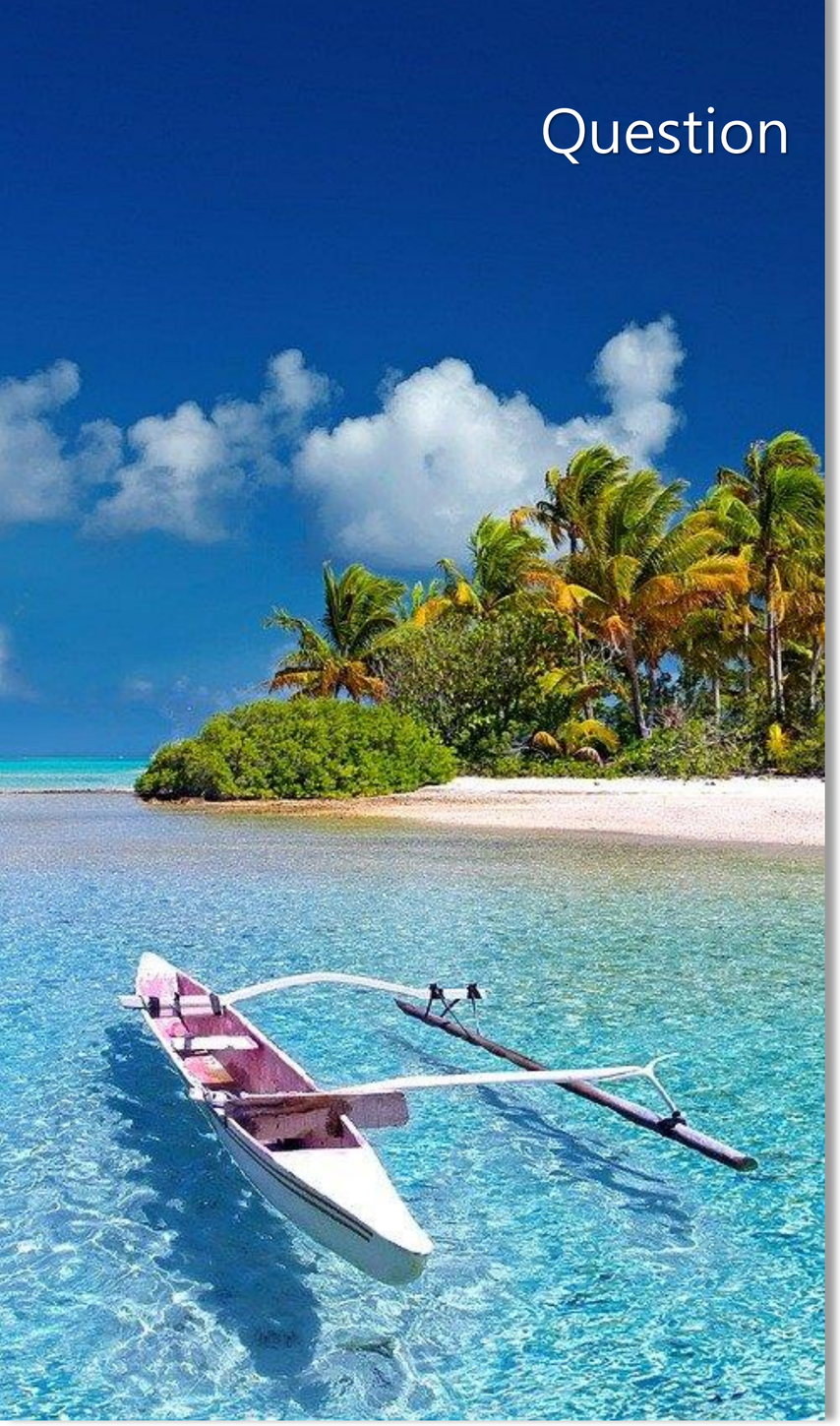
* ID Number Primary Key
* NAME Text Nullable2

Given code fragment:

```
PreparedStatement stmt=con.prepareStatement("INSERT INTO CUSTOMER VALUES(?,?)");
stmt.setInt(1,42);
/*Insert code here*/
int n=stmt.executeUpdate();
```

- ◯ A. Stmt.setNull(2, java.sql.Types.VARCHAR);
- ◯ B. Stmt.setNull(2 string, class);
- ◯ C. Stmt.setNull(2, null);
- ◯ D. Stmt.setNull(2, java.lang, string);

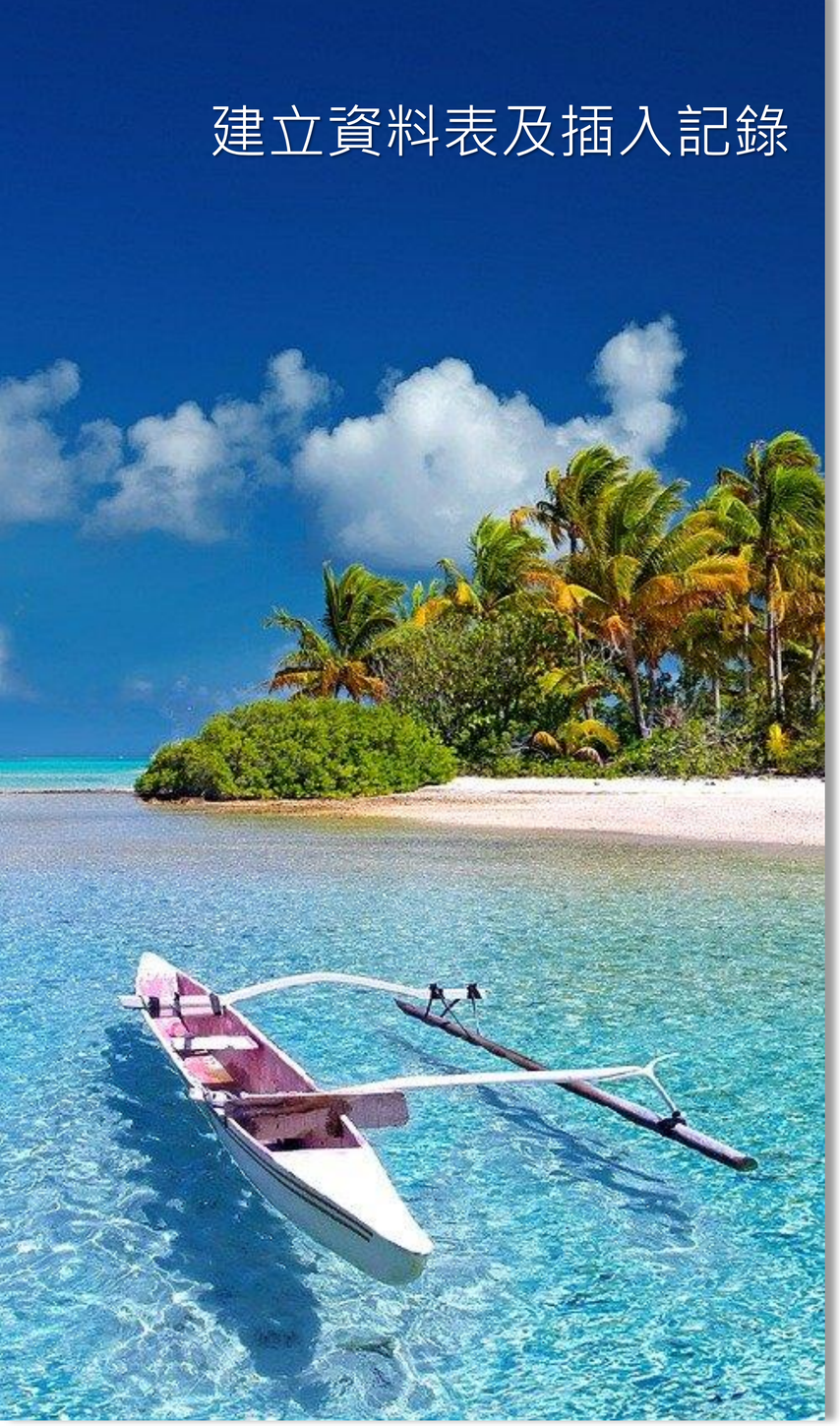
Question



Assuming the user credentials are correct,which expression will create a Connection?

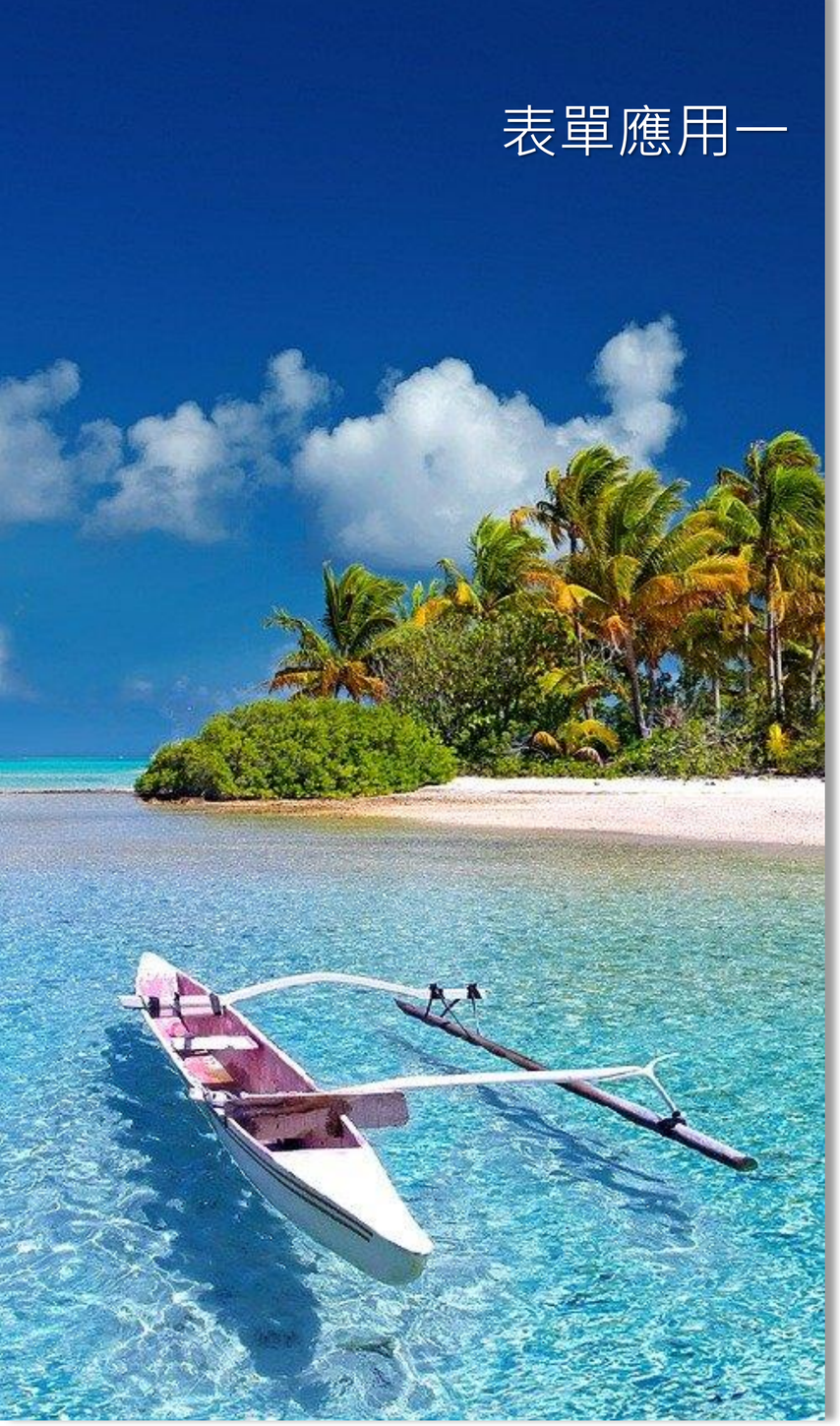
- A. DriverManager.getConnection("jdbc.derby.com");
- B. DriverManager.getConnection("http://database.jdbc.com","J_SMITH","dt12%2f3");
- C. DriverManager.getConnection("jdbc:derby:com");
- D. DriverManager.getConnection( );
- E. DriverManager.getConnection("J_SMITH","dt12%2f3");

建立資料表及插入記錄



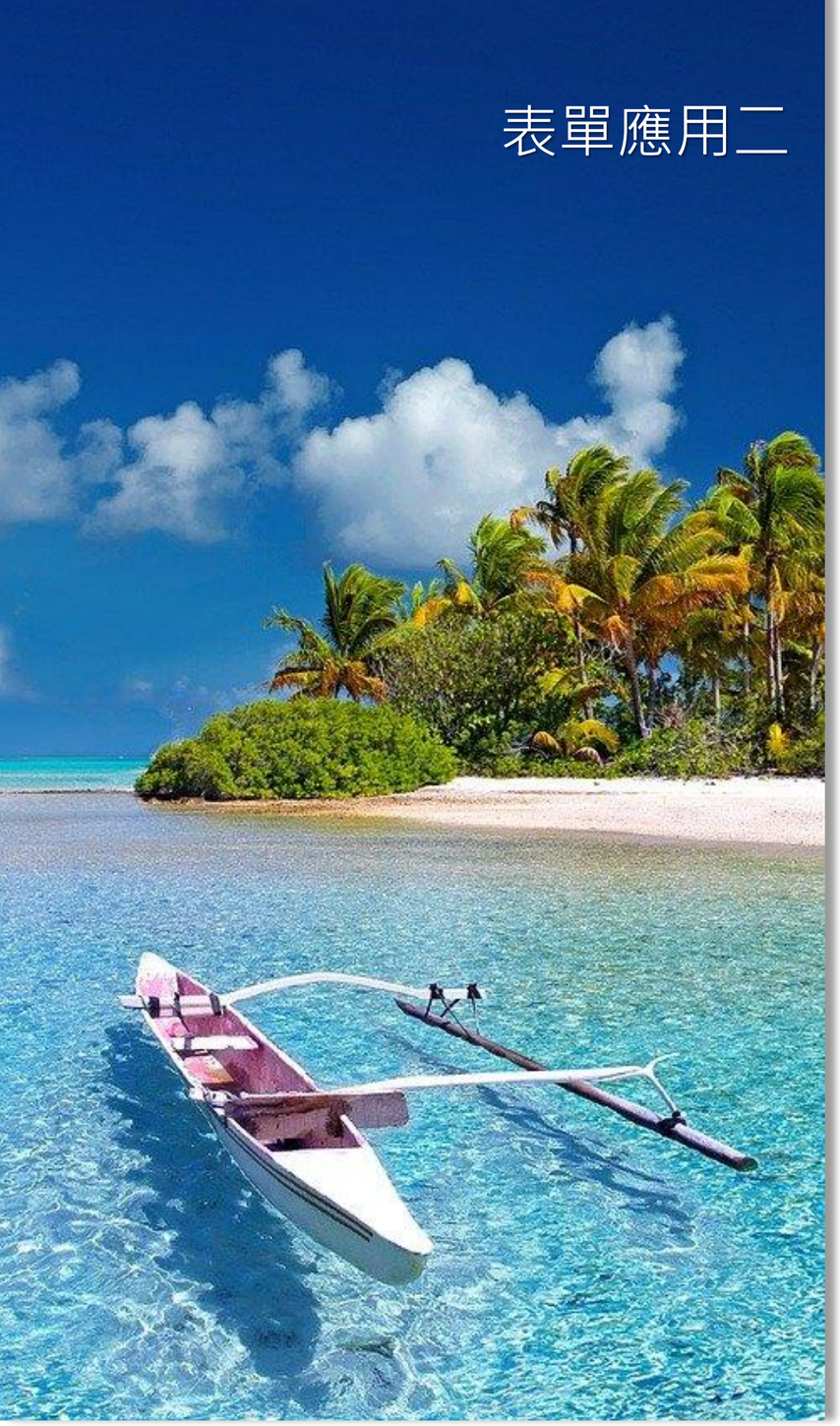
```
String url = "jdbc:mysql://localhost:3306/member?serverTimezone=Asia/Taipei&useSSL=false&useUnicode=true&characterEncoding=utf-8";
        String user = "root";
        String pw = "52680614";
        Connection conn = null;
        Statement st = null;
        try {
            conn = DriverManager.getConnection(url, user, pw);
            System.out.println("資料庫連結成功");
            st = conn.createStatement();
            //建立資料表
            String sql = "CREATE TABLE member.test(" + "ID INTEGER not null primary key," + "NAME VARCHAR(10)," + "GENDER VARCHAR(2)," +
"EMAIL VARCHAR(50)" + ")";
            st.executeUpdate(sql);
            //插入記錄
            String sql2 = "INSERT INTO member.test(id,name,gender,email)" + "VALUES(1,\'巨匠電腦\',\'先生\',\'pc@gmail.com\')";
            st.executeUpdate(sql2);
            //查看記錄
            String sql3 = "select * from test";
            ResultSet rs = st.executeQuery(sql3);
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                String gender = rs.getString("gender");
                String email = rs.getString("email");
                System.out.printf("%d\t%s\t%s\t%s\n", id, name, gender, email);


            }
        } catch (SQLException ex) {
            System.out.println(ex);
        }
```

表單應用一



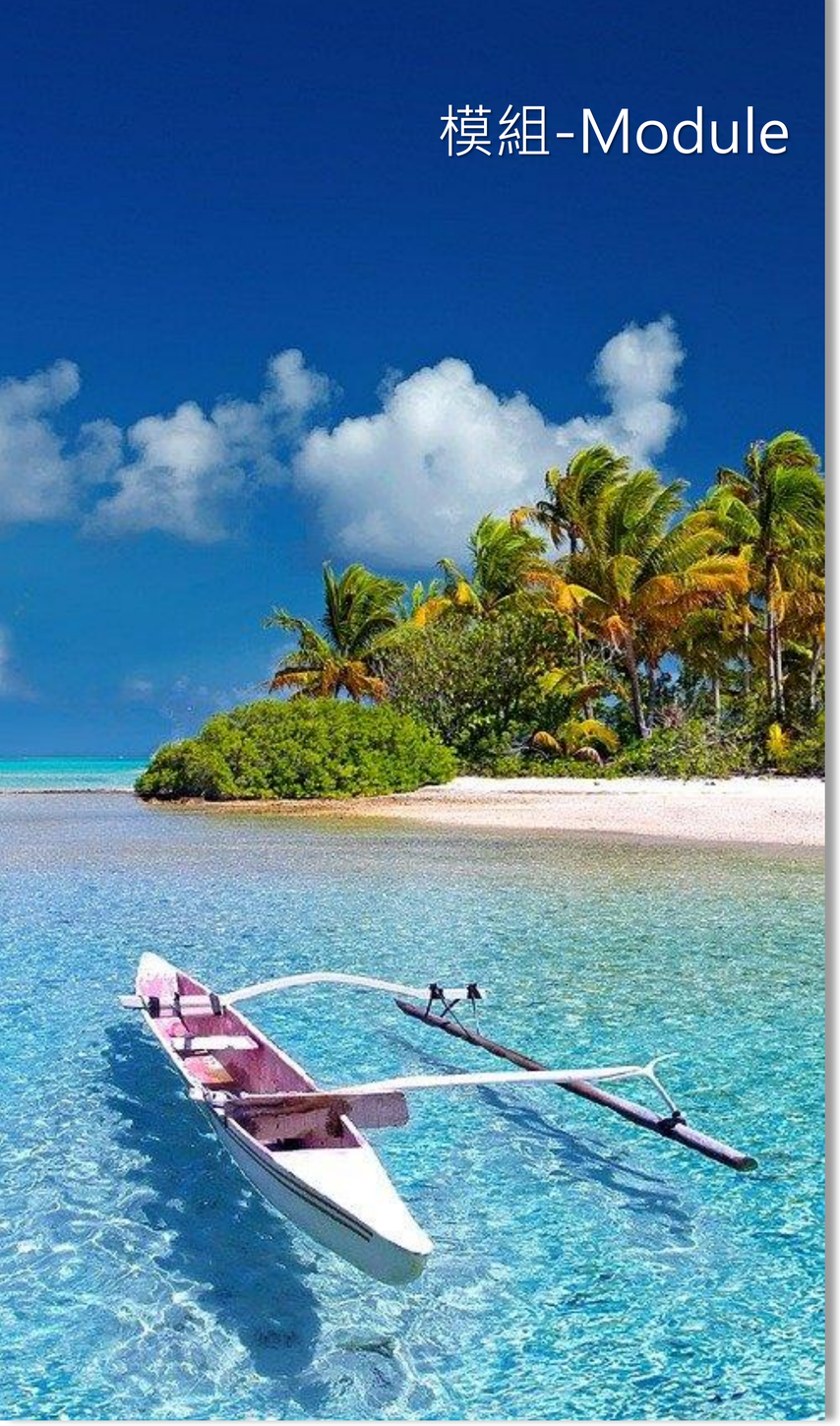
```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class MemberFrame extends JFrame {

   JPanel contentPane;
   JTextField textName, textID, textSex, textEmail;
   JButton buttonNext, buttonPrev, buttonEnd;

   public MemberFrame() {
      contentPane = (JPanel) this.getContentPane();
      contentPane.setLayout(null);
      textID = new JTextField("");
      textID.setBounds(75, 60, 90, 28);
      textName = new JTextField("");
      textName.setBounds(75, 90, 90, 28);
      textSex = new JTextField("");
      textSex.setBounds(75, 120, 90, 28);
      textEmail = new JTextField("");
      textEmail.setBounds(75, 150, 90, 28);
      buttonPrev = new JButton("上一筆");
      buttonPrev.setBounds(5, 190, 80, 35);
      buttonNext = new JButton("下一筆");
      buttonNext.setBounds(90, 190, 80, 35);
      buttonEnd = new JButton("結束");
      buttonEnd.setBounds(175, 190, 80, 35);
      contentPane.add(textID);
      contentPane.add(textName);
      contentPane.add(textSex);
      contentPane.add(textEmail);
      contentPane.add(buttonPrev);
      contentPane.add(buttonNext);
      contentPane.add(buttonEnd);
      this.setBounds(200, 100, 280, 300);
      this.setTitle("會員資料");
      this.setVisible(true);
   }

}
```

表單應用二



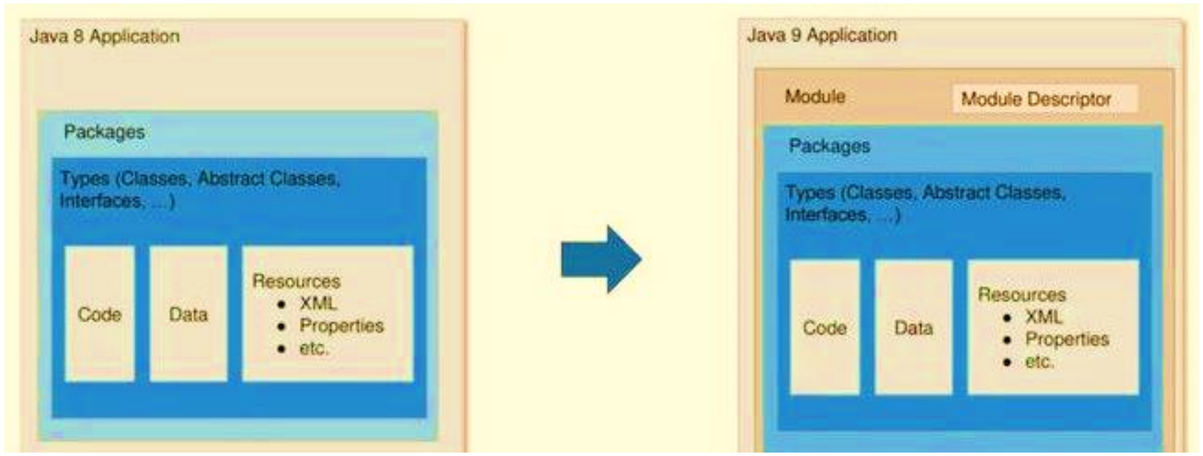
```
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;
public class DatabaseOk {
  public static void main(String[] args) throws Exception{
    MemberFrame member = new MemberFrame();
    Connection conn = null;
    Statement st = null;
    String url = "jdbc:mysql://localhost:3306/amashop?serverTimezone=Asia/Taipei&useSSL=false&useUnicode=true&characterEncoding=utf-8";
    String user = "root"; // 帳戶 root
    String password = "12345678"; // root 密碼
    try {

      conn = DriverManager.getConnection(url, user, password);

      System.out.println("成功連結 Bank_db 資料庫");
    } catch (Exception e) {
      System.out.println(e);
    }
    Statement stmt = conn.createStatement();

    String sql6 = "SELECT * FROM test.pcok";
    ResultSet rs = stmt.executeQuery(sql6);
    ResultSetMetaData rsm = rs.getMetaData();
    System.out.println("總欄數為:" + rsm.getColumnCount());
    if (rs.next()) {

      member.textID.setText(String.valueOf(rs.getInt("ID")));

      member.textName.setText(rs.getString("NAME"));
      member.textSex.setText(rs.getString("SEX"));

      member.textEmail.setText(rs.getString("EMAIL"));
    }

    member.buttonNext.addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        try {
          if (rs.next()) {

            member.textID.setText(String.valueOf(rs.getInt("ID")));

            member.textName.setText(rs.getString("NAME"));

            member.textSex.setText(rs.getString("SEX"));

            member.textEmail.setText(rs.getString("EMAIL"));
          }
        } catch (SQLException ea) {
        };
      }
    });
    member.buttonPrev.addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        try {
          if (rs.previous()) {

            member.textID.setText(String.valueOf(rs.getInt("ID")));

            member.textName.setText(rs.getString("NAME"));

            member.textSex.setText(rs.getString("SEX"));

            member.textEmail.setText(rs.getString("EMAIL"));
          }
        } catch (SQLException ea) {
        };
      }
    });
    member.buttonEnd.addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent e) {
        System.exit(0);
      }
    });
  }
}
```

模組 Module



模組-Module

- JDK 8之前 問題
  - JAVA 運行環境的膨脹和過於肥胖,每次Jvm運行時需加載rj.jar,不管有無運行到全部會加載上去
  - 每一個public類別都可以被其它類別訪問到,這樣會導致無意中使用了並不想被公開的 api
- JDK 9之後 模組化解決問題
  - 所以模組就是用來管理各個package,通過設定package的顯示與隱藏,設定誰可存取,所以模組使得程式更安全
  - 可看成一個電商系統,可區分為 支付模組 購物車模組 會員模組 商品模組.....
- java –list-modules



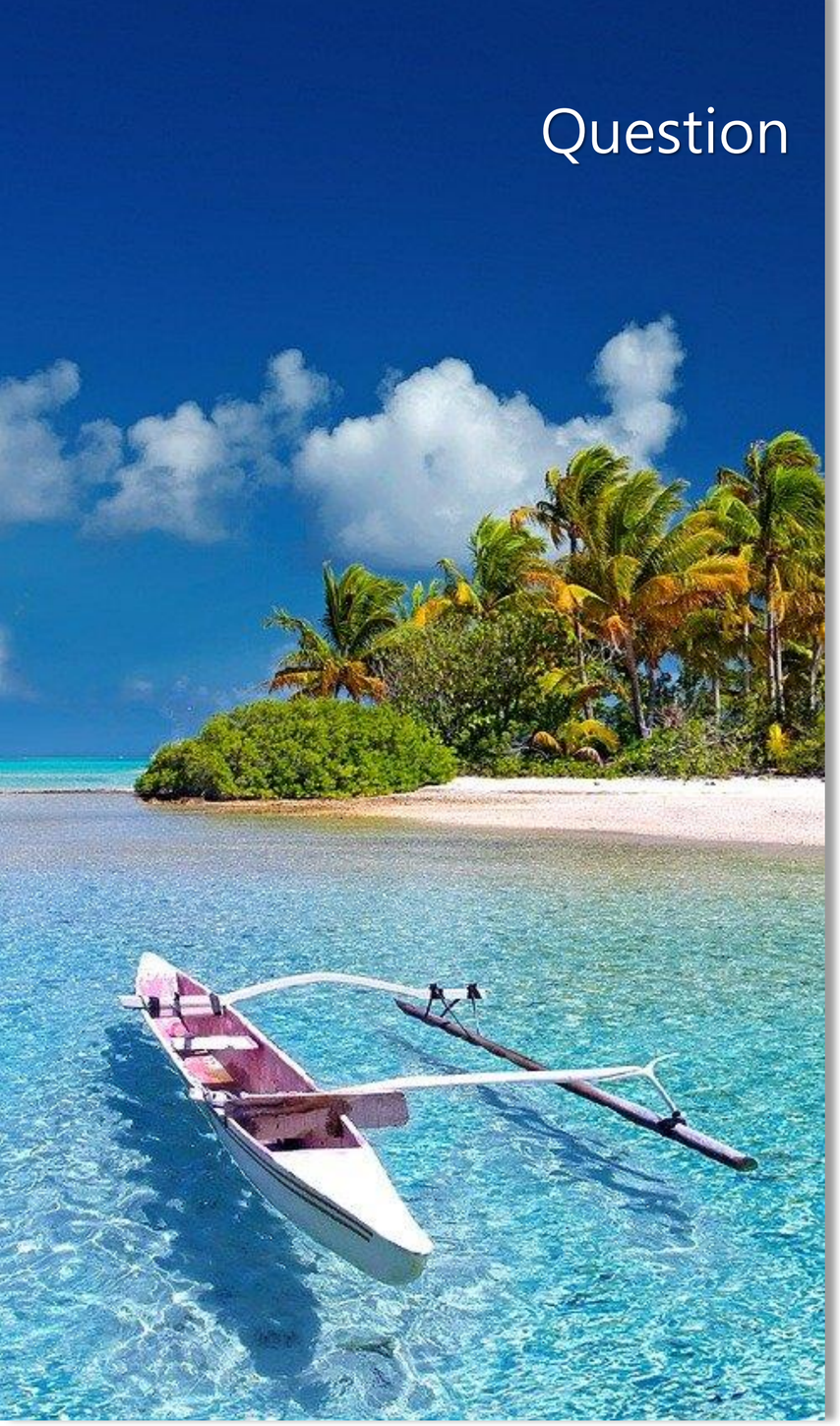
Accessibility (JDK 1 – JDK 8)

- public
- protected
- package
- private

Accessibility (JDK 9)

- *public to everyone*
- *public but only to friend modules*
- *public only within a module*
- protected
- package
- private

Question



Which two modules include APIs in the Java SE Specification?

(Choose two.)

A. java.logging

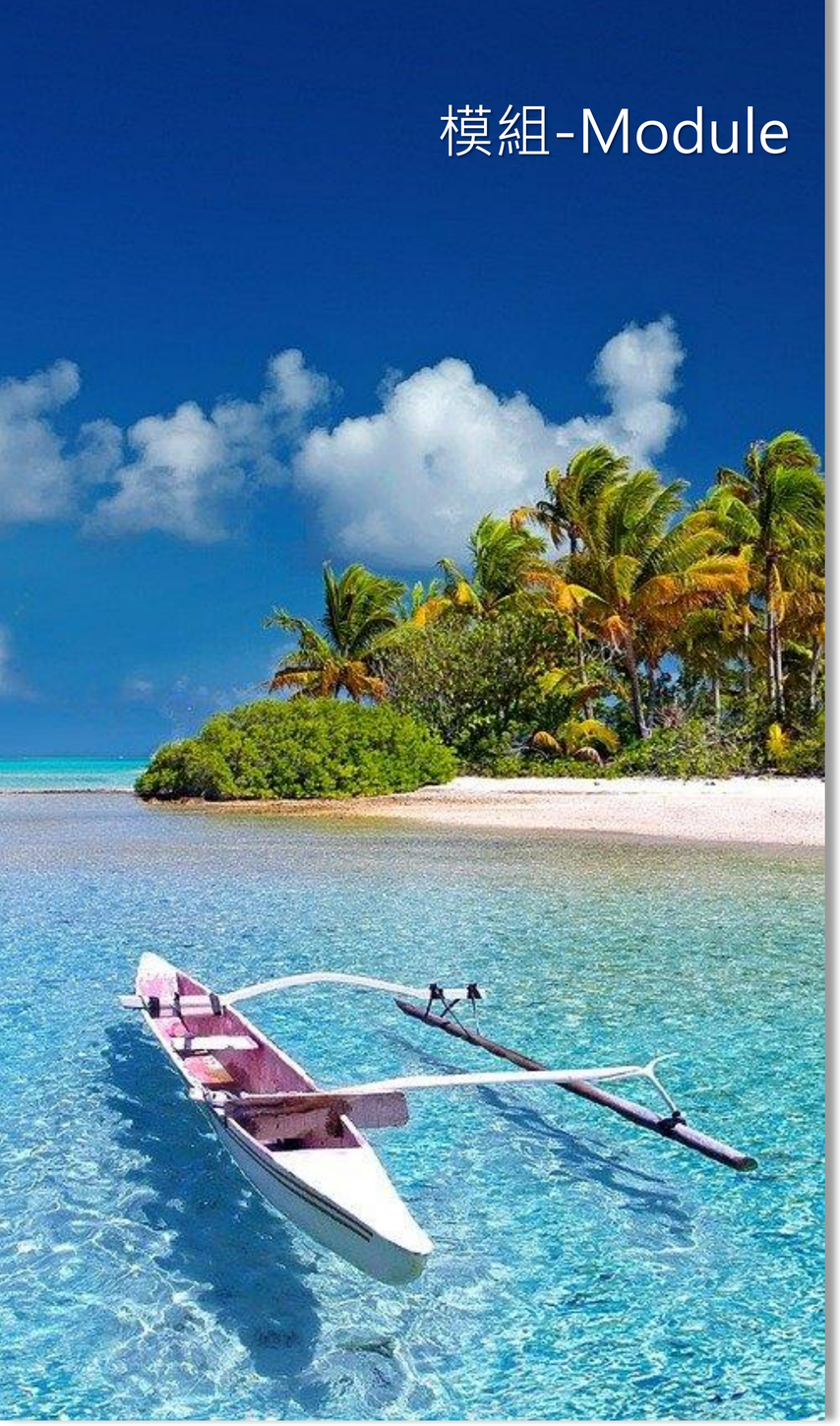
B. java.desktop

C. javafx

D. jdk.httpserver

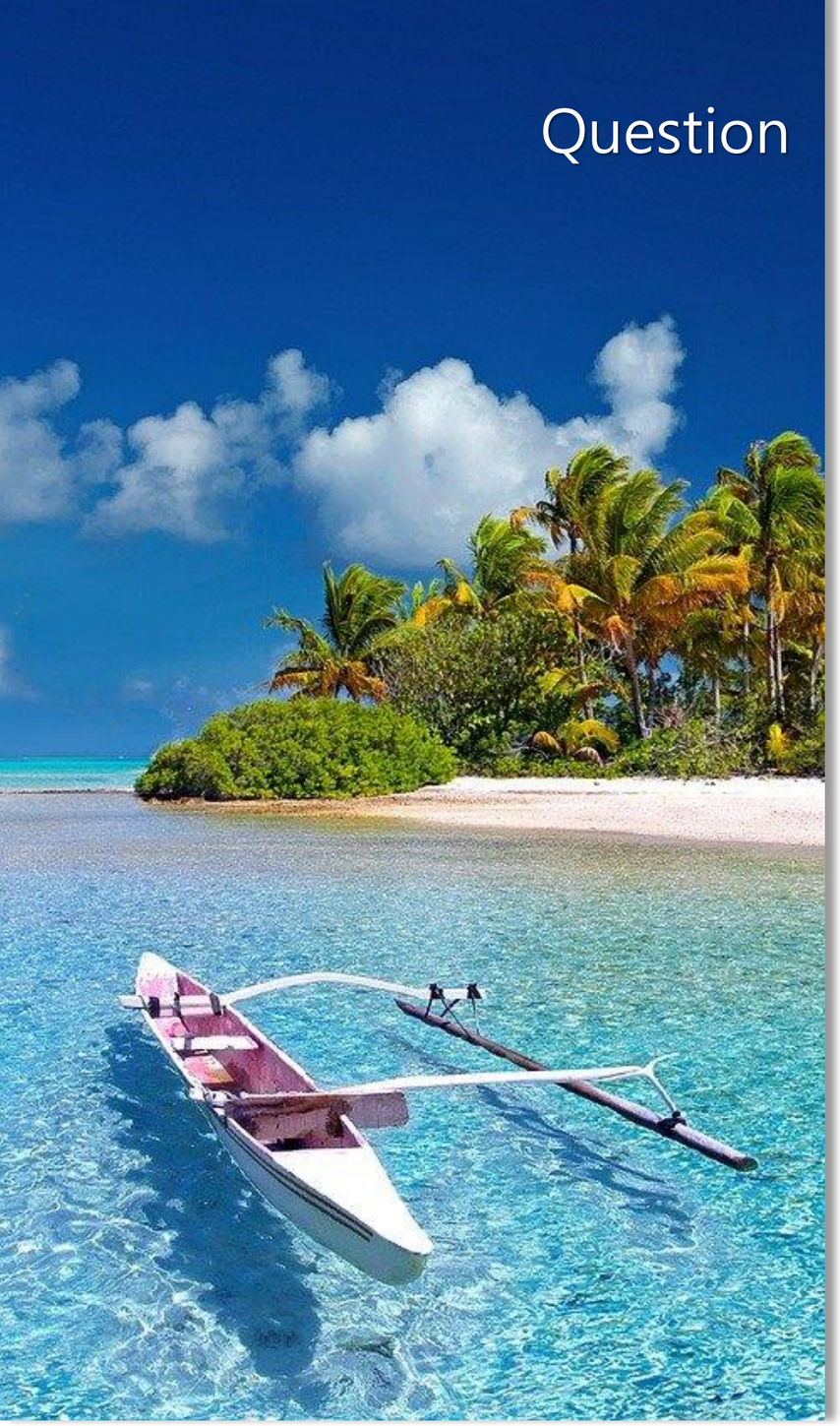
E. jdk.jartool

模組-Module



- 一個模組必須提供一個模組描述器 — 描述模組相依性、該模組能讓其他模組使用的套件等中介資料。模組描述器是一個經過編譯的模組宣告，定義於 module-info.java 檔案中，每個模組宣告從關鍵字 module 開始，緊接著一個獨一無二的模組名稱，以及括在大括弧內的模組內容：
- module modulename {
- }
- 模組宣告的主體可以是空的，或是包含各種模組指令 (directive)，像是 requires、 exports、 provides...with、 uses 及 opens ，編譯模組宣告會建立模組描述器，存放在模組根目錄的 module-info.class 檔案中，

Question



Which two statements are correct about modules in Java? (Choose two.)

A. java.base exports all of the Java platforms core packages.

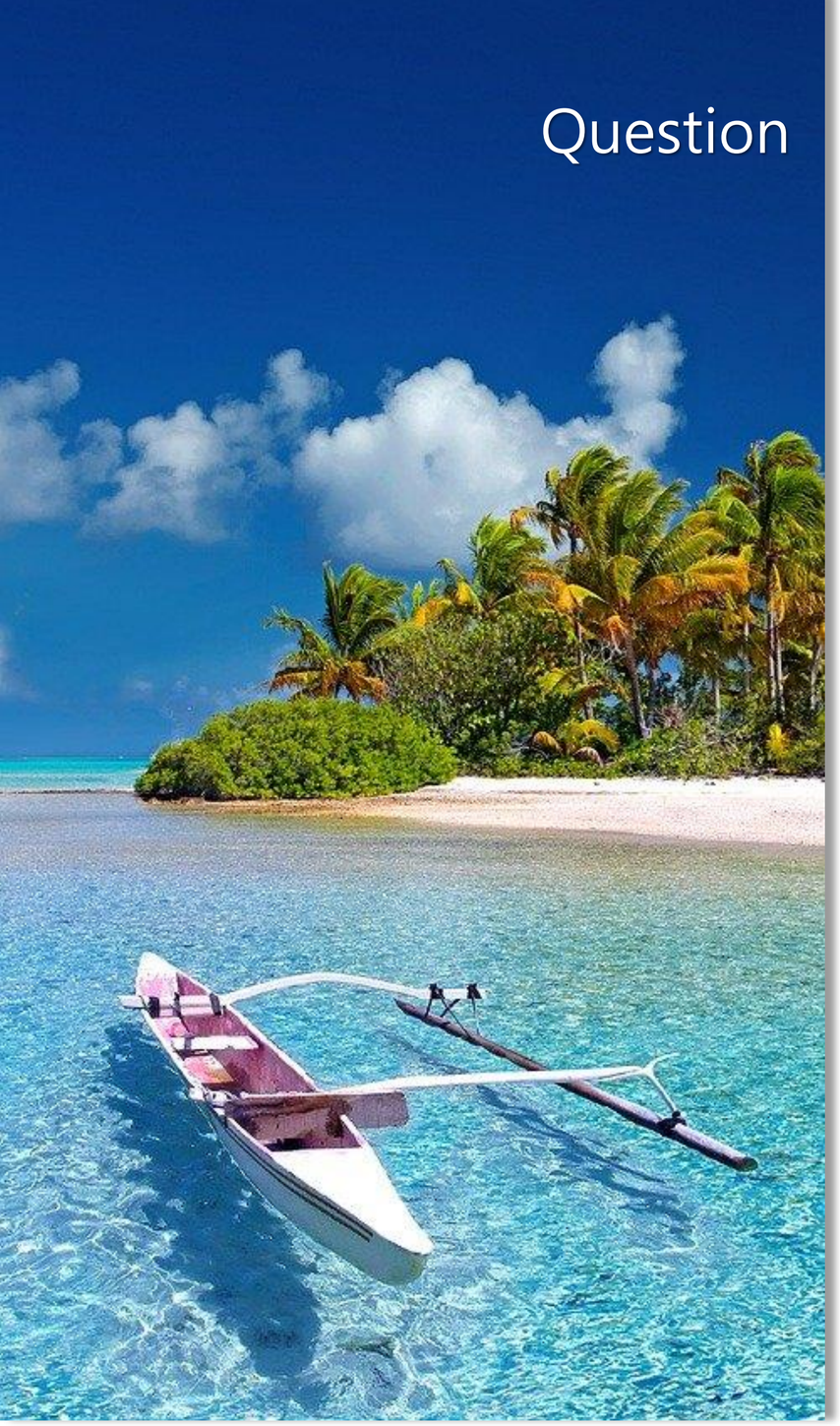
B. module-info.java can be placed in any folder inside module-path.

C. A module must be declared in module-info.java file.

D. module-info.java cannot be empty.

E. By default, modules can access each other as long as they run in the same folder.

Question



Which two describe reasons to modularize the JDK? (Choose two.)

A. easier to understand the Java language

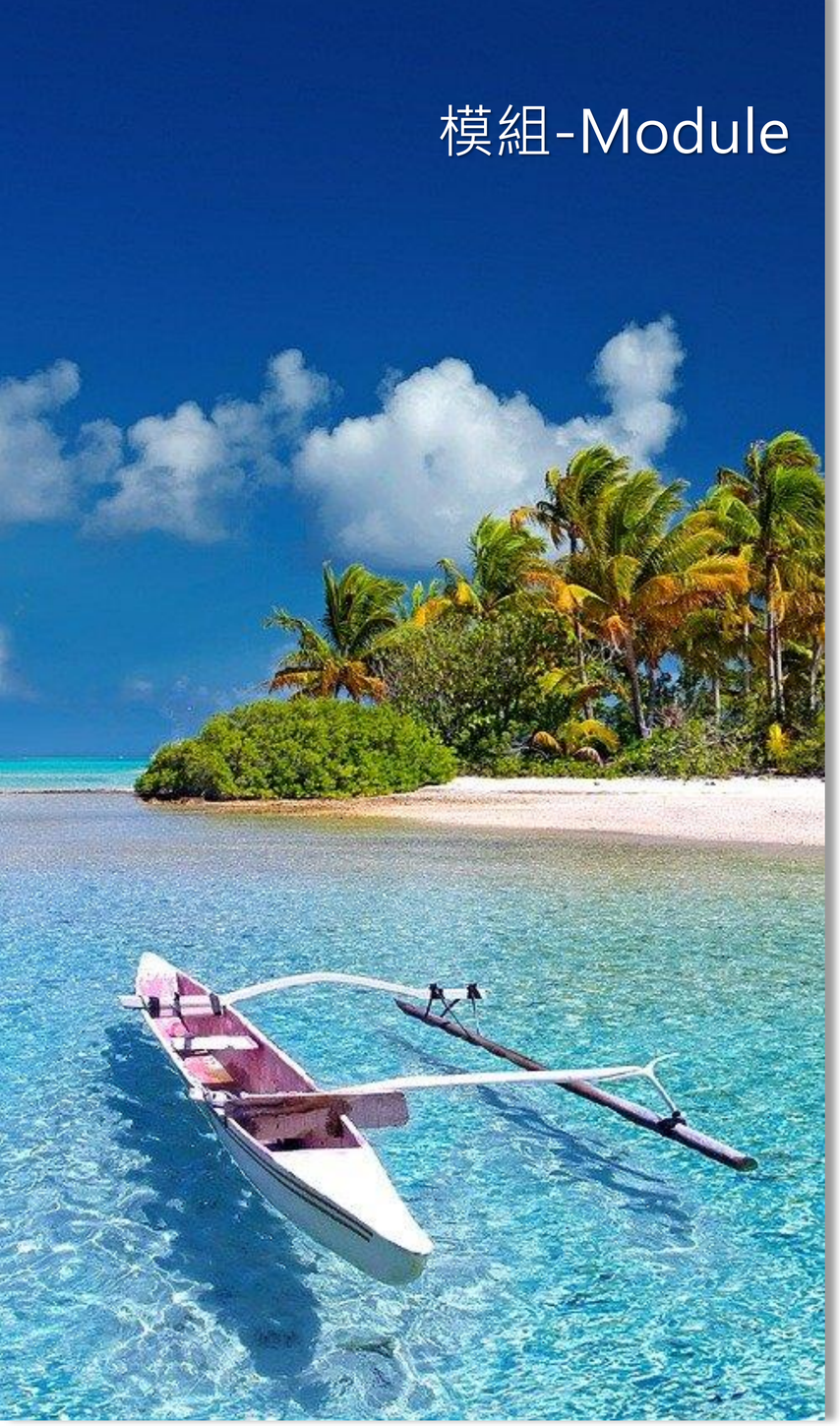
B. improves security and maintainability

C. easier to expose implementation details

D. improves application robustness

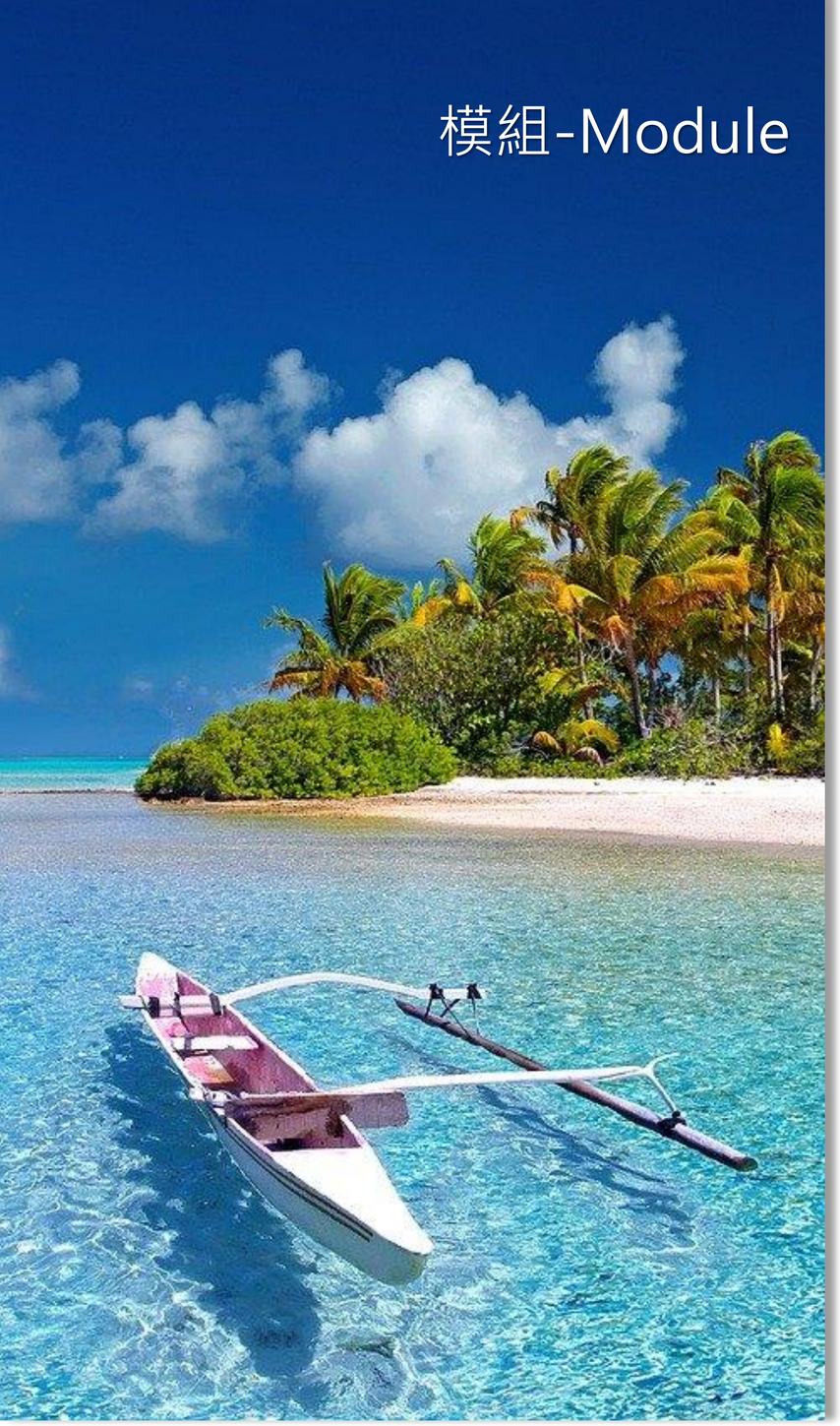
E. easier to build a custom runtime linking application modules and JDK modules

模組-Module



- requires。 require 模組指令指定該模組依賴其他模組 — 這關係稱作模組相依性，每個模組必須明確聲明它的相依性，當模組 A 需要模組 B，模組 A 稱為讀取模組 B，模組 B 則是給模組 A 讀取。要指定與另一個模組的相依性，使用 requires
- requires transitive — 隱含的可讀性，指定相依另一個模組，並確保其他模組讀取您的模組時亦能讀取這相依模組
- java.desktop 模組宣告中的下列指令：
- requires transitive java.xml;
- 在這情況下，任何讀取 java.desktop 的模組也隱含地能讀取 java.xml。例如：一個 java.desktop 模組中的函式還傳一個 java.xml 模組中的型別，那讀取 java.desktop 模組的程式變成依賴 java.xml。若 java.desktop 的模組宣告中沒有 requires transitive 指令，相依的模組則無法編直到牠們明確讀取 java.xml。

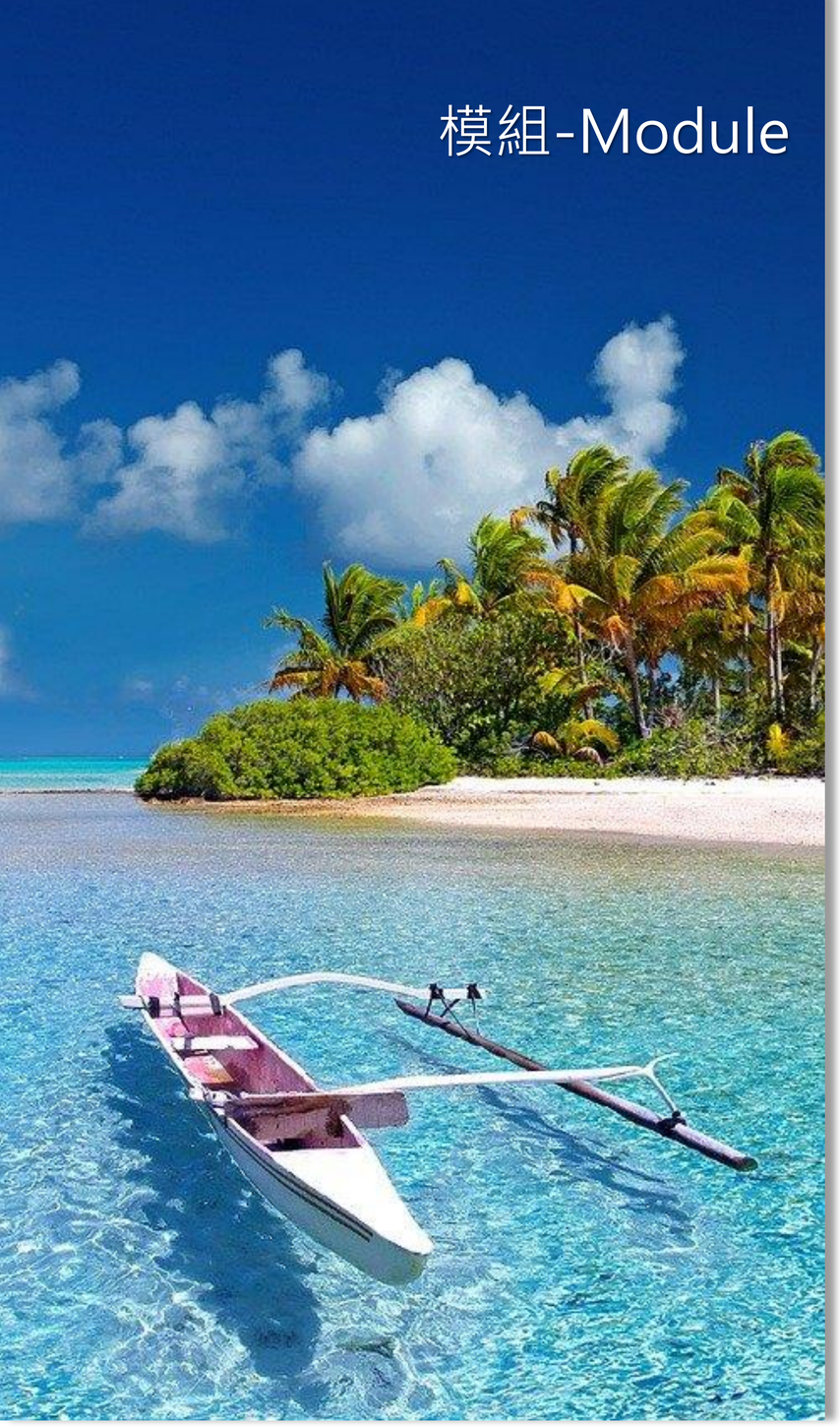
模組-Module



- requires。 require 模組指令指定該模組依賴其他模組 — 這關係稱作模組相依性，每個模組必須明確聲明它的相依性，當模組 A 需要模組 B，模組 A 稱為讀取模組 B，模組 B 則是給模組 A 讀取。要指定與另一個模組的相依性，使用 requires
- requires transitive — 隱含的可讀性，指定相依另一個模組，並確保其他模組讀取您的模組時亦能讀取這相依模組
- java.desktop 模組宣告中的下列指令：
- requires transitive java.xml;
- 在這情況下，任何讀取 java.desktop 的模組也隱含地能讀取 java.xml。例如：一個 java.desktop 模組中的函式還傳一個 java.xml 模組中的型別，那讀取 java.desktop 模組的程式變成依賴 java.xml。若 java.desktop 的模組宣告中沒有 requires transitive 指令，相依的模組則無法編直到牠們明確讀取 java.xml。
- exports 及 exports…to。 exports 指令指定模組中的某套件的 public 型別 (及其內部的 public 及 protected 型別) 可以被所有其他模組存取，而exports…to 指令能讓您以逗號分隔的列表精確地指定哪些模組能存取匯出的套件 — 所謂有限制的匯出。
- uses。uses 模組指令指定該模組使用的服務 — 讓此模組成為一個服務使用者。這個服務通常是一個介面或抽象類。它不應該是一個實現類。

- provides…with。provides…with 模組指令指定模組提供服務的實作 — 讓該模組成為一個服務提供者。指令中 provides 的部分指定列在 uses 指令中的介面或抽象類別，with 的部分指定實作介面或繼承抽象類別以提供服務的類別名稱。

模組-Module



- 模組打包程序
- 1.打包專案為.jar檔
  - a,建立一個普通的.jar檔,及一個指定為運行主類的.jar檔
- jar --create --file sales.jar -C com.taipei . (.為當前目錄)
- 完成後會有一個sales.jar的檔案
  - b.再建立一個主執行jar檔
  - jar --create --file mainclass.jar --main-class com.taichung.english.MainClass -C com.taichung .
- 2.打包.jar為 .jmod檔
  - jmod create --class-path sales.jar com.taipei.jmod
- 3.根據.jmod建立一個新的jre環境,只包含需要的模組
  - jlink --module-path mod --add-modules java.base,com.taipei,com.taichung --output amashop
- 4.進入bin資料夾,進入命令提示字元模式
- 5.輸入 java –module com.taichung即可執行