

Java 與 UML

目錄

章節目錄

1. UML 概論	1
Uniform Modeling Language	1
行爲圖型	2
使用案例圖型	2
架構圖型	4
類別圖型	4
物件圖型	7
元件圖型	8
佈署圖型	8
動態圖型	9
循序圖型	10
合作圖型	11
狀態圖型	11
活動圖型	12
2. 類別圖型	13
類別節點	13
樣式	13
屬性區格	19
方法區格	21
結合關係	25
繼承 (extends)	25
實作 (implements)	27
結合	28
聚合	32
組合	33
巢狀類別	35
結合類別	37
限定結合	39
3. 套件圖型	41

套件標記	41
相依性	44
4. 物件圖型	47
物件節點	50
樣式	50
屬性區格	52
結合關係	53
實作	53
結合	54
類別屬性	56
5. 元件圖型	57
元件	58
元件與介面	59
使用元件圖型	61
6. 佈署圖型	63
節點	64
關聯	65
7. 循序圖型	67
元素	69
物件節點	69
生命線與活化區塊	72
訊息	72
內部訊息	75
解構物件	76
迴圈	77
多執行緒	78
8. 合作圖型	81
元素	82
物件節點	82
連結	83
訊息	83
物件狀態的改變	86
解構物件	87
迴圈	88

多執行緒	89
9. 狀態圖型	91
狀態節點	92
名稱區格	93
內部轉換區格	93
轉換	94
子狀態	96
10. 活動圖型	97
圖型元素	98
狀態與活動	98
轉換	99
分支	100
水道	101
分岐與結合	102

1. UML 概論

Uniform Modeling Language

「統一塑模語言、Uniform Modeling Language」，簡稱「UML」。它是一種使用圖型化的表示方式，用來表示軟體系統的圖型。「UML」不是一種程式語言，經由一組相關的圖型來規範與架構軟體系統，它是目前分析與設計物件導向軟體系統常用的一種工具。

UML 的規格在 90 年代早期由 Grady Booch, James Rumbaugh 和 Ivar Jacobson 制定，目前由 Object Management Group、OMG 負責規格的維護，UML 相關的規格，都可以在 OMG 的網站中取得：
<http://www.omg.org/uml>。

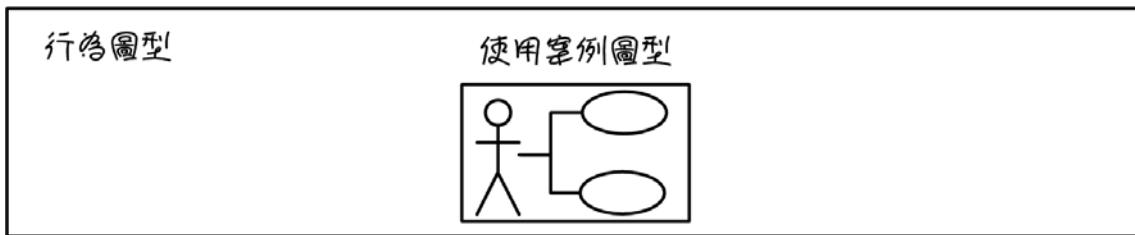
UML 規格中制定了九種主要的圖型(Diagram)：

- ❑ 使用案例圖型、Use Case Diagrams
- ❑ 類別圖型、Class Diagrams
- ❑ 物件圖型、Object Diagrams
- ❑ 元件圖型、Component Diagrams
- ❑ 佈署圖型、Deployment Diagrams
- ❑ 循序圖型、Sequence Diagrams
- ❑ 合作圖型、Collaboration Diagrams
- ❑ 狀態圖型、Statechart Diagrams
- ❑ 活動圖型、Activity Diagrams

以層次來看 UML 圖型，是以 UML 圖型的內容為觀察角度；以軟體系統架構的角度來看 UML 圖型，可以把它們分為「行為圖型」、「架構圖型」和「動態圖型」。本書的章節以這樣的分類來討論這些圖型。

行為圖型

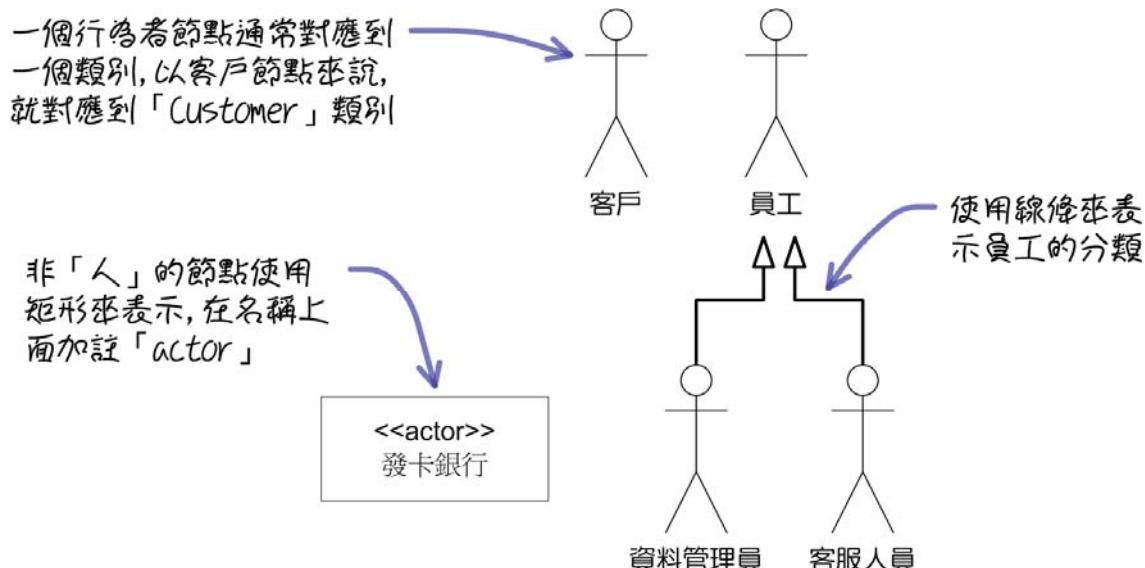
行為圖型用來表示軟體系統組成的基本單位和彼此之間的關聯，是屬於概念層次的圖型，在這個角度下只有一個「使用案例圖型」：



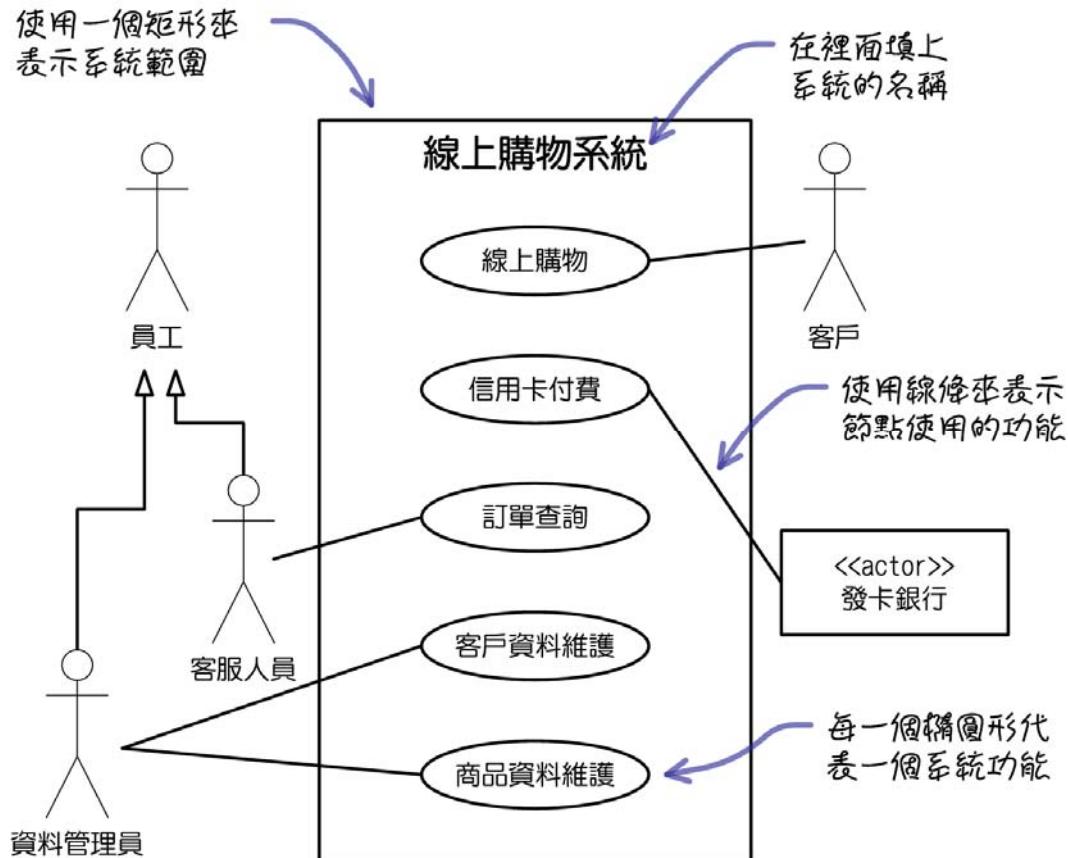
使用案例圖型

「使用案例圖型、Use Case Diagrams」是概念層次的產物，它使用簡單的圖型來表示軟體系統的基本單位，例如使用者、裝置和系統的功能；圖型中也會標示出基本單位之間的關聯，例如使用者操作的系統功能，和系統功能之間的關係。

使用案例圖型使用一個人型來表示軟體系統的使用者，稱為「行爲者節點、Actor Node」，並且在圖型下方標示行爲者節點的類別。行爲者節點之間也存在著類似繼承的關係，如果使用口語來說的話，可以把它們當成分類的觀點來看：



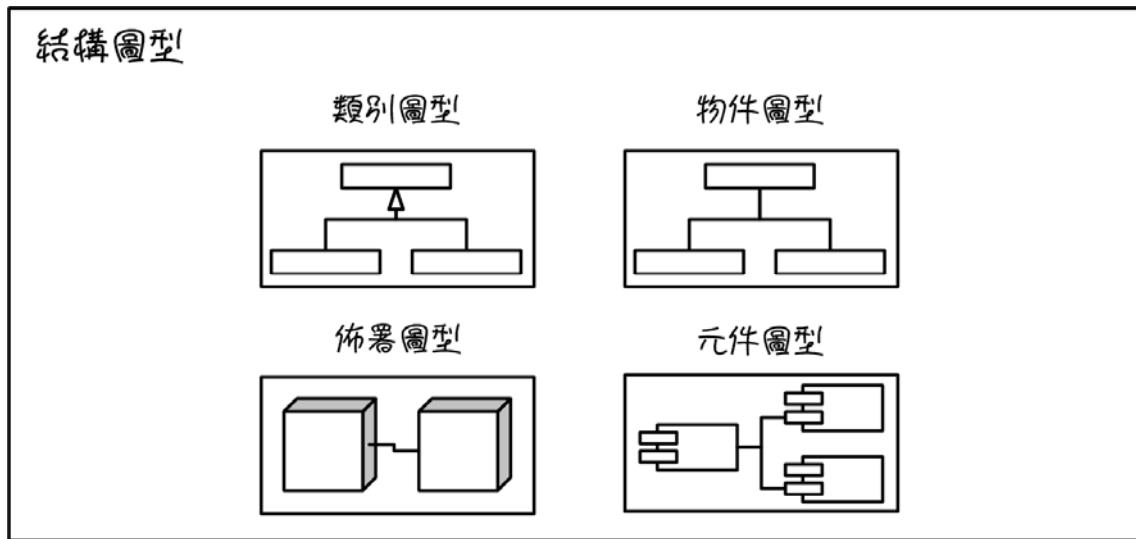
在確定使用者節點之後，接下來就要加入系統的定義，還有標示系統與使用者節點之間的關聯：



使用者圖型還有許多在細節上的表現，並不在本書的討論範圍，如果需要關於使用者圖型詳細的資訊，請參考「UML Specification、3.6 Use Case Diagrams」的說明。

架構圖型

架構圖型用來表示軟體系統中「不變」的結構，以及元素之間的關聯，它們是屬於規格層刺的圖型：



類別圖型

下列是一個使用「`java.util.List`」來模擬「`Map`」資料結構的類別「`MyMap`」。接下來以這個簡單的模擬程式來討論「類別圖型、Class Diagrams」：

```

import java.util.List;
import java.util.ArrayList;

public class MyMap {
    private List keyList = new ArrayList();
    private List valueList = new ArrayList();

    public void add(Object key, Object value) {
        if (keyList.size() == 0 || !keyList.contains(key)) {
            keyList.add(key);
            valueList.add(value);
        }
    }
}

```

```
        }

    }

public Object get(Object key) {
    Object result;
    int index = 0;

    if ((index = keyList.indexOf(key)) != -1) {
        result = valueList.get(index);
    } else {
        result = null;
    }

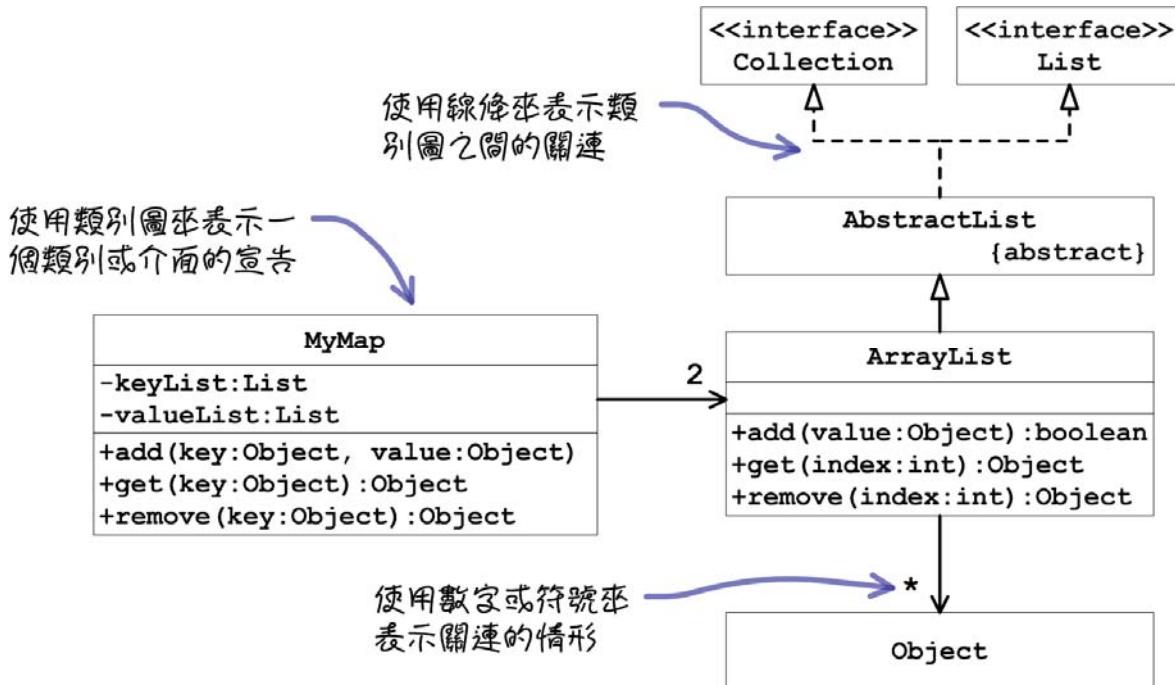
    return result;
}

public Object remove(Object key) {
    Object result;
    int index = 0;

    if ((index = keyList.indexOf(key)) != -1) {
        keyList.remove(index);
        result = valueList.remove(index);
    } else {
        result = null;
    }

    return result;
}
}
```

上列的程式碼使用兩個「`ArrayList`」物件來儲存鍵值(`keyList`)和元素值(`valueList`)，鍵值和元素值的型態都宣告為「`Object`」。以類別圖型來表示這個程式的話，看起來會像這樣：



從類別圖型中，你可以清楚的看出一個軟體系統中全部或局部的類別架構，和類別之間的關聯。

物件圖型

同樣以在討論類別圖型時的「MyMap」類別來說明物件圖型。在類別圖型中，你只能知道類別還有類別彼此之間關聯的架構，「物件圖型、Object Diagrams」用來補充類別圖型，它可以呈現軟體系統「某一個時間點」的物件和物件彼此之間關聯的架構。一般會把它稱為「記憶體的快照、memory snapshot」：

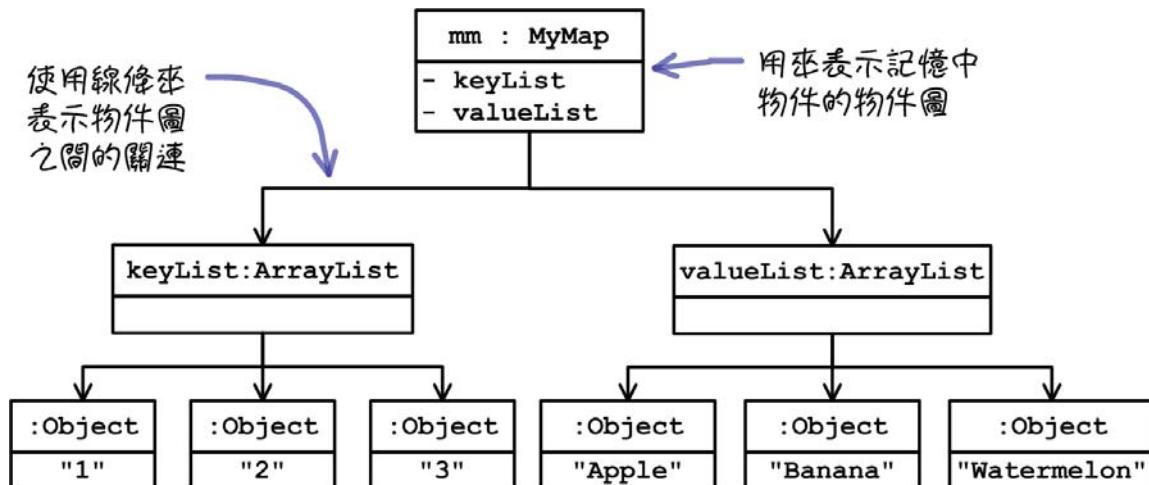
```
public class TestMyMap {
    public static void main(String args[]) {
        MyMap mm = new MyMap();

        mm.add("1", "Apple");
        mm.add("2", "Banana");
        mm.add("3", "Watermelon");

        ...
    }
}
```

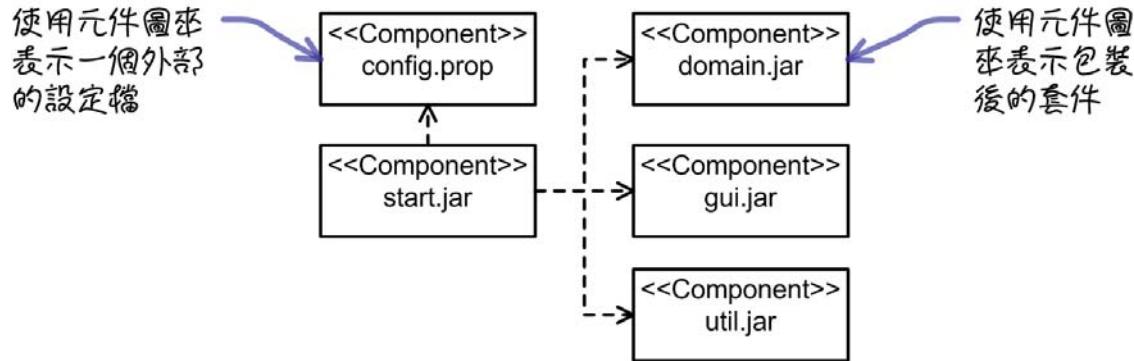
← 程式流程進行到這裡時的物件狀態

在上列的敘述中，使用「MyMap」類別加入了三個對照資料元素，這個時候可以使用物件圖型來表示目前的物件狀態：



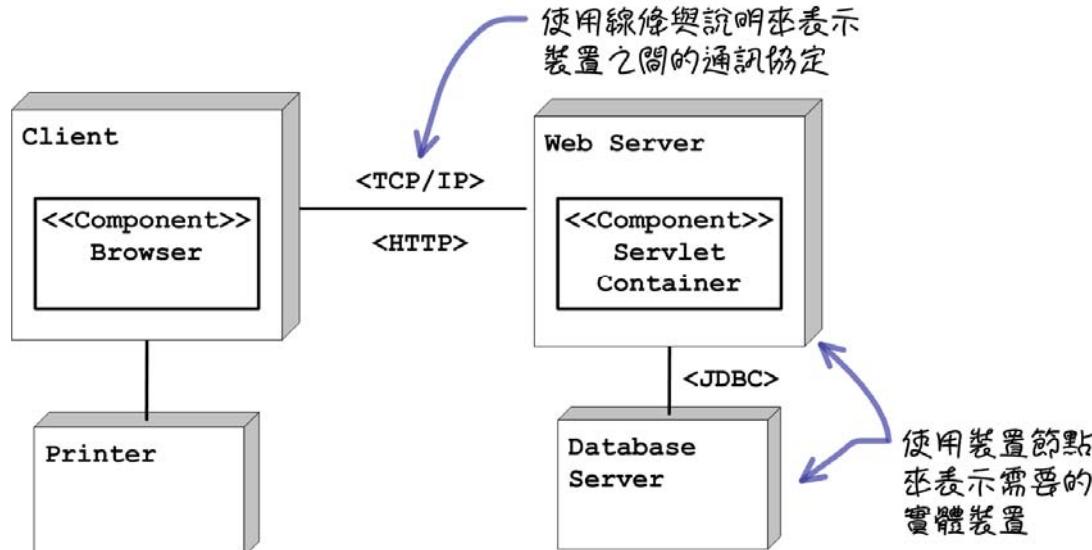
元件圖型

在「元件圖型、Component Diagrams」中，一個包裝過後的 Java 可執行檔、Java 原始程式碼，或是一個套件的包裝檔，都可以當成元件。元件圖型可以隱藏部份細節，更容易表現出軟體系統的架構：



佈署圖型

「佈署圖型、Deployment Diagrams」用來規劃軟體系統開發完成後，提供系統管理人員將元件安裝在實體裝置的圖型。佈署圖型中也可以包含所有軟體系統需要的實體裝置和裝置裡需要的元件，還有使用的通訊協定：

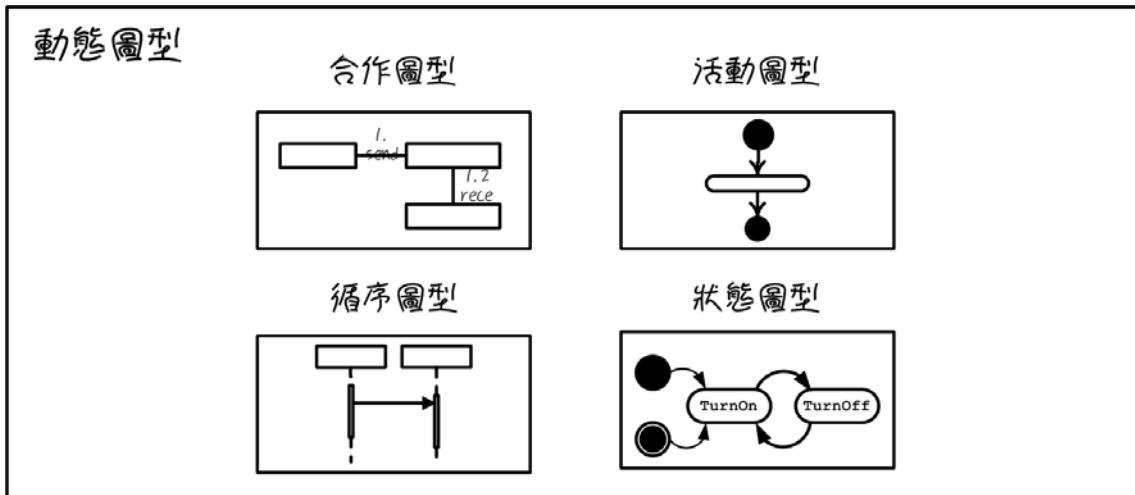


動態圖型

動態圖型用來表示軟體系統中元素之間的互動與合作方式，它們是屬於實作層次的圖型：

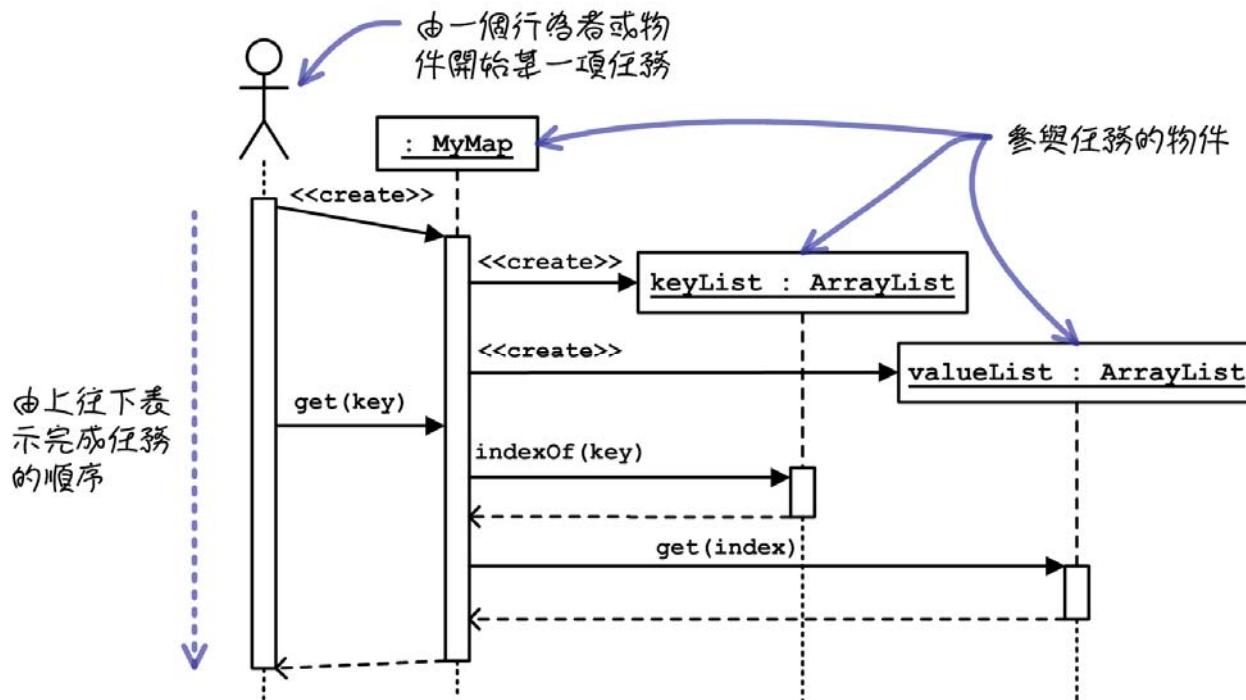
循序圖型

「循序圖型、Sequence Diagrams」是互動圖型的一種，它以「時間順序」和「合作物件」為主要的概念，以時間的先後順序來呈現物件之間的合作關係。循序圖型與「合作圖型、Collaboration Diagrams」所顯示的內容同樣都是一組合作物件之間的互動，也都具有順序性，但是循序圖型應該以呈現軟體系統中的「任務」為考量：



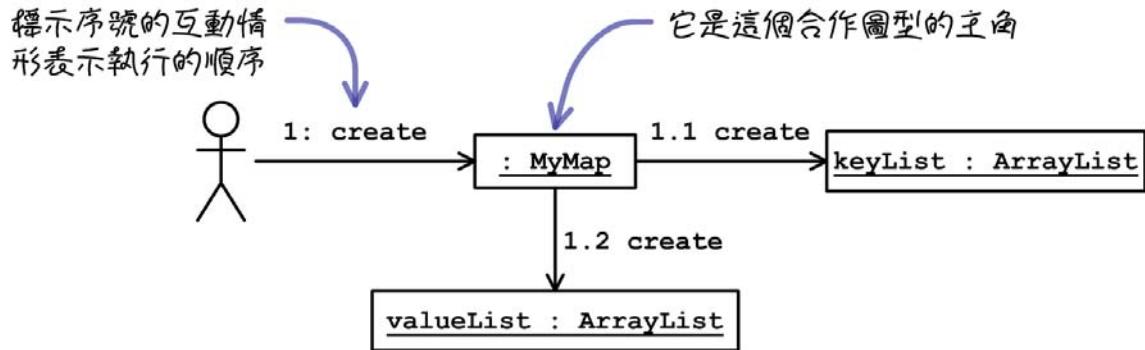
循序圖型

「循序圖型、Sequence Diagrams」是互動圖型的一種，它以「時間順序」和「合作物件」為主要的概念，以時間的先後順序來呈現物件之間的合作關係。循序圖型與「合作圖型、Collaboration Diagrams」所顯示的內容同樣都是一組合作物件之間的互動，也都具有順序性，但是循序圖型應該以呈現軟體系統中的「任務」為考量：



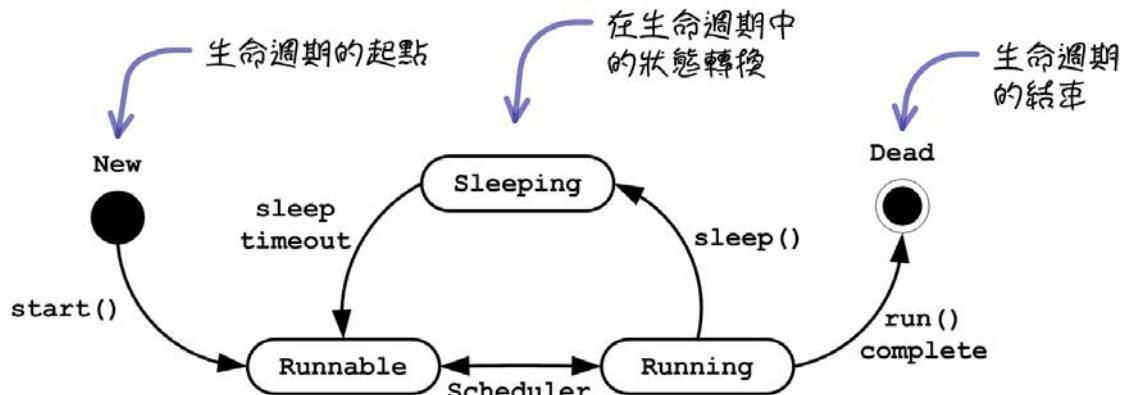
合作圖型

「合作圖型、Collaboration Diagrams」與上列討論的循序圖型同樣是互動圖型，所顯示的資訊也很類似，同樣是一組合作物件之間，具有順序性的互動，不過合作圖型的角度是以「關鍵物件」為考量：



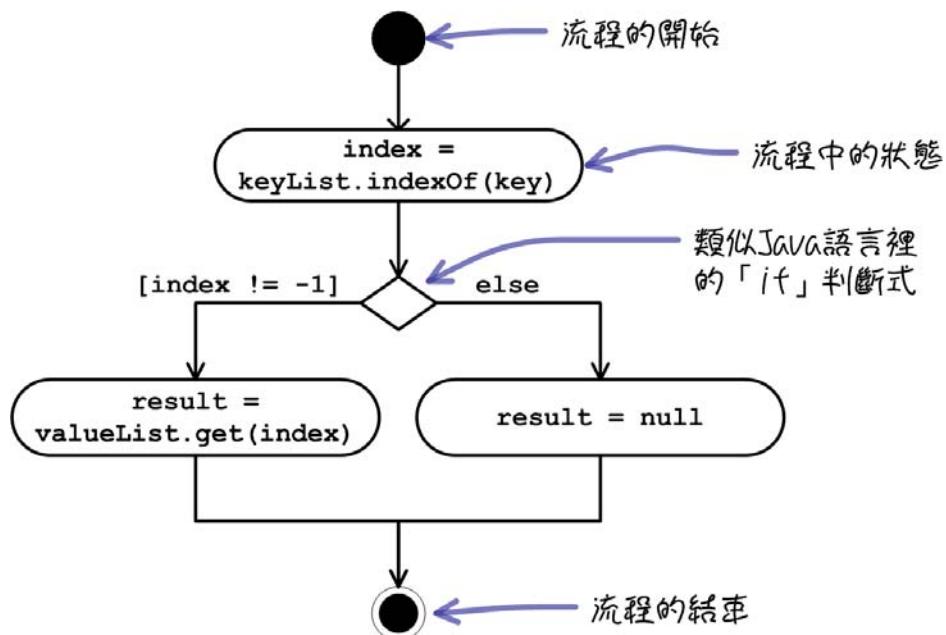
狀態圖型

「狀態圖型、Statechart Diagrams」顯示的資訊以「生命週期」為主要的概念，它呈現在軟體系統中的關鍵物件或一組物件，從開始到結束的所有狀態和轉換狀態的條件，顯示這個生命周期的所有資訊：



活動圖型

「活動圖型、Activity Diagrams」顯示以任務為主的活動流程，在活動圖型裡，並不會加入架構圖型的元素，例如「類別節點、Class Node」，它會以類似傳統的流程圖方式來呈現軟體系統的流程；不過在有些時候，為了表現流程之間的物件輸出與輸入，也可以加入「物件節點、Object Node」：



2. 類別圖型

「類別圖型、Class Diagrams」使用圖型的表示法來顯示軟體系統中類別的內容，還有類別和類別之間的關聯，它呈現的是一個靜態的類別架構。

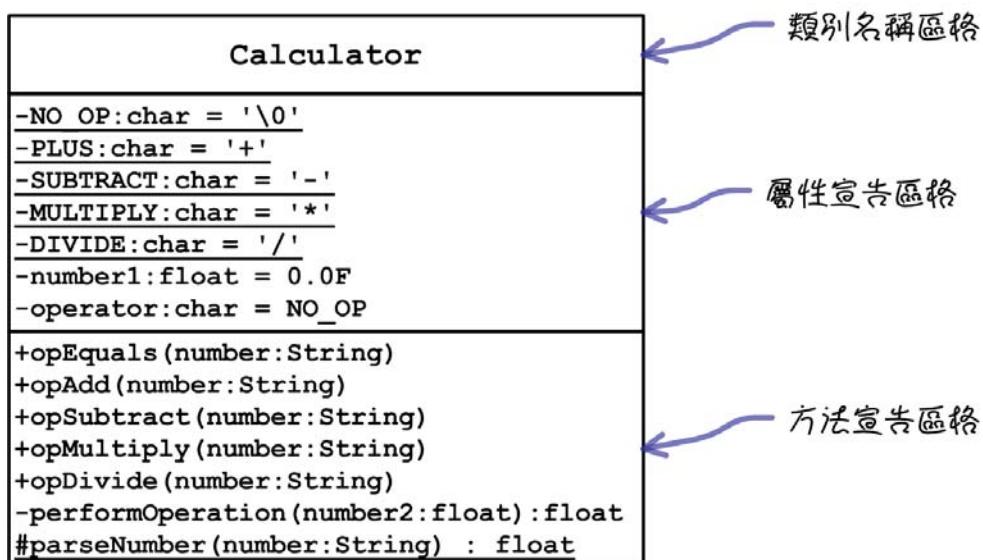
類別圖中的類別節點可以把每一個類別所有的內容呈現出來，也可以只有顯示類別之間的關聯，這些關聯就是程式碼之間的相依性。這一點非常重要，因為要從一大堆原始程式碼中找出它們之間的相依性，相對之下是比較困難的。使用類別圖型可以讓你清楚的瞭解某一些特定的相依性，尤其在一個高度抽象化的軟體系統中，瞭解這個特性更加重要。

類別節點

「類別節點、Class Node」是用來表示一個 Java 類別的基本單位，它可以用來表示 Java 程式語言中三種基本單位：類別(class)、抽象類別(abstract class) 和介面(interface)。

樣式

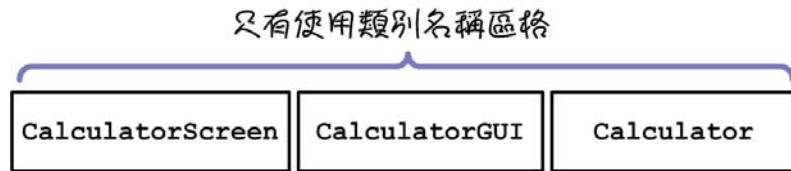
使用類別節點表示類別的時候，可以根據你想要表現的詳細程度，以不同的樣式來呈現這個類別，以下列的類別節點來說，它顯示類別節點的三個區格：



一個詳細表現的類別節點分成上、中、下三個區格 (compartments)，分別代表「類別名稱」、「屬性宣告」和「方法宣告」，有了這個詳細的類別節點以候，就可以宣告出下列的類別：

```
-----  
public class Calculator {  
  
    private static final char NO_OP = '\0';  
    private static final char PLUS = '+';  
    private static final char SUBTRACT = '-';  
    private static final char MULTIPLY = '*';  
    private static final char DIVIDE = '/';  
  
    private float number1 = 0.0F;  
    private char operator = NO_OP;  
  
    public String opEquals(String number) {...}  
    public String opAdd(String number) {...}  
    public String opSubtract(String number) {...}  
    public String opMultiply(String number) {...}  
    public String opDivide(String number) {...}  
    private float performOperation(float number2) {...}  
    protected static float parseNumber(String number) {...}  
}
```

但是如果你要顯示比較多的類別，又不想讓整個類別圖型太複雜時，可以省略「屬性宣告」和「方法宣告」區格，這樣的表示方式並不代表這個類別沒有屬性和方法的宣告：



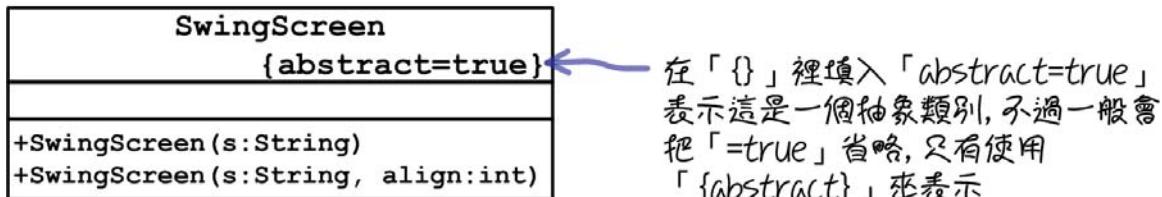
上列的圖型呈現了三個類別的宣告：

```
public class Calculator {...}
```

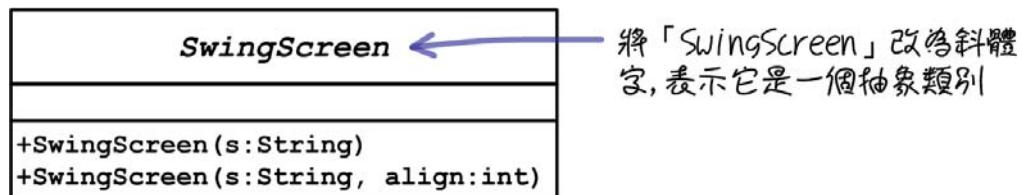
```
public class CalculatorScreen extends SwingScreen {...}
```

```
public class CalculatorGUI {...}
```

類別名稱區格除了類別名稱外，還可以填入附加的資訊，用來表示類別的特性或是額外的說明。例如抽象類別：

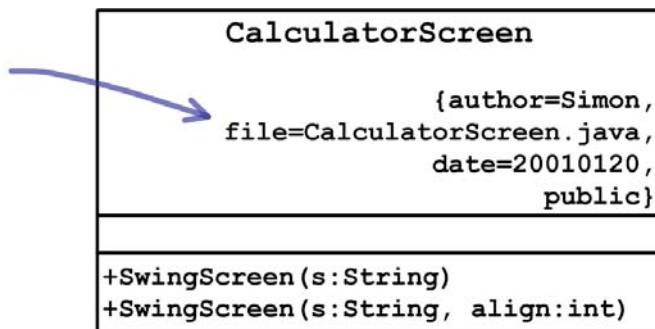


上列用來表示抽象類別的類別節點是目前比較建議的作法，舊有的方式是將類別名稱使用「斜體字」來表示抽象類別，不過因為斜體字與一般字體比較不容易分辨，建議使用上一種作法：



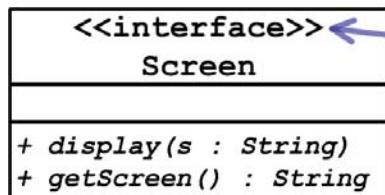
有時候可能會希望在類別節點上顯示一些額外的資訊，例如作者或原始檔名，你可以使用下列的方式來表示：

使用「名稱=值」的方式來表示資訊，有多個時使用逗號隔開。如果只有名稱而沒有值的時候，預設的值是「true」



以上列圖中的訊息來說，真正對程式設計師有用的並不多，除了最後一個「public」，與表示抽象類別時一樣，你如果省略值的指定，表示值為預設的「true」，所以「public=true」告訴你這個類別的存取範圍要宣告為「public」外，其它的部份大概也只能放到註解裡而已。

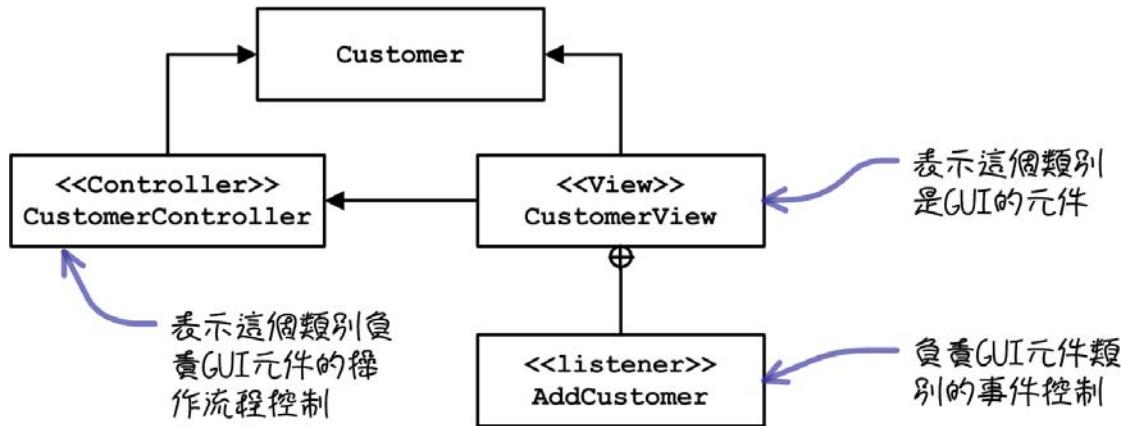
如果要表示某一個類別節點是一個介面的話，就會使用到「Stereotypes」：



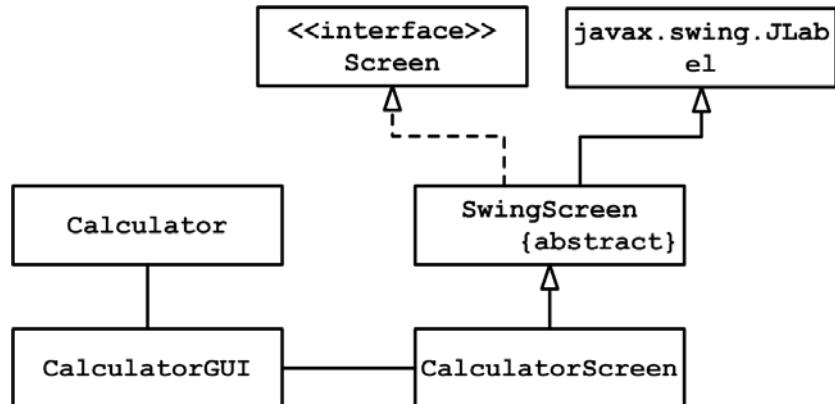
加入「Stereotype」的標示：「<<interface>>」，表示這是一個介面

UML 制定了許多「Stereotypes」，它指的是在「<<」和「>>」裡填入特定的標示，來表示特殊的意義。想要單純的以圖型和線條來呈現一個軟體系統是不夠的，所以就產生了「Stereotypes」，包含「物件限制語言、Object Constraint Language」，一般會簡稱為「OCL」，也是另外一種用來加強圖型資訊的規格。

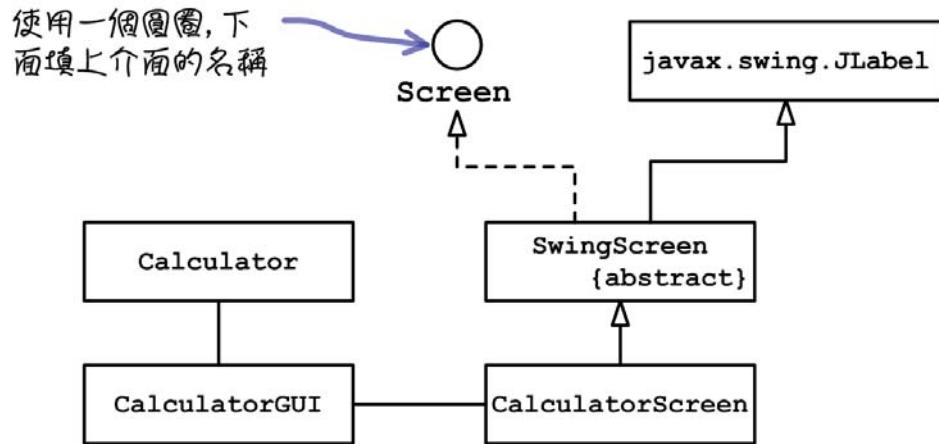
上列用來表示介面的「`<<interface>>`」是 Stereotypes 的一種。另外也可以使用 Stereotypes 來表示某個類別特殊的角色，下列的圖型是一般常見的「MVC Design Pattern」，是一種關於使用者操作介面的設計模式：



不論是類別、抽象類別或是介面，都可以省略「屬性宣告」和「方法宣告」區格，讓你把軟體系統中比較多的類別節點顯示出來，更完整的呈現出軟體系統架構。下列的圖形是一個模擬計算機的類別架構，不過都省略了「屬性宣告」和「方法宣告」區格：



在各種樣式當中，「介面」有一種比較特殊的表示方式，這是為了讓比較大型的類別圖型，可以增加可讀性的一種作法：



屬性區格

在屬性區格中可以完整的告訴你關於屬性宣告的資訊，在區格裡使用下列的語法來宣告屬性：

<存取範圍> <屬性名稱> [[<數量>] [<順序性>]] : <型態> [= <初始值>]

■ 存取範圍：

修飾子	符號
public	+
protected	#
package private	~
private	-

■ 屬性名稱：使用 Java 程式語言中合法的識別字。

■ 數量和順序性：

表示法	說明
數字	確定的數量
*	零到多個
0..*	零到多個
0..1	零到一個
1..*	一到多個
n..m	最少 n 個，最多 m 個

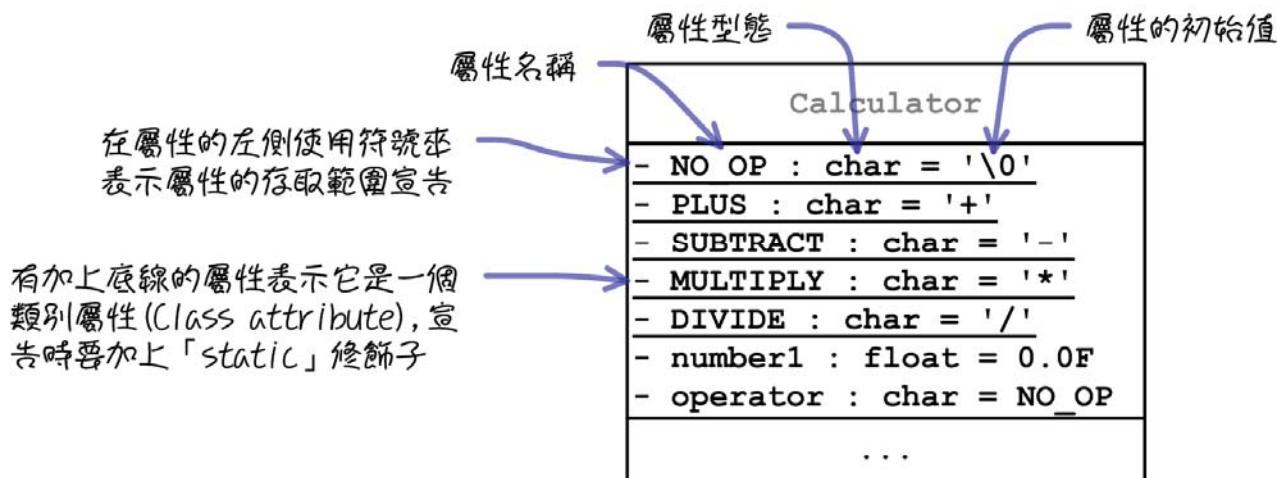
當屬性的數量是一個以上的時候，你可以使用順序性的宣告來表示這個屬性的值是否需要排序，使用下列兩個關鍵字：

關鍵字	說明
Unordered	不需要排序
Ordered	需要排序

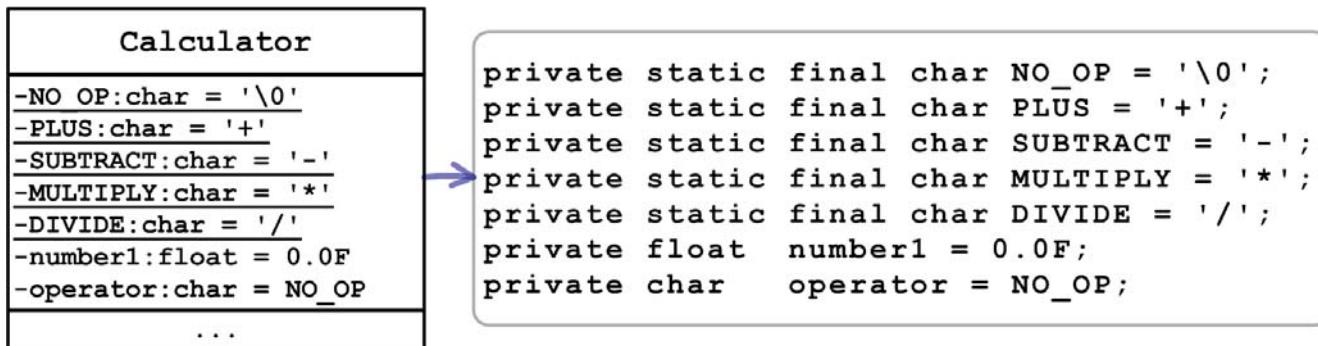
■ 型態：屬性的資料型態，使用八種基本資料型態或類別名稱。

■ 初始值：可選擇性的宣告，為屬性宣告一個合法的初始值。

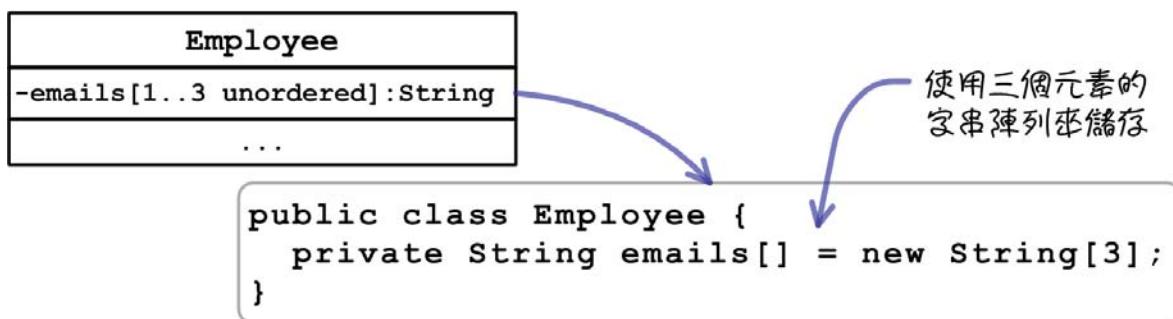
以下列的類別節點來說明關於屬性宣告的資訊：



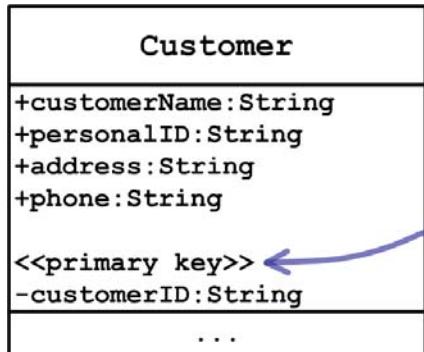
屬性區格在屬性的定義，把屬性的名稱放在型態之前，中間使用冒號「：」隔開，這樣的前後順序與 Java 的語法是相反的。下列是 Java 宣告屬性敘述與屬性區格中屬性定義的對照：



下列是屬性數量和順序性宣告的範例：



除了屬性的定義外，你可能也想表示某些屬性的特性，例如某些屬性是主要的鍵值。想要表示這些額外的特性，同樣是使用「Stereotypes」：



使用額外的「Stereotype」的說明「<<primary key>>」表示屬性「Stereotype」是一個主要鍵值

UML 中並沒有關於「final」屬性的表示方法，你可以依照 Java 的命名慣例來判斷是否要加上「final」的宣告，它們的屬性名稱應該都是大寫，字與字之間使用底線連接。

方法區格

在方法區格中可以完整的告訴你關於方法宣告的資訊，在區格裡使用下列的語法來宣告方法：

<存取範圍> <方法名稱> ([<參數>]) : <回傳型態>

在宣告方法的語法中：

■ 存取範圍：

存取範圍修飾子	符號
public	+
protected	#
package private	~
private	-

一般也會稱它為「package friendly」，以前的作法是不加任何符號，使用「~」是目前比較新的作法

■ 方法名稱：使用 Java 程式語言中合法的識別字。

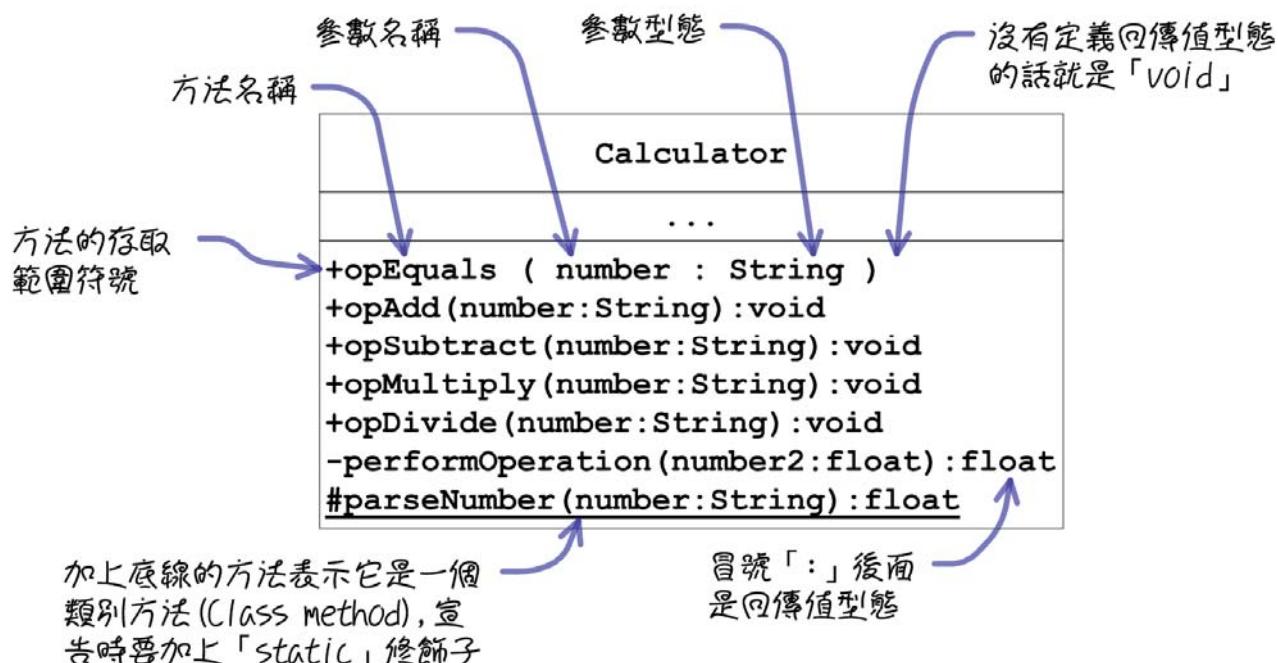
【圖】參數：可選擇性的宣告，是一個使用逗號分隔的清單，使用下列的語法來宣告方法的參數：

[種類] <參數名稱> : <參數型態>

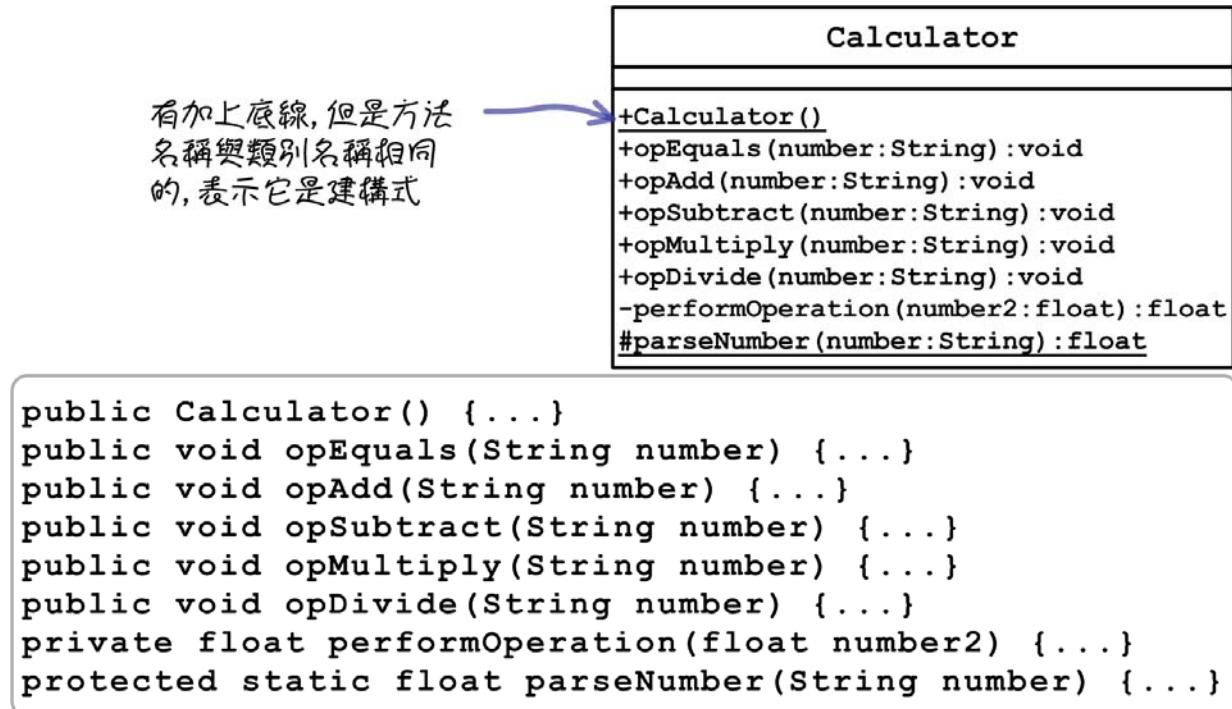
- 種類：可選擇性的宣告，在 Java 實作上通常會省略，但是如果特別加上「in」的宣告，就要把這個參數宣告為「final」。
- 參數名稱：使用 Java 程式語言中合法的識別字。
- 參數型態：參數的資料型態，使用八種基本資料型態或類別名稱。

【圖】回傳型態：方法回傳值的資料型態，使用八種基本資料型態或類別名稱。

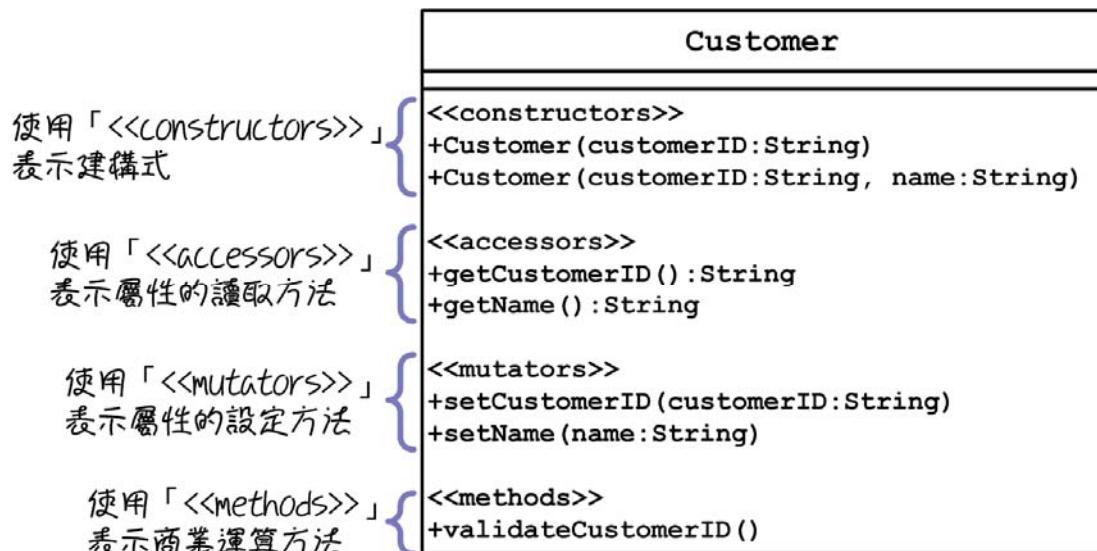
以下列的類別節點來說明關於方法宣告的資訊：



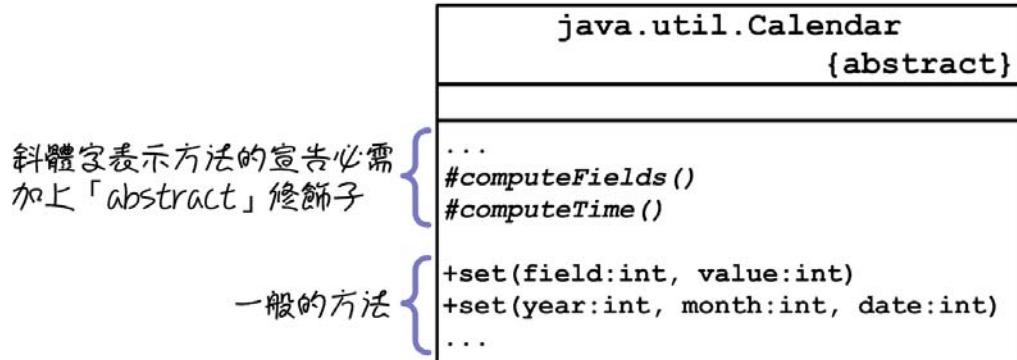
方法區格在參數的定義，把名稱放在型態之前，中間使用冒號「：」隔開，這樣的前後順序與 Java 的語法是相反的。下列是 Java 告訴方法敘述與方法區格中方法定義的對照：



與屬性區格一樣，你也可以使用「Stereotypes」來標示某些方法的特性：



在上面兩個範例中都沒有提到「抽象方法、Abstract methods」，一個只有宣告沒有實作的方法。在方法區格中使用斜體字來表示抽象方法：

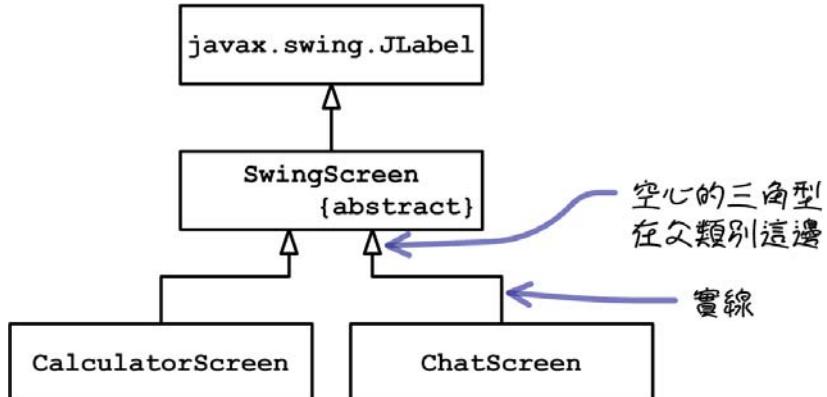


結合關係

類別圖型最重要的特性之一，就是可以表現出軟體系統裡，所有關鍵類別的關聯。使用類別圖型來呈現這些關聯的資訊，不論是應用在系統的分析與程式設計師之間的討論，都比直接使用原始程式碼來得容易許多。

繼承 (extends)

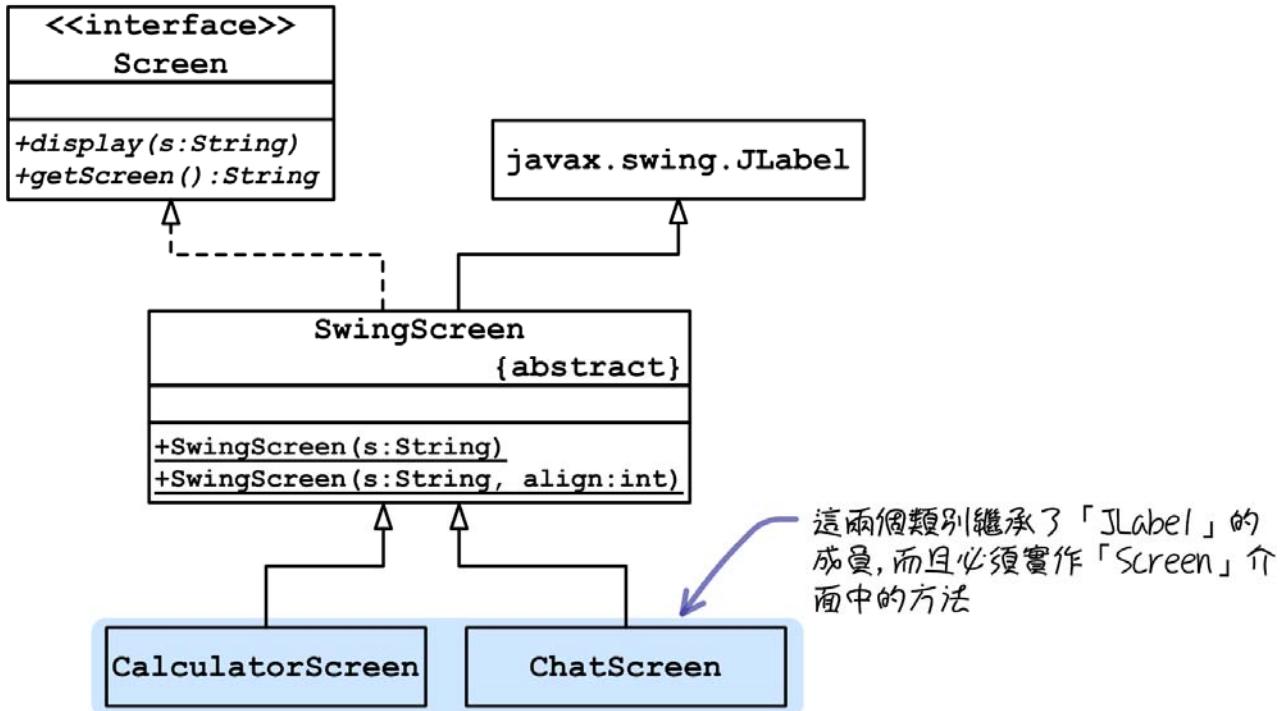
經過抽象化的動作後，往往會產生許多類別之間的繼承關係，使用類別圖型的繼承表示方法可以清楚的瞭解複雜的繼承架構，類別圖型使用「Generalization arrow」來表示類別之間繼承的關聯：



從上列的類別圖型，可以很清楚的知道這些類別之間的繼承關係：

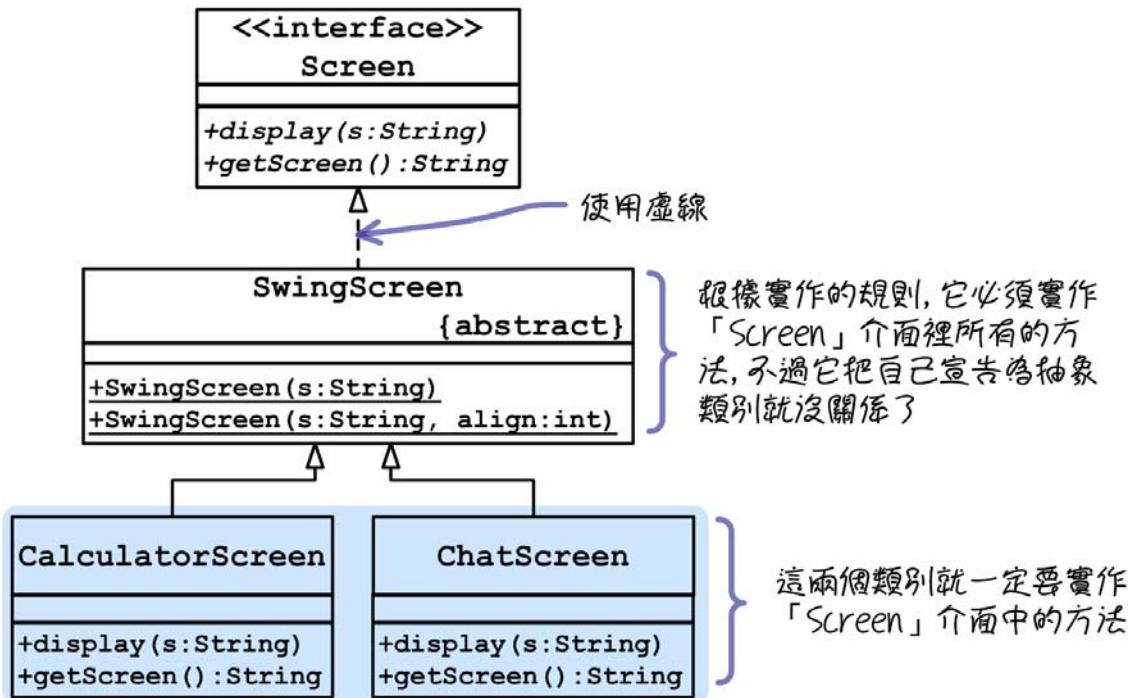
```
public abstract class SwingScreen extends javax.swing.JLabel {  
}  
  
public class CalculatorScreen extends SwingScreen {  
}  
  
public class ChatScreen extends SwingScreen {  
}
```

在類別圖型中如果詳細的表現出類別節點的資訊，包含所有的屬性和方法定義，你可以從圖中知道子類別繼承了哪些成員，如果子類別繼承的是一個抽象類別，也可以知道子類別必需實作哪些方法，這在龐大的繼承架構裡非常有用：

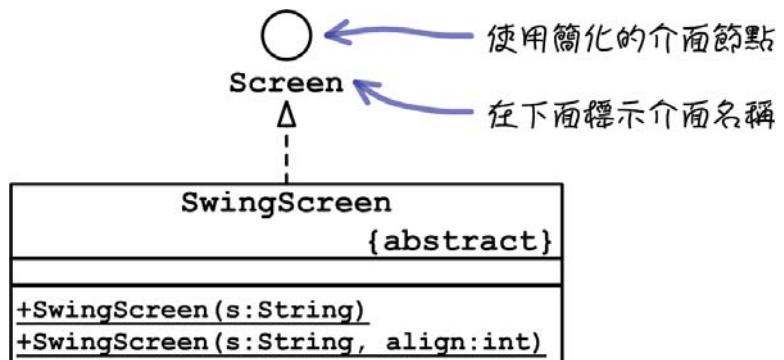


實作 (implements)

類別圖型使用「Realization arrow」來表示類別之間的實作關聯：



上列的類別圖型也可以使用簡化的介面表示方法：

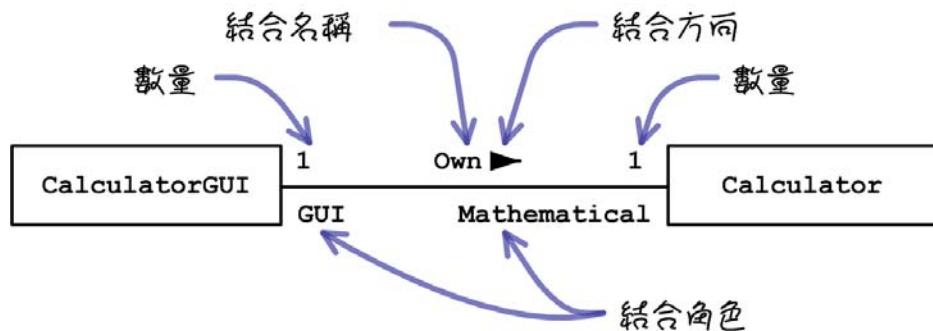


結合

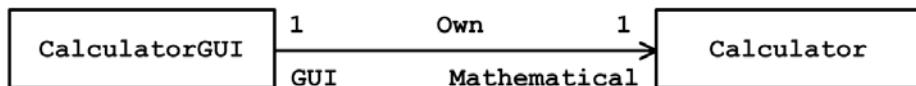
類別和類別之間的關聯都可以稱為「結合、Association」。使用結合可以表現類別之間的：

- ▣ 抽象的結合關聯。例如「擁有」或「使用」
- ▣ 抽象的結合角色。例如「線上客戶」或「系統管理員」
- ▣ 結合的數量。例如「Car」與「Wheel」的結合數量是 1 跟 4
- ▣ 結合的方向。例如表示「使用者」與「被使用者」

將這些結合的元素使用在圖型的話，會像這樣：



以上列的圖型來說，可以在結合線條上直接使用箭頭來表示方向，這樣就可以省略掉結合方向的符號，這也是一般比較常用的作法：



上列的結合範例圖型，對應到 Java 類別的宣告會像這樣：

```

public class CalculatorGUI {
    private Calculator calculator;
}
  
```

在擁有着類別裡
會有一個被擁有着型態的資料

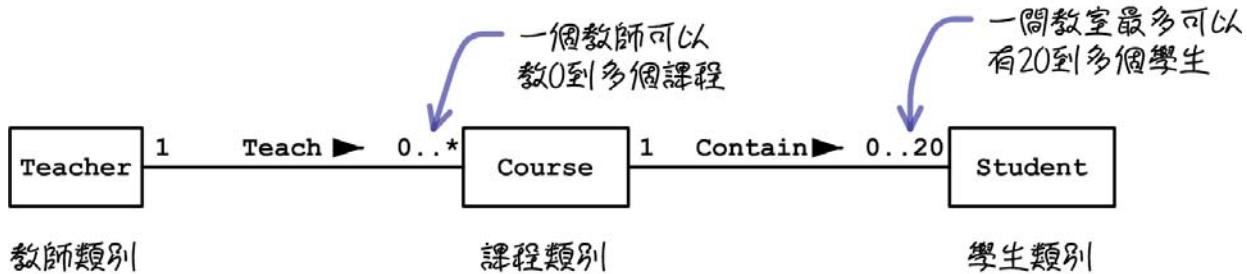
在結合關係裡，結合數量通常表示軟體系統中一些重要的行為，關於結合數量的標示可以使用下列幾種方法：

表示法	說明
數字	確定的數量
*	零到多個
0..*	零到多個
0..1	零到一個
1..*	一到多個
n..m	最少 n 個，最多 m 個

Java 在結合數量上的實作，可以使用下列幾種方法：

- 允許零個時，表示物件變數的值可以是「null」
- 確定的數量可以使用陣列或集合資料結構
- 不確定的數量可以使用集合資料結構

下列是一個使用不同結合數量的類別圖型範例：



上列的結合範例圖型，對應到 Java 類別的宣告會像這樣：

```

public class Teacher {
    private Set courses = new HashSet();
}

public class Course {
    private Student students[] = new Student[20];
}
  
```

Handwritten annotations explain the Java code:

- An arrow points to the declaration 'private Set courses = new HashSet();' with the text '使用集合資料結構來表示不確定的數量' (Use a collection data structure to represent an uncertain quantity).
- An arrow points to the declaration 'private Student students[] = new Student[20];' with the text '使用陣列來表示已知數量' (Use an array to represent a known quantity).

在軟體系統的實作上，類別之間產生相依性的狀況無所不在，上列討論的幾種結合狀況，是比較明顯的。但是相依性的意思應該是說「在軟體系統中的 A 類別會使用到 B 類別」，從這個條件來看，如果缺少了 B 類別，那這個軟體系統當然會出問題。

所以在結合關係裡，也可以使用「Stereotypes」來表示一些特殊的結合特性，讓類別圖型的結合特性可以更接近軟體系統的實作：

 <<local>>

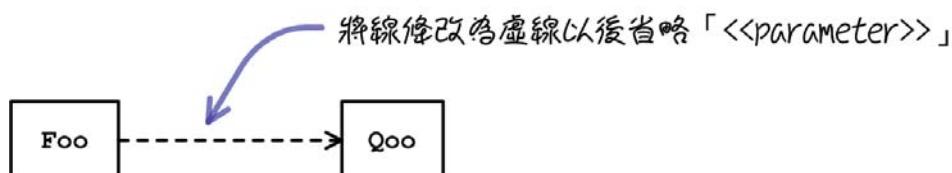


在「Foo」類別的區塊中建立與使用「Qoo」物件，只是一種區域變數，生命週期只有在區塊裡，彼此之間沒有直接的關係。

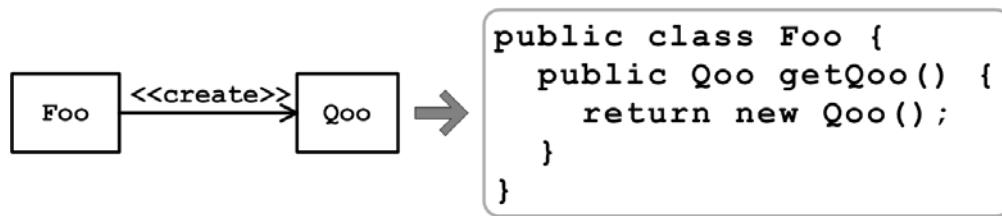
 <<parameter>>



在「Foo」類別中透過參數來取得「Qoo」物件，建立「Qoo」物件的責任在呼叫方法的類別。這種結合情況有另外一種常見的表示方法：

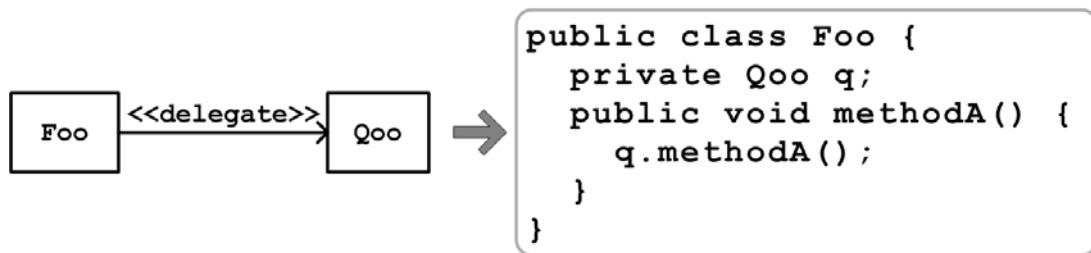


💻 <<create>>



「Qoo」物件由「Foo」負責建立，然後傳給需要的地方，這種作法稱為「Factory method」，是一種常見的「設計模式、Design pattern」。

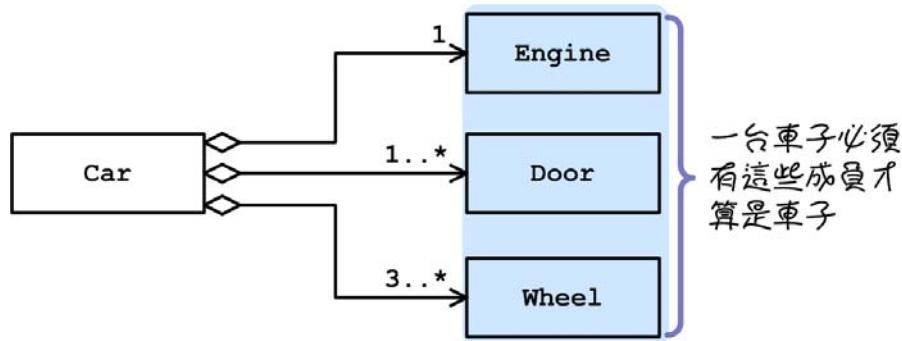
💻 <<delegate>>



「Delegate」是代理的意思，「Foo」類別在這種情況下扮演「代理人」的角色，由「Foo」類別負責將方法的呼叫轉為呼叫「Qoo」的方法，有許多設計模式都會使用這種結合，例如 Command、Proxy 或 Composite。

聚合

「聚合、Aggregation」是一種特殊情況的結合關係，它建立在「整體、whole」與「成員、member」的概念上，這個整體必須擁有這些成員才具有意義：



以上列的範例來說，如果一台沒有有引擎、車門和輪子的車子，應該就不算是一台車子；但是引擎、車門和輪子都是可以獨立存在，並且有它們各自的用途，例如引擎不一定要裝在車子上。

聚合在 Java 的實作上，並沒有特別明顯的方法，它跟一般的結合並沒有什麼差別，對 Java 程式設計師來說，唯一該注意的地方，就是要確定被擁有者一定會被建立：

```

public class Car {
    private Engine engine = new Engine();
    private Set doors = new HashSet();
    private List wheels = new ArrayList();

    public Car(Engine e, Set doors, List wheels) {
        this.engine = engine;
        this.doors = doors;
        this.wheels = wheels;
    }
}

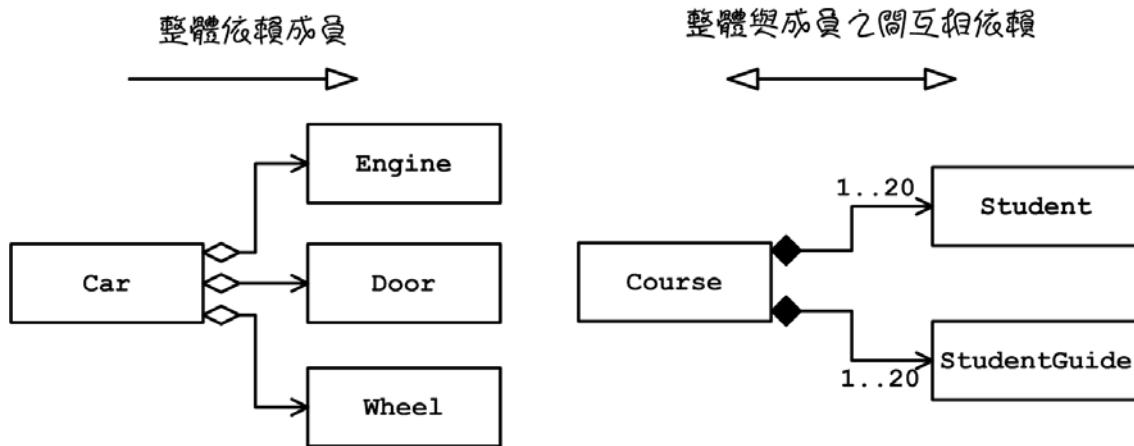
```

這個部份與一般的結合並沒有什麼差別

但是要確定建構汽車類別的時候，也要建構成員物件

組合

「組合、Composition」是一種特殊情況的聚合關係，它跟聚合的差別在於方向的角度，你可以把聚合視為「單向」，而把組合看成「雙向」的結合關係：



在上列的範例中，一個課程必須擁有學生與教材，才算是一個課程，這個特性跟聚合是一樣的；但是學生與教材也必須有課程開課了，才能夠去上課，也就是說如果沒有課程存在，學生與教材是沒有意義的。

組合在 Java 的實作上，跟聚合一樣，同樣要確定被擁有者一定會被建立：

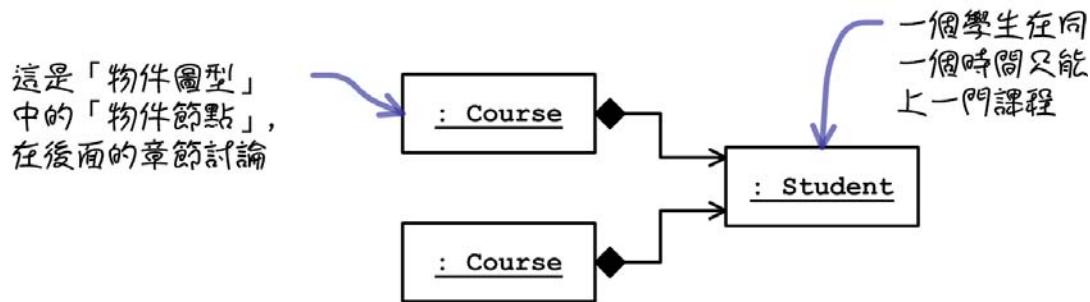
```
public class Course {
    private Student students[];
    private SudentGuide sg[];

    public Course(Student[] students, SudentGuide[] sg) {
        this.students = students;
        this.sg = sg;
    }
}
```

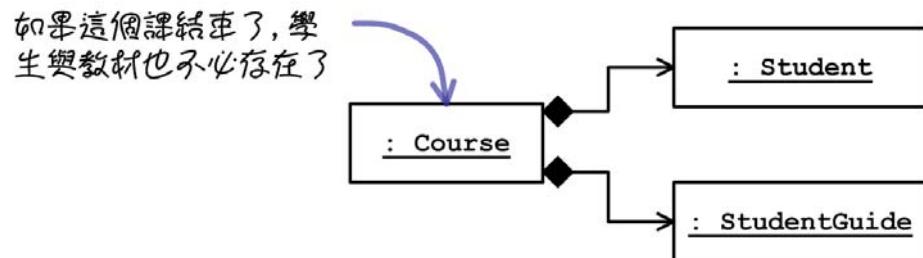
確定建構課程類別的時候，也要建構成員物件

因為組合的特性，另外要注意兩個在實作上的狀況：

- 一個成員物件不能同時被兩個擁有者擁有



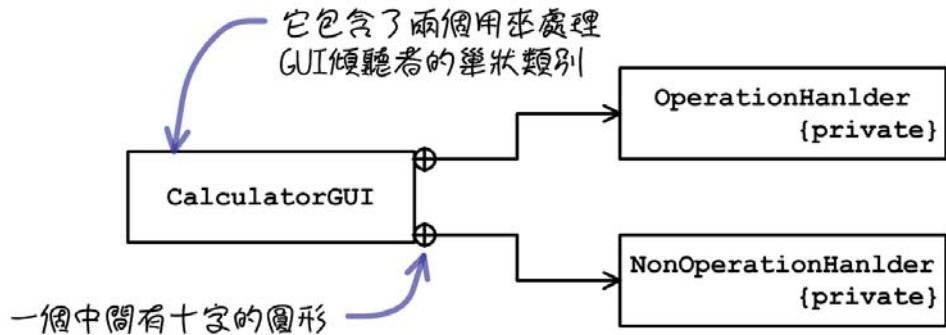
- 擁有者要負責成員的的生命週期



在 Java 技術中，物件的「解構、deconstruct」是透過「Garbage collection」來完成的，程式設計師並不需要去管理物件的生命週期；但是在一些特殊的情況下，你可能需要覆寫「Object」類別中的「finalize」方法，在這個方法中處理解構物件該作的事情，例如中斷取得的網路連線。

巢狀類別

「巢狀類別、Nested class」的應用非常多，你可以使用下列的結合關係來表示內部類別：



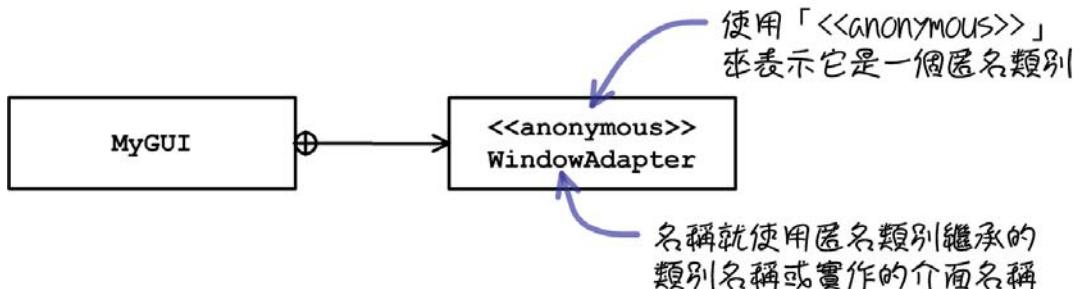
上列的圖型範例對應到類別的宣告會像這樣：

```
public class CalculatorGUI {
    private class OperationHanlder
        implements ActionListener { ... }

    private class NonOperationHanlder
        implements ActionListener { ... }
}
```

宣告在外層類別中的兩個巢狀類別

巢狀類別中有一種比較特殊的情形是「匿名類別、Anonymous class」，UML 並沒有特別制定關於匿名類別的表示方法，所以同樣使用巢狀類別的表示方法，但是在類別節點上作一些調整：



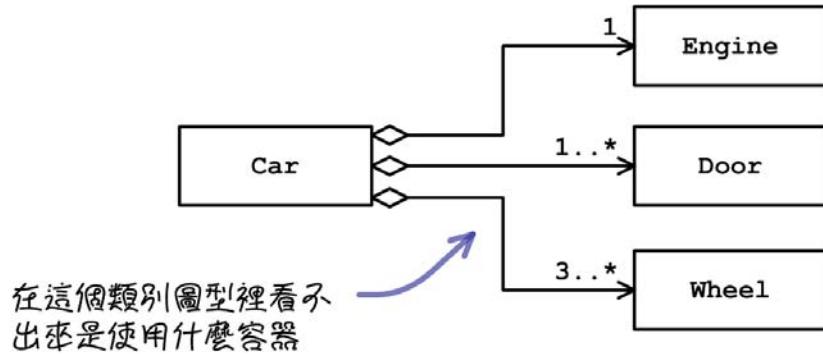
上列的圖型在傳統的「AWT」圖形介面中很常見，對應到類別的宣告會像這樣：

```
public class MyGUI {  
    public void startGUI() {  
        frame = new Frame();  
        frame.addWindowListener(  
            new WindowAdapter() {  
                public void windowClosing(WindowEvent e) {  
                    frame.dispose();  
                }  
            }  
        );  
    }  
}
```

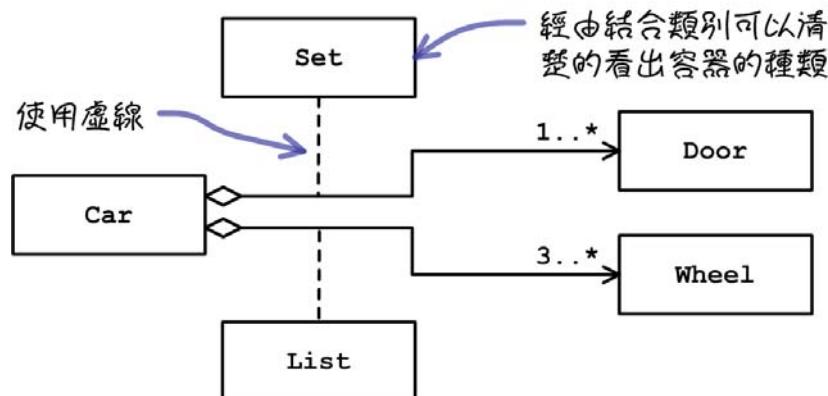
繼承自「WindowAdapter」
的匿名類別

結合類別

在討論聚合時使用的範例圖型，它所顯示出來的資訊還是不夠詳細：



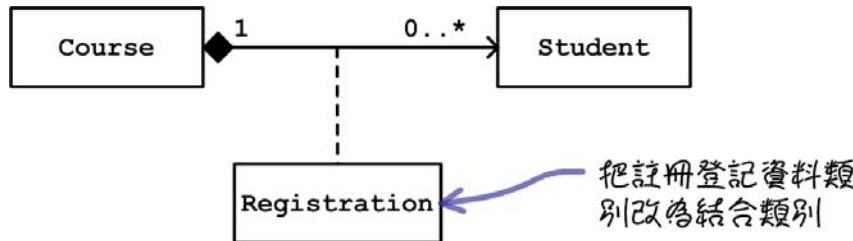
如果想要圖型中顯示容器的種類，可以使用「結合類別、Association class」來表示：



這樣的顯示方法，在比較複雜的類別結合情況下，可以很清楚的看出彼此的結合關係。以下列的範例圖型來說，這種類別圖型並沒有辦法表達出所有的資訊：



改用結合類別來表示的話，可以補足上圖欠缺的資訊：



上列的圖型範例對應到「Course」和「Registration」的類別宣告會像這樣：

```

public class Registration {
    private Course course;
    private Student student;
}

```

結合類別包含兩側的類別資料

```

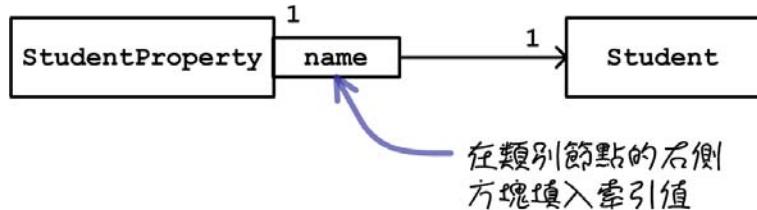
public class Course {
    public Registration getRegistration(Student student) {
        return new Registration(student, this);
    }
}

```

在這裡使用結合類別建立結合關係

限定結合

「限定結合、Qualifier associations」是一種特殊的結合表示方法，可以在結合關係中顯示索引的資訊。最常見的情況是在使用「Map」資料結構的時候：



上列的圖型範例，對應到「StudentProperty」類別的宣告會像這樣：

```

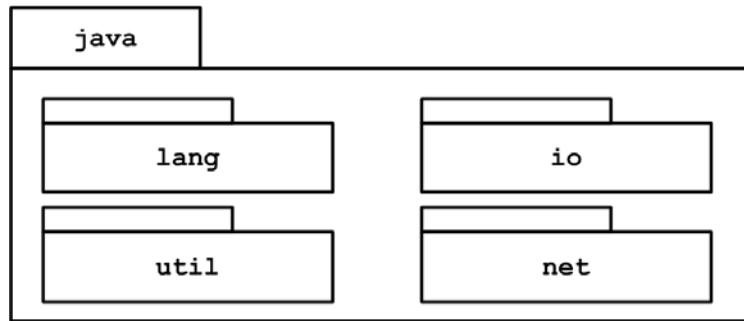
import java.util.*;
public class StudentProperty {
    private Map students = new HashMap(); ← 使用「Map」  
資料結構
    public Student getStuednt(String name) {
        return (Student) students.get(name);
    }
    public void addStudent(Student s) {
        students.put(s.getName(), s);
    }
}
  
```

} 透過索引值「name」
進行資料結構的操作

Memo

3. 套件圖型

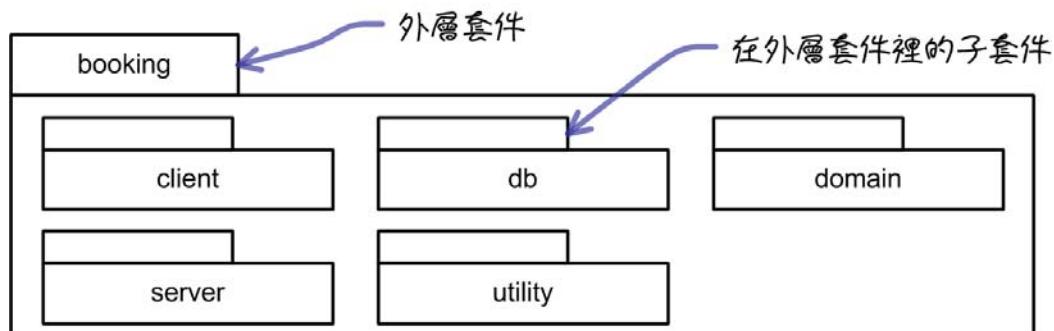
在 UML 中並沒有規定「套件圖型、Package Diagrams」，「套件、Package」在 UML 中只是一種特殊的「標記、Notation」，套件標記用來表示 Java 程式語言中的套件結構。例如 J2SE 內建的套件，可以使用下列的圖型來表示：



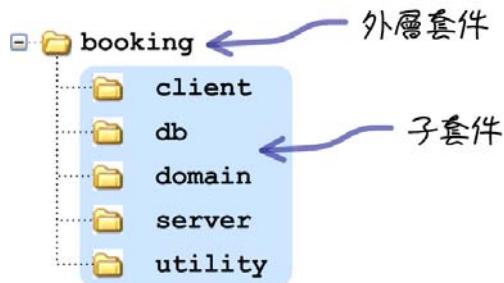
使用 Java 的套件規格將軟體系統切割為套件之後，套件與套件之間，也會產生類似類別之間的關聯，你可以把套件看成是一個包裝多個類別的「邏輯單位」，從邏輯單位來看整個軟體系統的架構，也就是說，把焦點從類別之間的關聯，提昇到套件之間的關聯，這對於龐大的軟體系統來說，幫助是非常大的。

套件標記

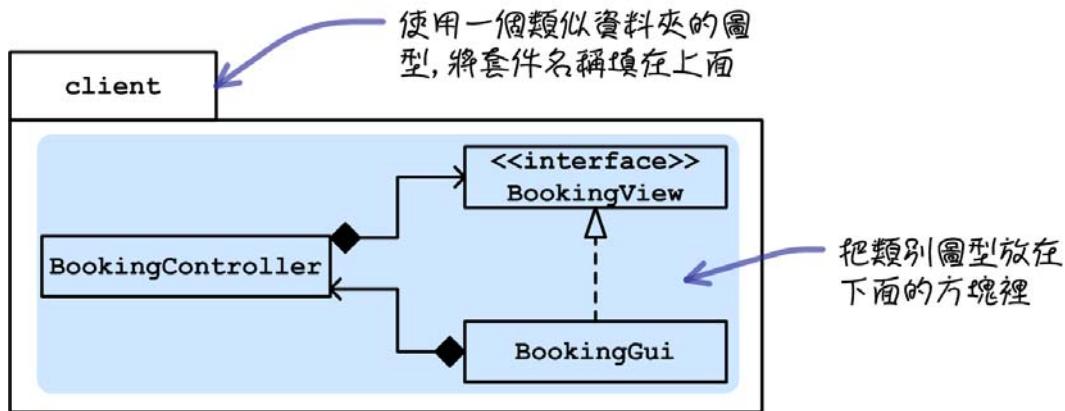
Java 的套件規格是採用「分類」與「資料夾」的概念，將不同分類的類別，使用資料夾來管理，在分割為套件的同時，也定義出「名稱空間、Namespace」，這樣就可以允許在不同的套件下有相同名稱的類別。一個套件標記 (Package Notation)，就相對於 Java 的套件宣告：



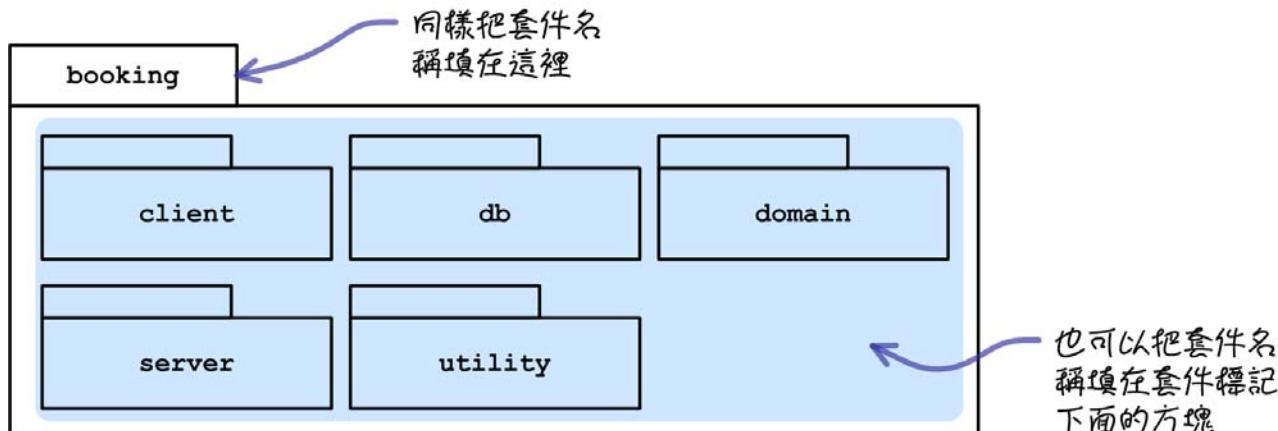
在 Java 裡的套件，會對應到檔案系統下的資料夾結構：



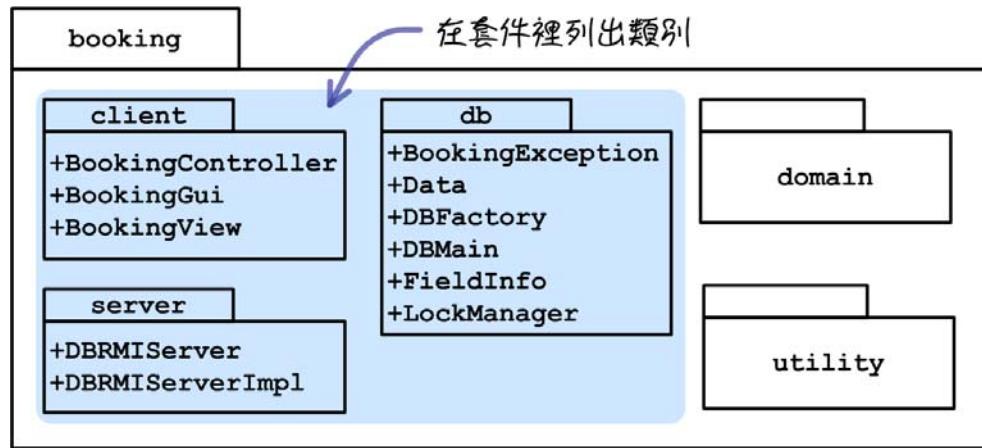
下列是一般常見的套件標記樣式：



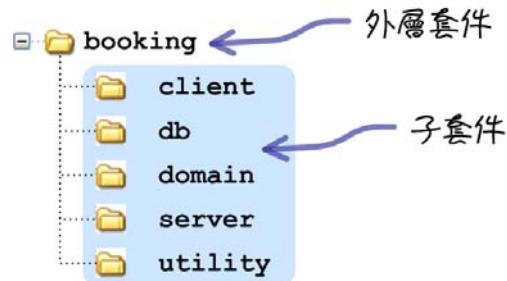
如果只是想呈現軟體系統中所有套件的邏輯單位，還要在套件中顯示類別圖型，會讓套件圖型過於複雜，在這樣的情形下，大多會省略類別圖型的顯示：



有時為了要顯示套件裡包含的類別，又不想讓類別圖型佔去太多空間，也可以使用下列圖型的作法：



在上列的圖型中，以「booking.client」套件來說，類別檔案在檔案系統中的位置應該要像這樣：



而對應到原始程式碼的宣告，應該要像這樣：

```

package booking.client;
public class BookingController { ... }

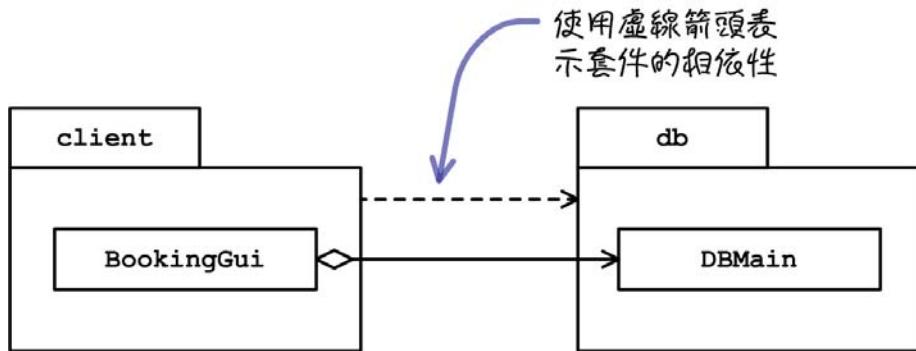
package booking.client;
public class BookingGui { ... }

package booking.client;
public class BookingView { ... }
  
```

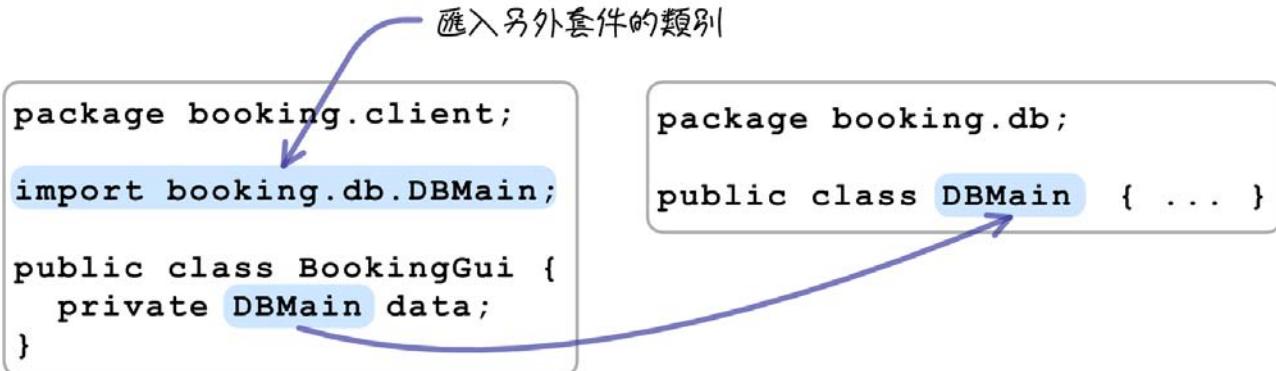
同一個套件下的
類別都會有相同的
套件宣告

相依性

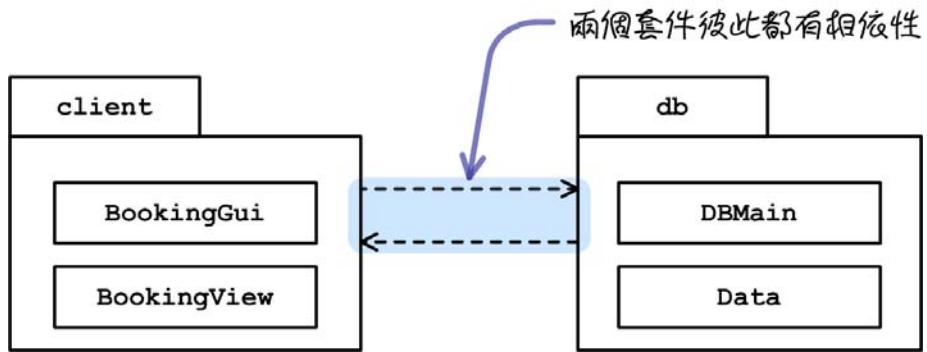
套件標記之間的「相依性、Dependency」，在 Java 實作裡指的就是「import」的關聯，如果在兩個套件標記之間有下列的結合關係，當你修改「db」套件時，就要考慮是否會影響到「client」套件的運作：



對應到 Java 的實作，表示在原始程式裡有這樣的宣告：



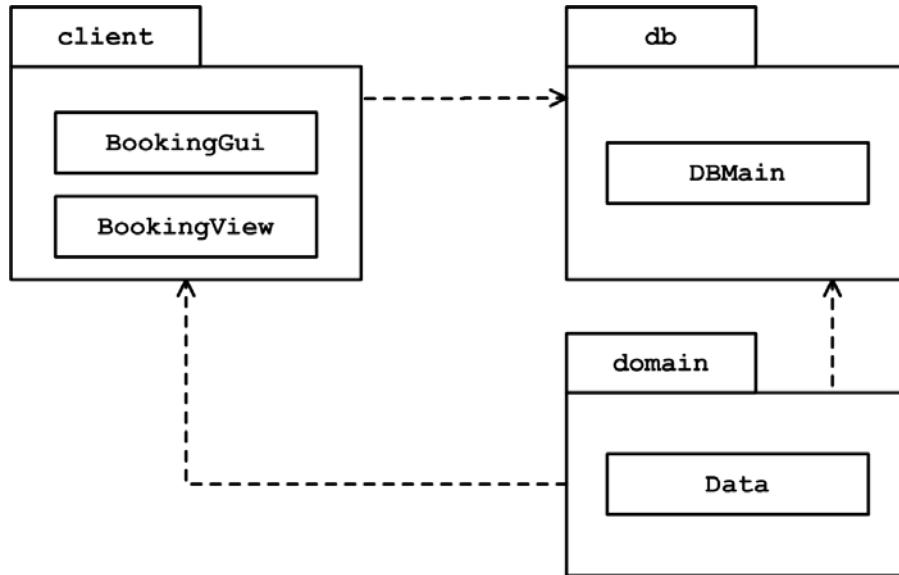
延續上列討論的「client」對於「db」的相依性，在軟體系統中，可能會出現這樣的情況：



對應到 Java 的實作，表示除了原有的「BookingGui」和「DBMain」之間的關聯外，在原始程式裡還有這樣的宣告：

```
package booking.db;  
import booking.client.*; ← 繞入「client」套件  
  
public class Data implements DBMain {  
  
    public void addChangeListener(BookingView bv)  
        throws BookingException {  
        changeListeners.add(bv);  
    }  
}  
  
package booking.client;  
public class BookingView { ... }
```

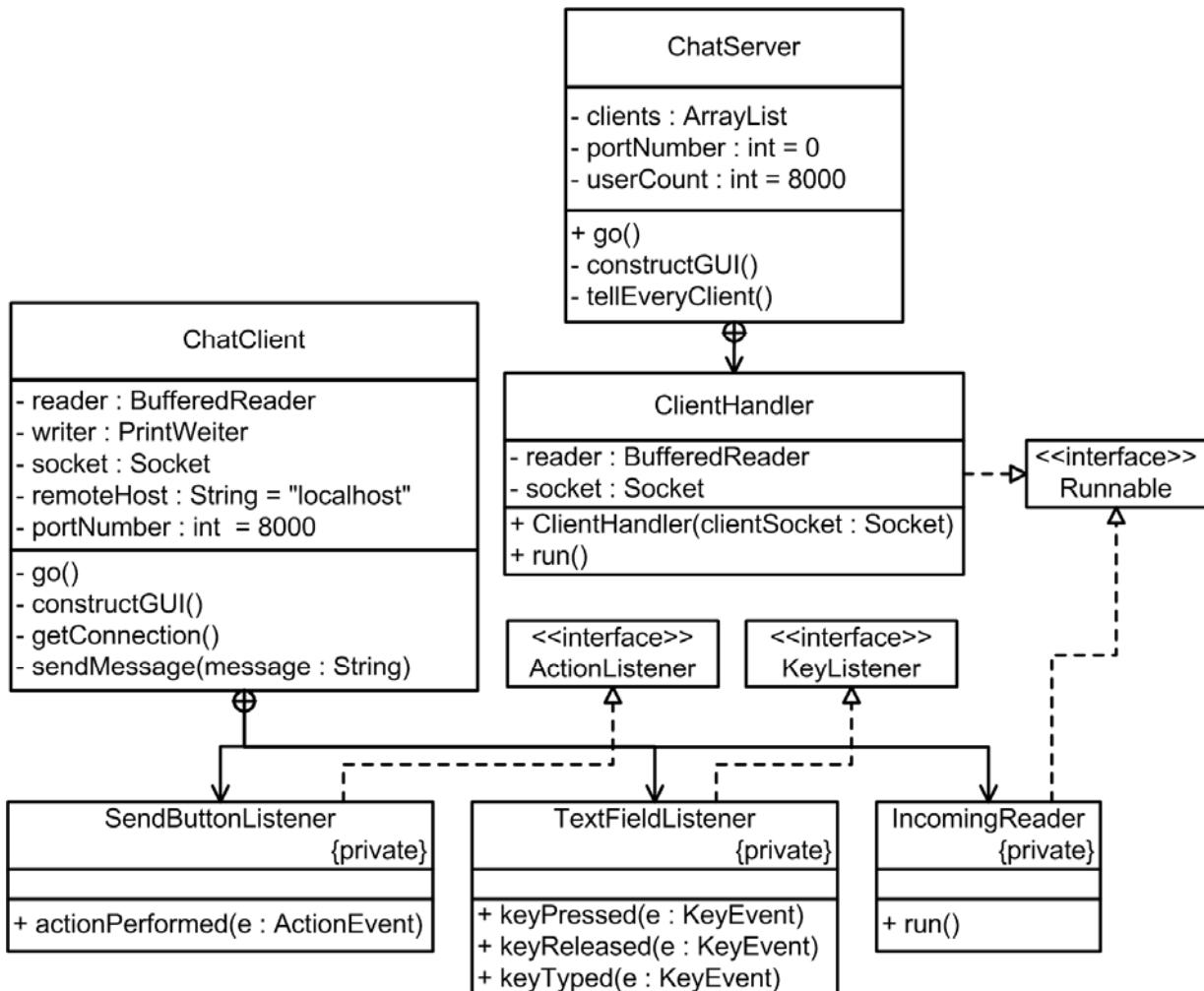
這種情形稱為「高度結合、High coupled」的套件相依性，或稱為「循環相依、Cyclic dependency」。雖然這樣的情形很常見，但這是比較不好的狀況，這個時候，通常會調整套件的規劃，避免循環相依的情形發生：



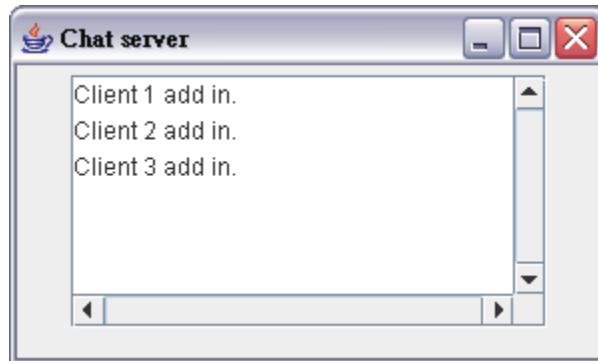
4. 物件圖型

「物件圖型、Object Diagrams」使用圖型的表示法來顯示軟體系統中「某一個時間點」的物件內容，還有物件彼此之間的關聯，一般會把它稱為「記憶體的快照、memory snapshot」。物件圖型所能夠顯示的資訊，有時在類別圖型裡是看不到的。

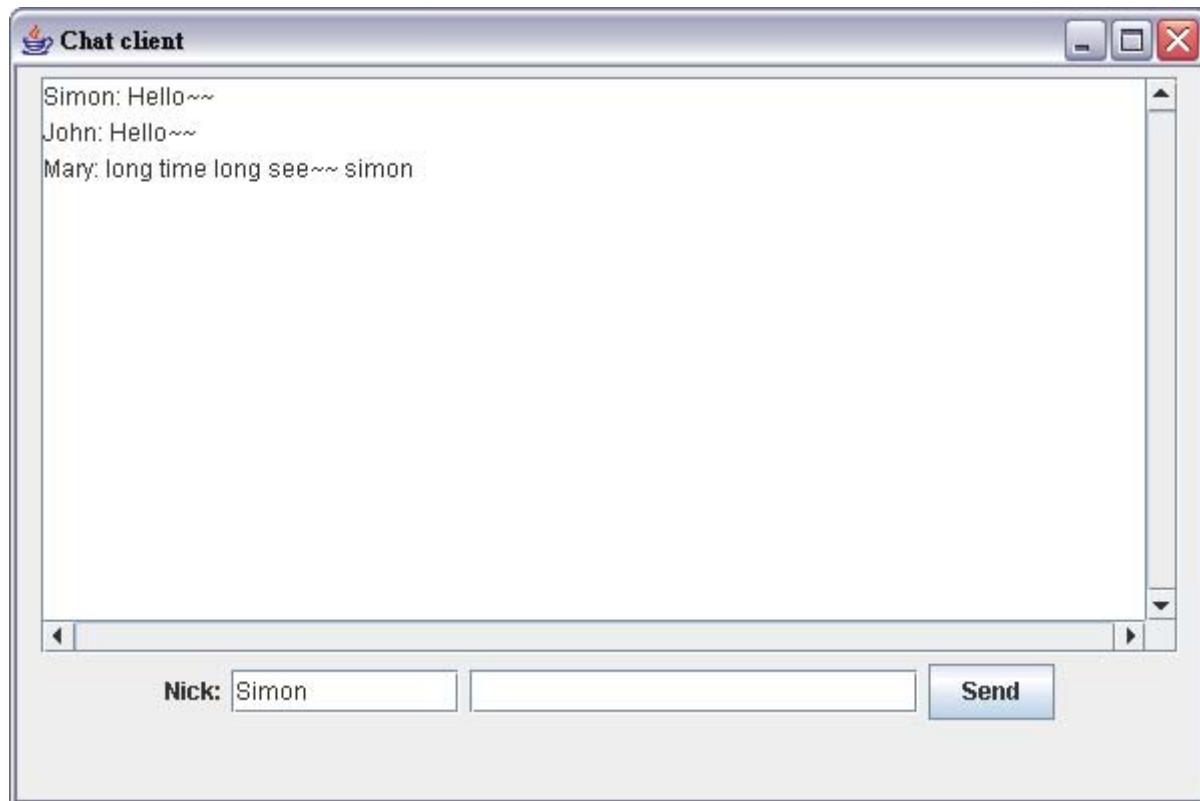
下列的類別圖型所呈現的應用程式，是一個可以讓多個用戶端同時進行聊天的程式：



「ChatServer」是一個簡單的 GUI 伺服器，它是一個可以接受多個用戶端連線的多執行緒程式。某一個用戶端發送訊息到「ChatServer」後，它會把這個訊息送到所有的用戶端：



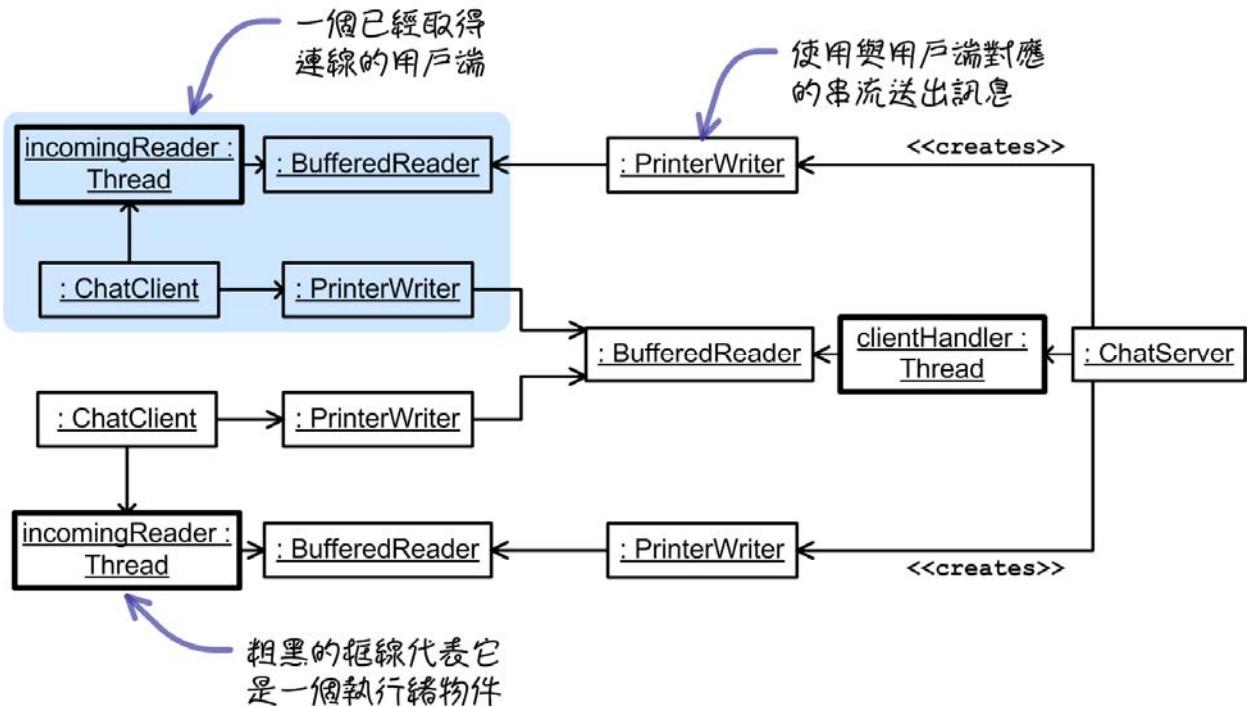
用戶端也是一個 GUI 程式，它在取得與「ChatServer」的連線後，就可以發送訊息給所有的用戶端：



如果你想要瞭解多個用戶連線到「ChatServer」後的運作，從上列的類別圖型中，是比較不容易看出來的，尤其是具有多執行緒特性的應用程式。

所以類似這樣的需求，應該要從物件圖型來瞭解會是比較好的方法。雖然物件圖型對於 Java 程式設計師來說，不像類別圖型代表類別的宣告這麼有意義，但是就像之前所討論的，當類別圖型表達的軟體系統架構不夠清楚時，你應該使用物件圖型來補充不足的部份，這樣的話，不論是在驗證類別圖型的完整性，或是在內部的討論和教學上，效果會是比較好的。

下列的物件圖型顯示在「ChatServer」啟動以後，有兩個用戶端取得連線時的物件狀態，圖型中並沒有畫出所有的物件節點：



上列的物件圖型中，每一個方塊代表程式運作時的物件。你可以從這個圖型看出物件之間的關聯，這樣的資訊，對於驗證類別圖型的完整性和瞭解應用程式的運作，幫助是很大的。

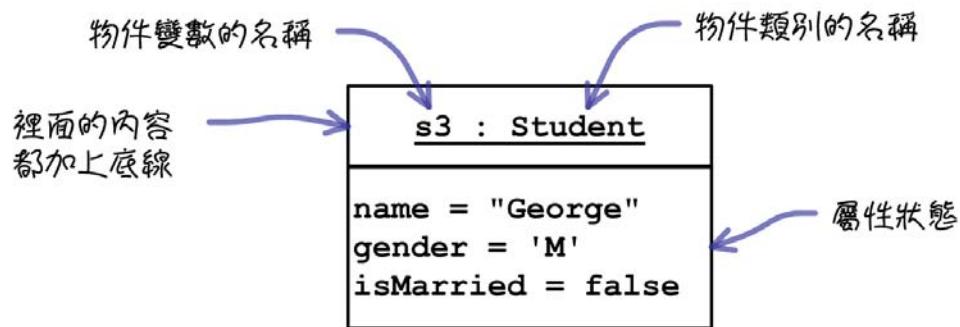
物件節點

「物件節點、Object node」用來表示一個在軟體系統中建立的物件，它是一種基礎的節點，在合作圖型和循序圖型中都會使用物件節點，表示目前軟體系統中正在活動的物件。物件節點可以表示：

- 具名物件 (Named object)
- 匿名物件 (Anonymous object)

樣式

一個物件節點分成上、下兩個區格，分別代表「物件名稱」和「屬性狀態」：

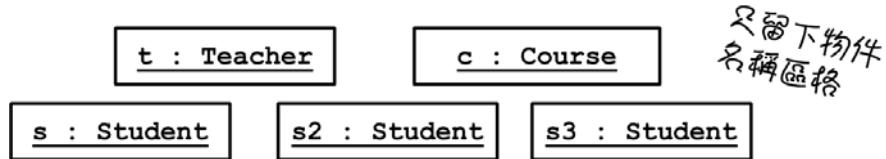


從上列的圖型中，你可以對應到 Java 建立物件的敘述：

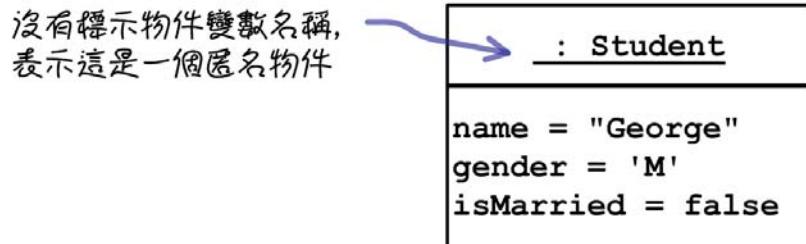
```
Student s3 = new Student( "George", 'M', false );
```

類別名稱
物件變數名稱
物件目前的屬性狀態

物件節點也可以省略屬性狀態區格，只保留物件名稱區格，這樣可以在物件圖型中放入更多的物件節點：



如果省略物件變數的名稱，表示這個物件是一個「匿名物件」：



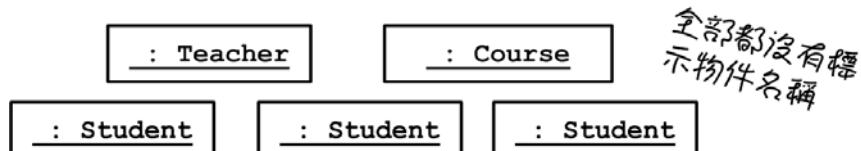
從上列的圖型中，對應到 Java 的敘述可能會像這樣：

```

students.add( new Student( "George", 'M', false ) );

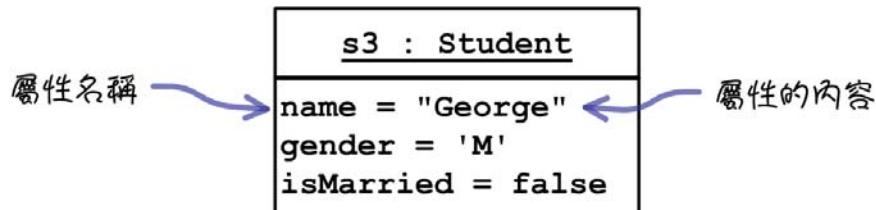
```

一般在物件圖型裡使用物件節點時，可能不會加上物件名稱，這是為了方便建立物件圖型的原因，而不是所有的物件節點都是匿名物件。為物件取什麼名稱，或是判斷這個物件是不是匿名物件，就留給程式設計師自己來判斷。所以像下列這樣的圖型也是可以的：

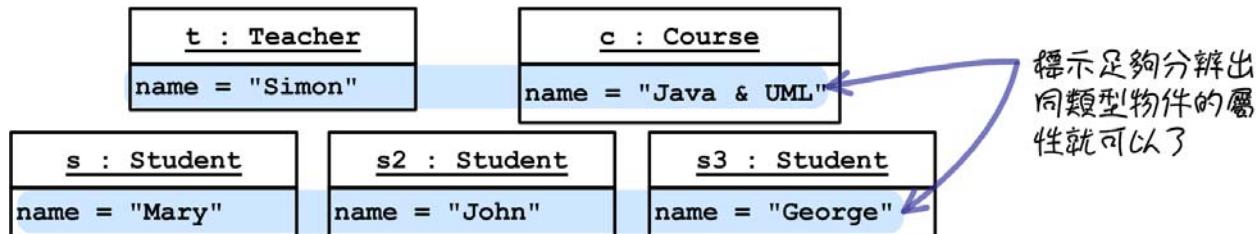


屬性區格

物件節點是用來表示軟體系統中，某一個時間點的物件狀態，物件狀態指的是屬性的內容，下列的物件節點顯示這個物件在應用程式裡的物件狀態：



在屬性區格裡比較普遍的表示方法，是只有標示關鍵的屬性值，你可以省略掉比較不重要的屬性，只要標示出來的屬性可以分辨這些物件的差異就可以了。例如下列的物件節點：

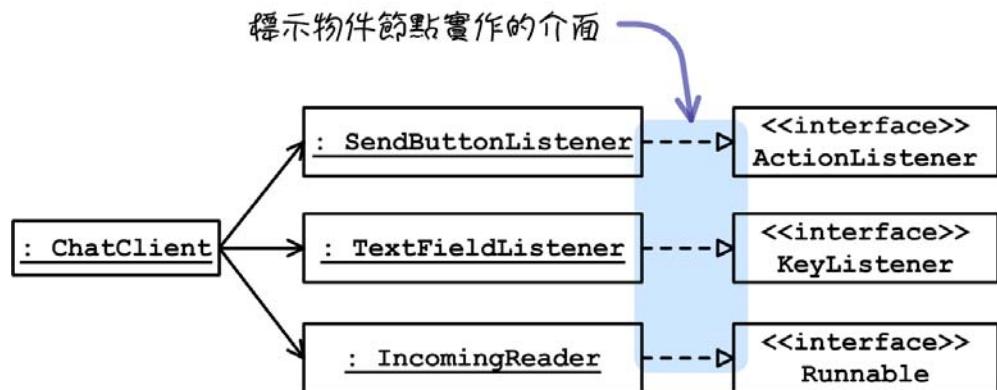


結合關係

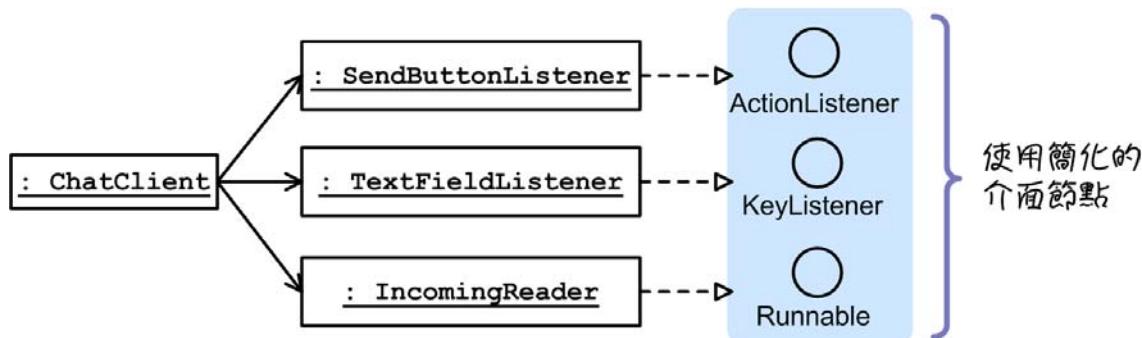
與類別圖型一樣，軟體系統裡的物件節點，彼此之間也會有許多不同的結合關係，讓你在物件圖型裡可以明顯的看出物件節點之間的關係。使用物件圖型來顯示某一個時間點的物件狀態，和這些物件彼此之間的相依性，不論是在應用在分析、設計或是程式設計師之間的討論，都會比直接使用原始程式碼來的容易。

實作

在物件圖型中，雖然可以藉由類別圖型得知某個物件節點的類別，是實作哪一個介面。但是通常會在物件圖型中加入物件節點與介面之間的實作關係，這樣的表示方法在物件多型的使用上，可以讓物件圖型呈現出來的資訊更加清楚：

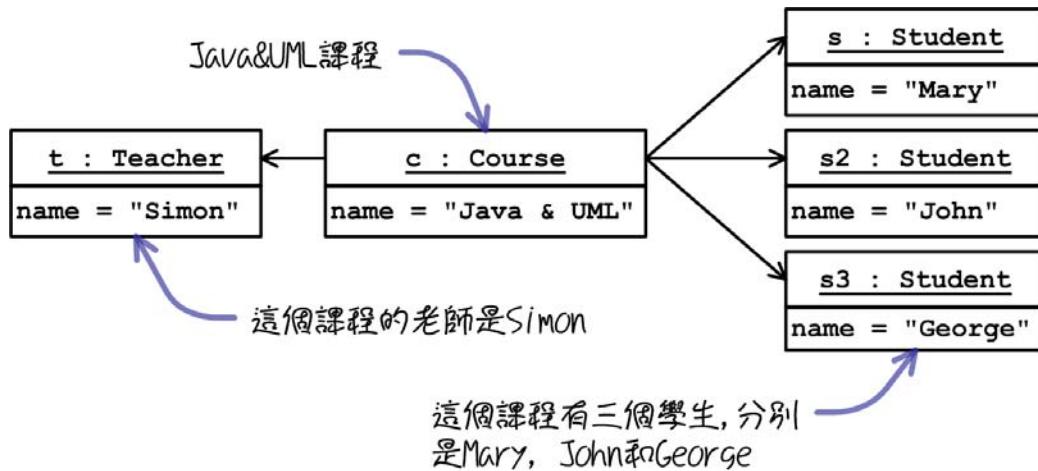


上列的類別圖型也可以使用簡化的介面表示方法：

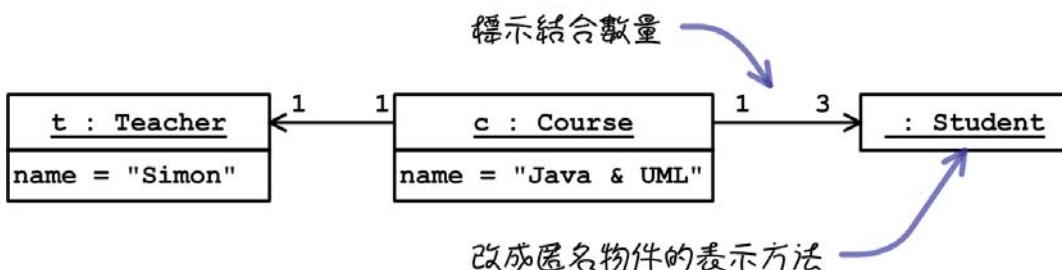


結合

物件節點的結合比類別節點相對來的單純，通常在物件圖型中，主要的目的是顯示物件節點之間的相依性。以下列的物件圖型來說，它可以很清楚的顯示老師、課程和學生物件之間的數量和相依性：



如果你需要瞭解軟體系統在某一個時間點，物件的相依性和確定的數量，這樣的表示方法非常有用。除了物件數量的呈現以外，上列的圖型還顯示每一個學生物件的狀態；但是如果學生的物件狀態對你來說並不重要的話，也可以使用下列的表示方法：



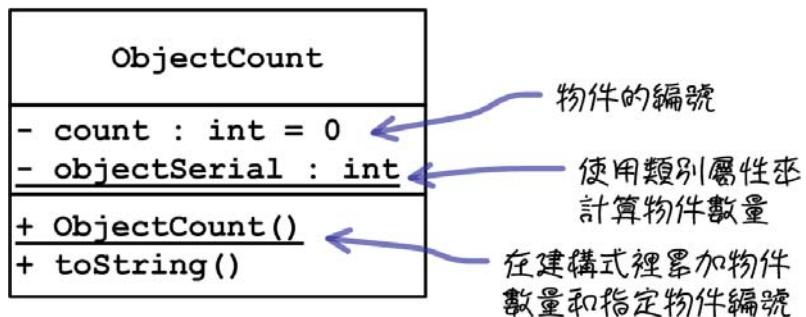
上列的圖型呈現的其實只有物件的相依性和數量的關係，在物件的「確定數量」對軟體系統很重要的情況下，你就可以使用上列的表示方法。在其它的情況下，從類別圖型中反而可以得到正確的資訊：



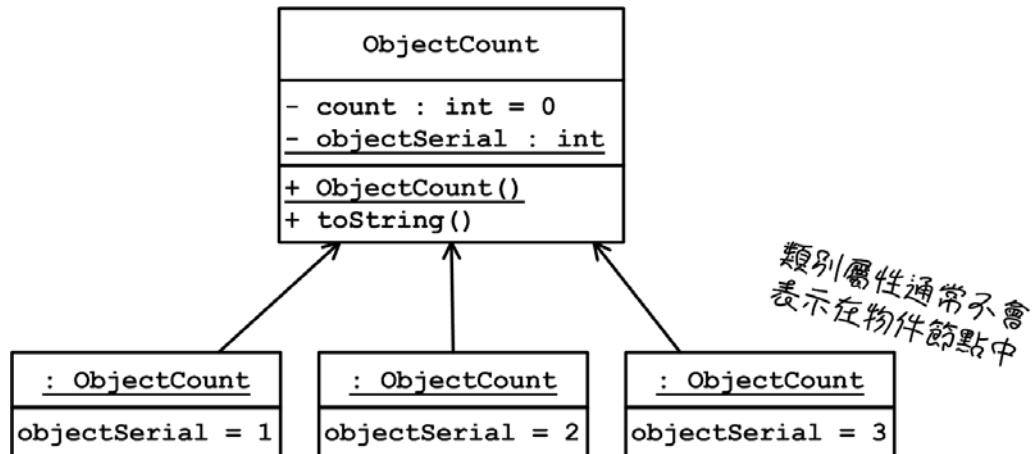
在類別圖型中其它的結合特性，例如聚合和組合，也會有跟上列討論同樣的情況，你可以根據軟體系統的特性來畫出物件圖型。一般來說，物件圖型的需求並不是那麼頻繁，與類別圖型相對之下，對於程式設計師的需求也不是那麼絕對，但是在需要的時候，卻也少不了它。

類別屬性

一般的類別屬性如果是應用在「公用變數」的時候，通常不會在物件圖型裡表現出來，因為它們通常會儲存一個固定的值，這些資訊在類別圖型就可以提供了。但是在某些情況下，類別屬性會是一種變動的值，例如使用下列的類別建立物件的話，它會為每一個物件編一個連續的號碼：



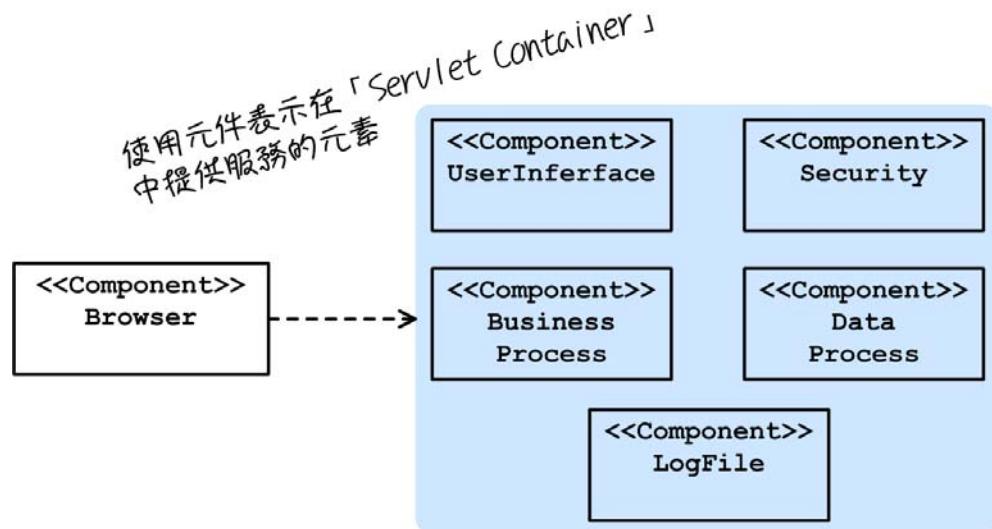
要呈現使用上列的類別建立三個物件以後的物件狀態，你可以結合類別節點和物件節點，讓物件圖型的資訊更容易瞭解：



5. 元件圖型

「元件圖型、Component Diagrams」運用一個最傳統的資訊技術概念來顯示軟體系統的資訊，那就是「模組化、Modularity」。模組化使用抽象的概念，把軟體系統的元素集合成「元件、Components」，在元件圖型裡，就可以顯示出軟體系統中的元件和元件之間的關聯，以元件為基礎來瞭解軟體系統的架構，會比使用類別圖型來得容易。

使用元件的概念來瞭解軟體系統的架構，在 Java 企業版中以「Container Base」為主的技術，更加明顯。以提供動態網頁技術的「Servlet Container」來說，在 Container 中可能會有上百個 JSP 或 Servlet，如果以元件的形式來呈現的話，會像這樣：



也因為元件可以把許多實體元素結合成一個邏輯單位，所以元件通常會使用在後面會討論到的「佈署圖型、Deployment Diagrams」。

元件

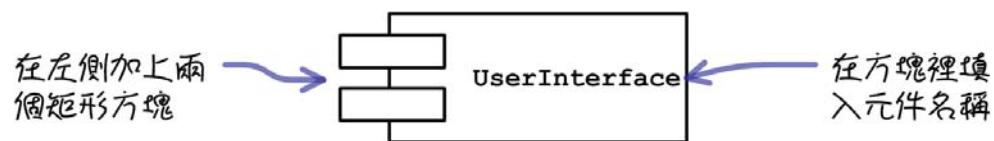
元件在軟體系統中是一個按照固定規則來定義系統功能的元素。你通常會因為下列的原因來架構元件和元件之間的關聯：

- 使用者可以預先瞭解系統功能的架構
- 提供軟體系統功能的邏輯性文件
- 提供更良好的封裝
- 方便取代與重複使用

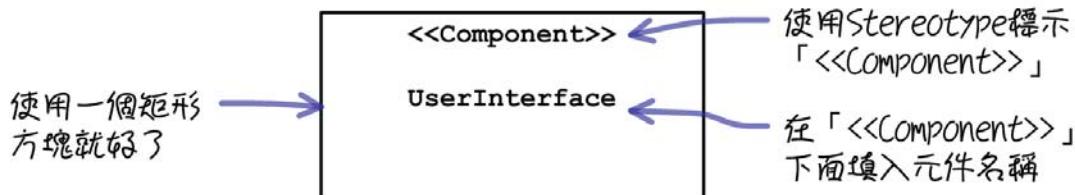
UML 對於元件的定義比較寬鬆，一個文字檔、資料庫表格、可執行檔、文件或是程式庫，都可以當成元件。目前比較新的定義是把元件分成三個種類：

- 佈署元件
- 工作產物元件
- 可執行元件

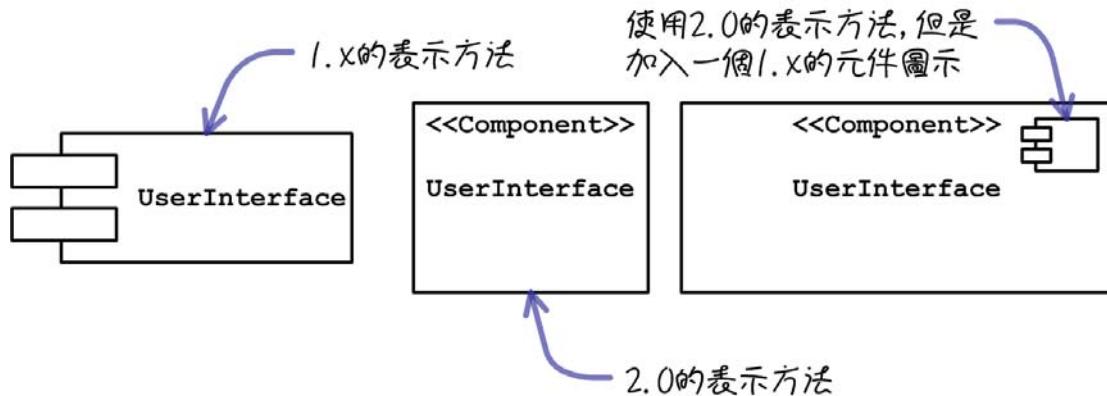
元件標記在 UML 1.x 使用下列的樣式：



這樣的元件標記表示方式會比較複雜一些。UML 2.0 在元件標記上有了很大變化：

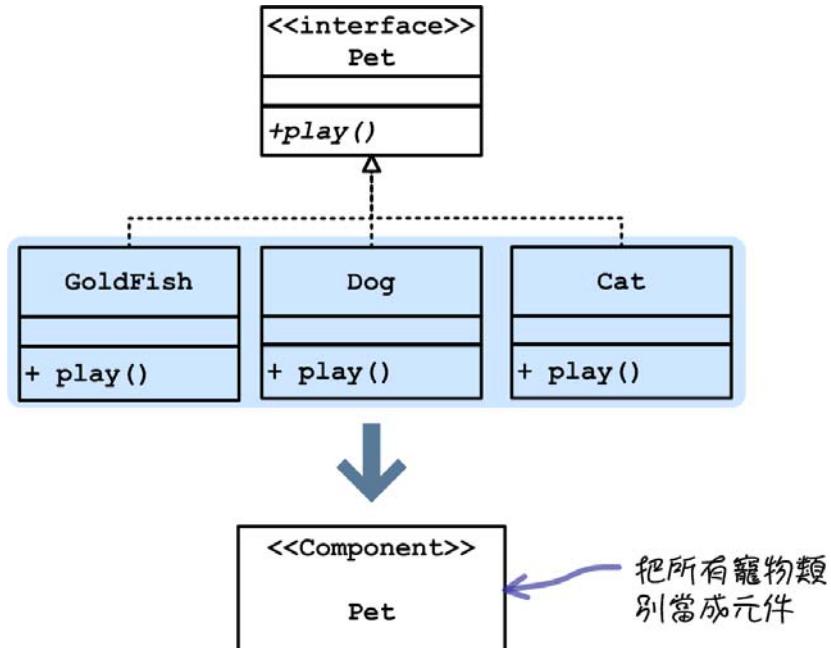


為了在 1.x 到 2.0 對於元件圖型的轉換，下列三種表示方法，都可以顯示一個元件標記：

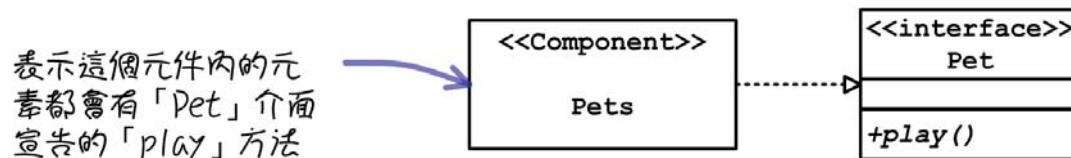


元件與介面

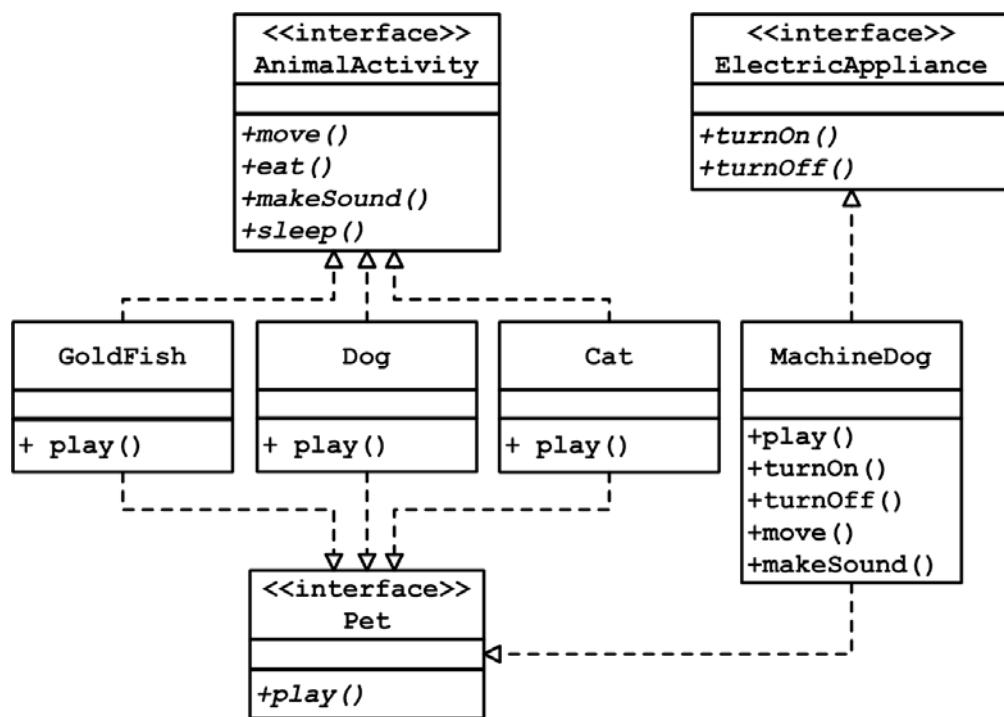
將所有實作一個介面的類別包裝為元件是一般常見的作法：



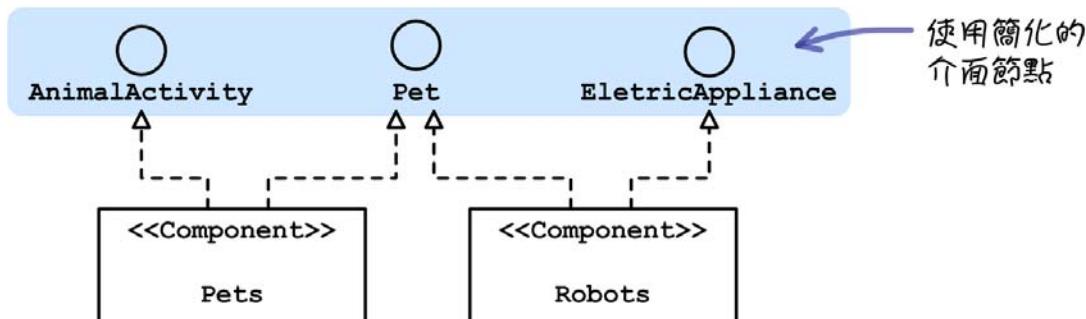
為了表示元件和介面之間的關聯，都會使用下列的方法來表示：



通常軟體系統內的類別架構都不會太簡單，下列是一個比較複雜的類別圖型：



將上列的類別圖型轉換為元件圖型時，你也可以使用簡化的介面節點：



使用元件圖型

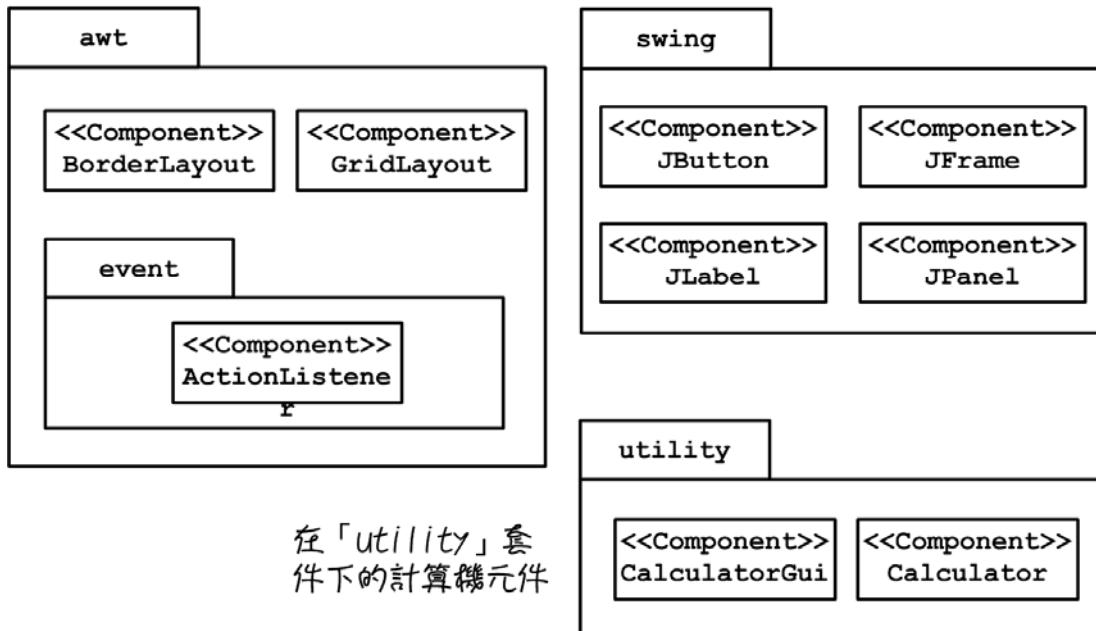
將複雜的類別圖型轉換成元件圖型，通常是瞭解軟體系統架構比較容易的方法。以下列的計算機模擬程式來說：



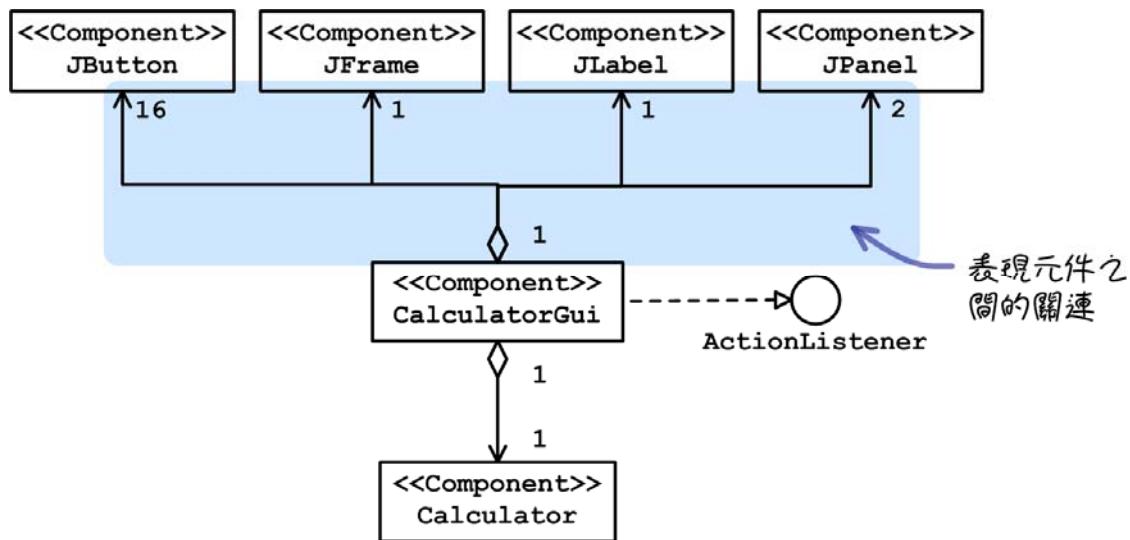
使用「swing」元件來架構計算機畫面與操作按鈕，按鈕元件內使用事件控制來處理操作

雖然這並不是一個很龐大的程式，但是加上使用的圖型相關套件以後，類別圖型會變得非常複雜。所以你可以先將整個架構轉換為以元件的形式來呈現：

將計算機使用到相關套件都以元件的方法顯示

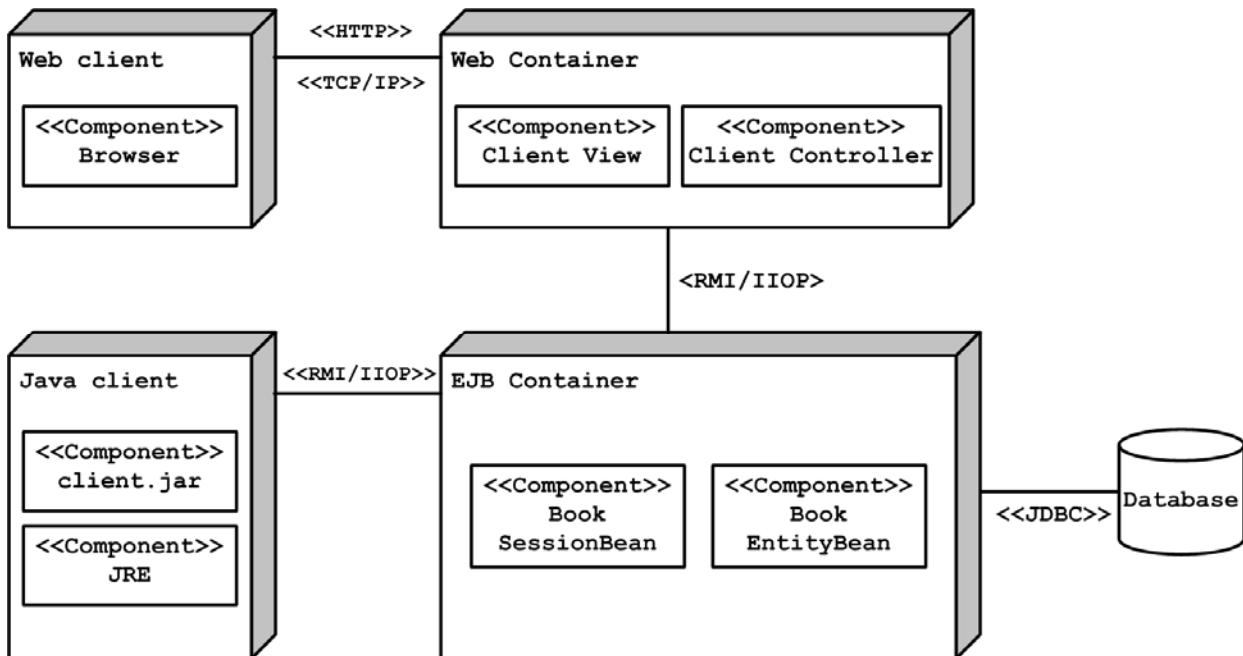


接下來使用元件圖型來顯示計算機程式的架構：



6. 佈署圖型

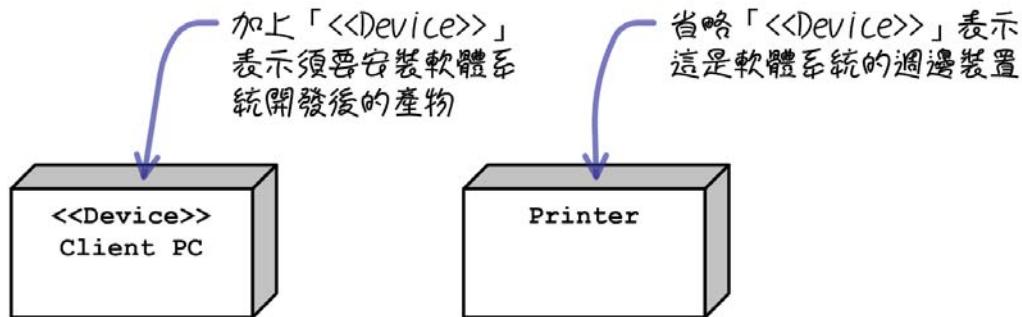
不論在軟體系統正在開發，或是開發完成後準備佈署到機器裝置，軟體系統都需要瞭解實體的裝置與資源，這些資訊通常對軟體系統有很大的影響。「佈署圖型、Deployment Diagrams」是 UML 圖型中用來顯示在軟體系統開發完成後，如何安裝到硬體的資訊，另外也可以包含合作的硬體和硬體之間的關聯。



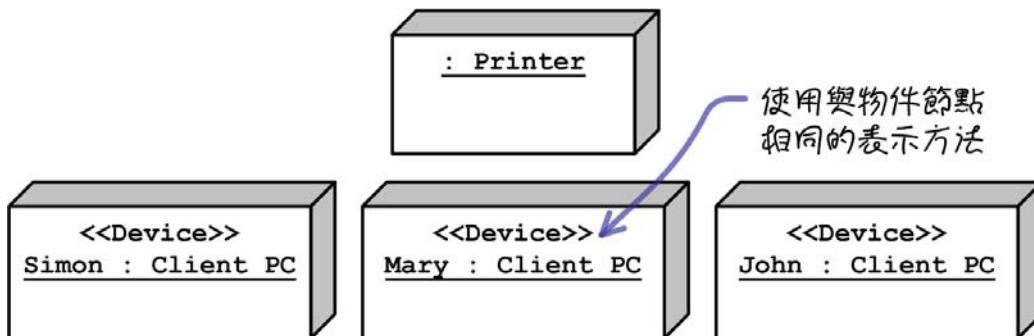
上列的佈署圖型顯示一個線上書局應用程式的佈署圖型，從圖型中可以看出來，這個應用程式可以有兩種用戶端，瀏覽器或一般的 Java 應用程式；提供服務的伺服器中也顯示出佈署的時候需要的元件；而資料庫的存取則統一由 Java 企業元件(Enterprise JavaBean)來負責。

節點

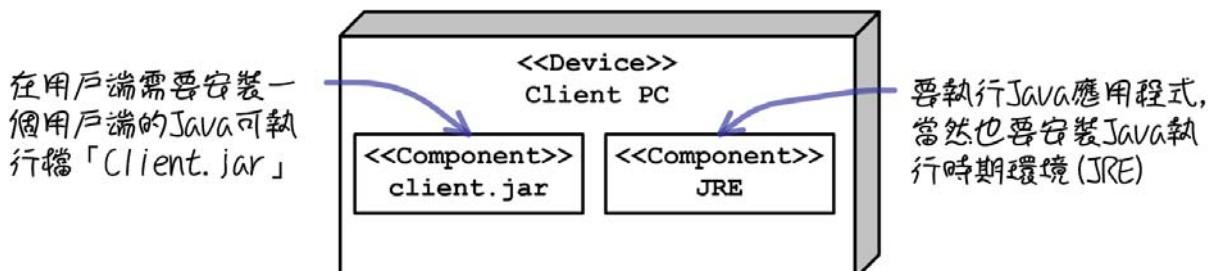
佈署圖型使用「硬體節點、Hardware node」來表示實體裝置，硬體節點的形狀是一個盒子方塊，依照會不會安裝軟體系統的產物分成下列兩種表示方法：



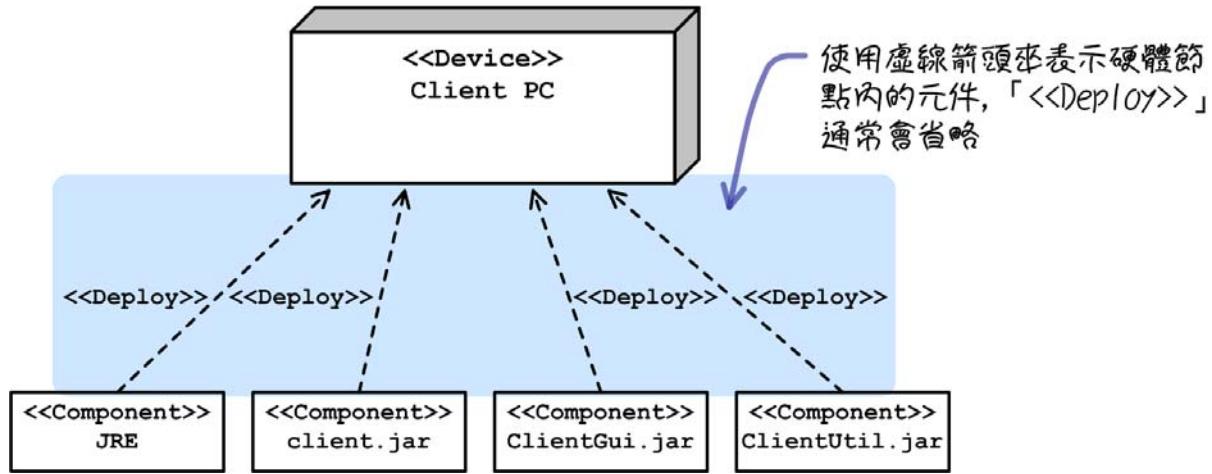
在使用硬體節點時，通常是一種分類別的概念，以上列的圖型來說，它顯示有兩種裝置，「Client PC」和「Printer」。如果你想要明確的顯示用戶端的實際裝置數量，可以使用「實體節點」的表示方法，它類似類別節點和物件節點之間的樣式：



在硬體節點中，最常放入的元素是元件節點。這些元件節點可以是軟體系統的產物或是其它需要的元件：

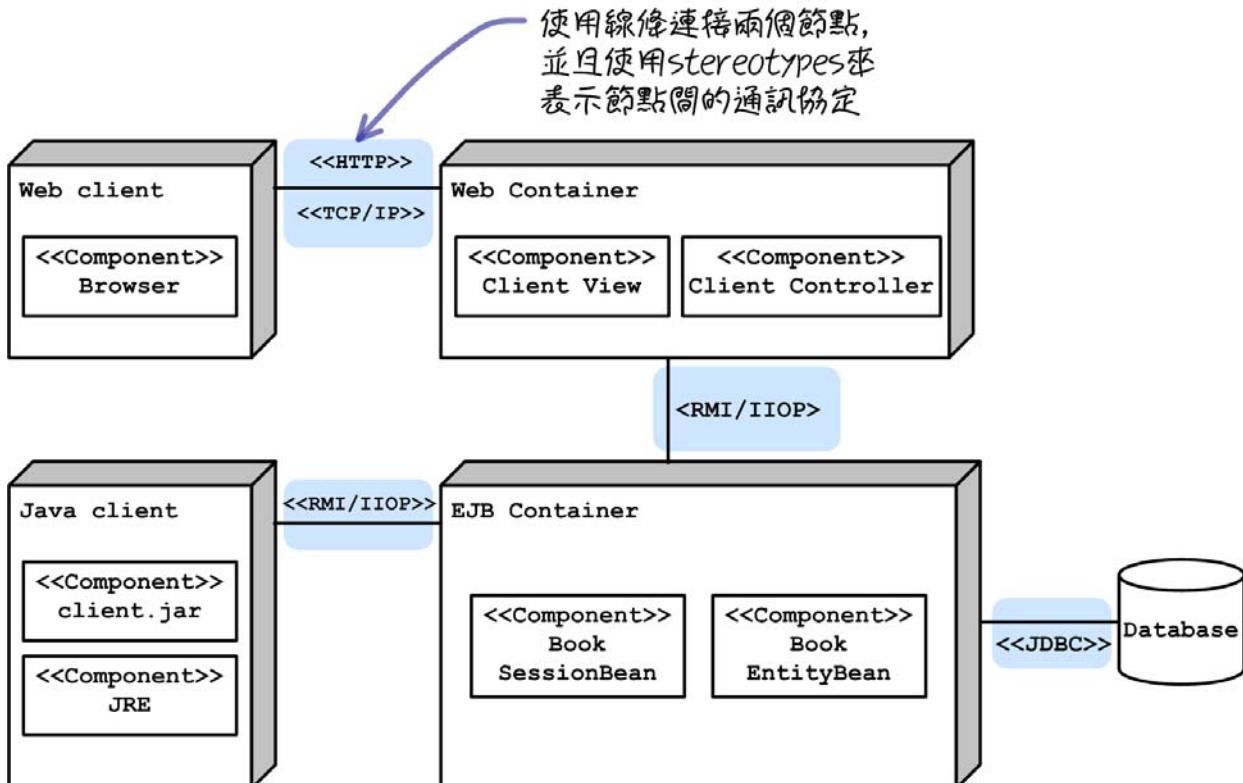


如果應體結點中的元件太多，你以也可以使用下列的表示方法：



關聯

在硬體節點之間通常會有實體的關聯，這些關聯就是硬體之間的通訊協定：

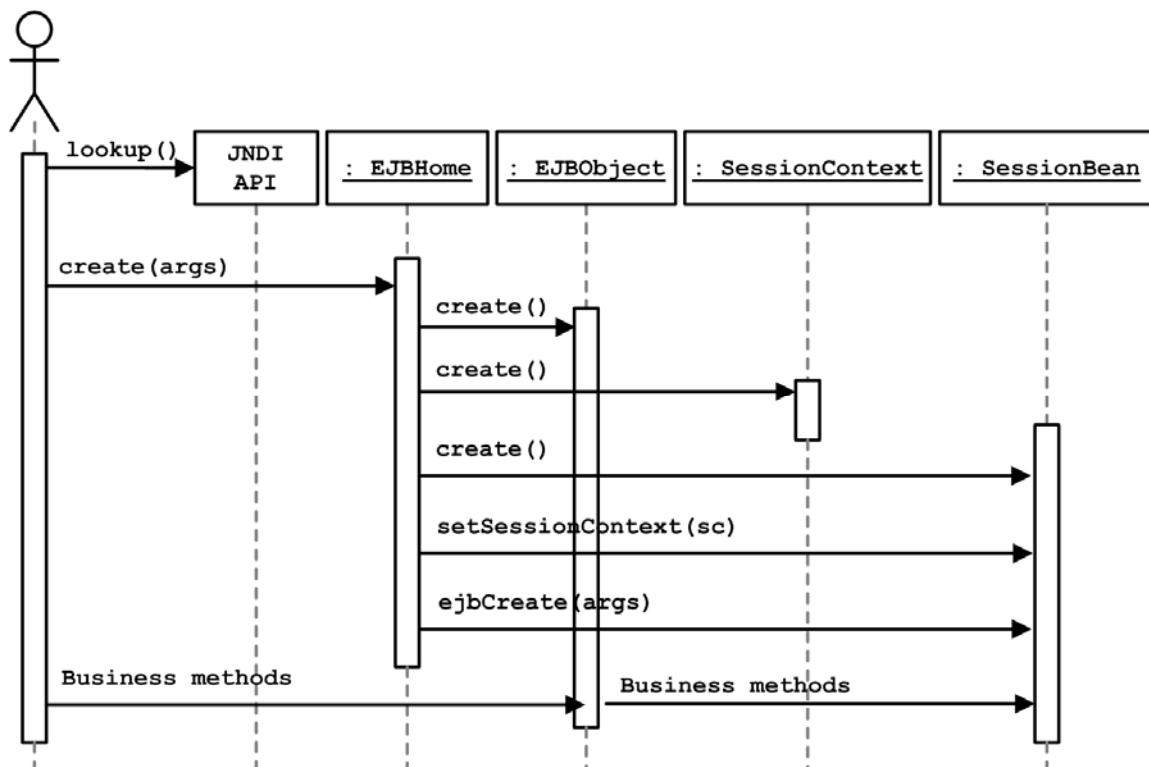


Memo

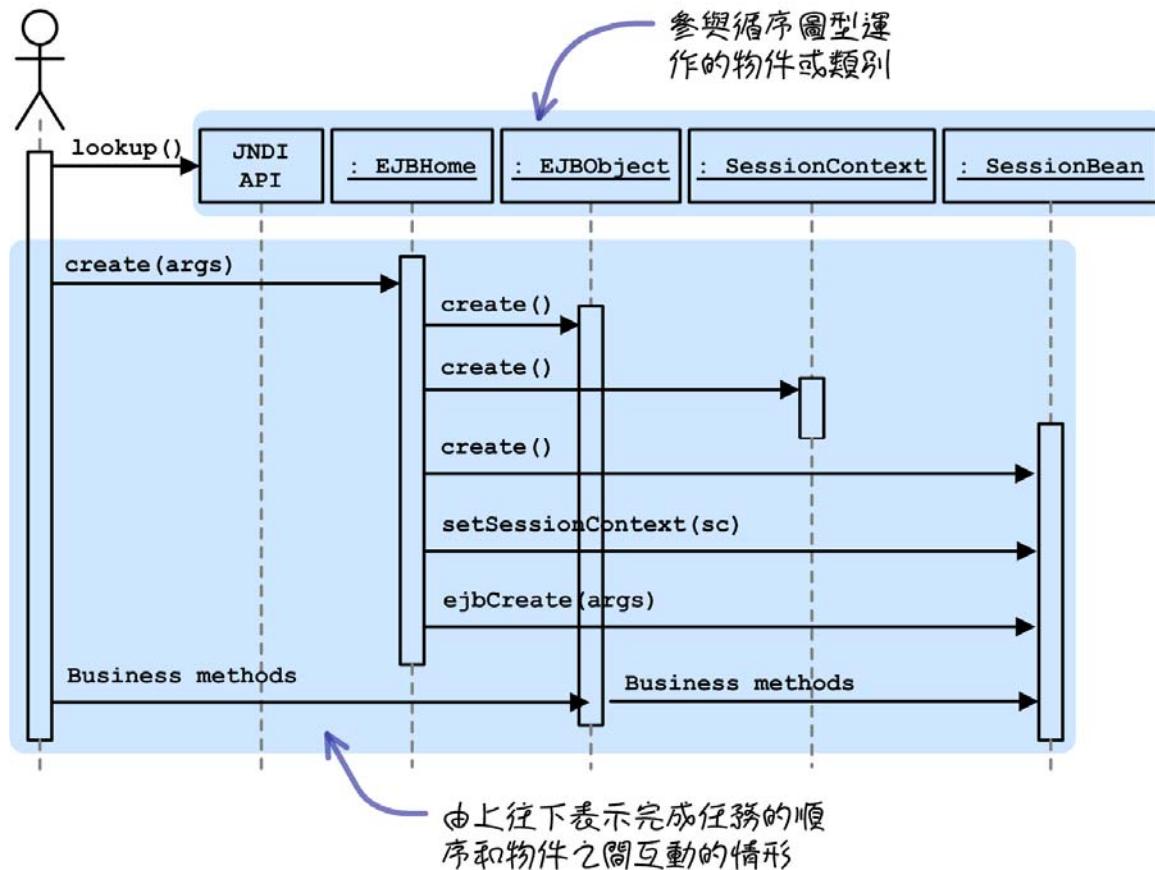
7. 循序圖型

「循序圖型、Sequence diagrams」應該是最常使用的動態模型。UML 制定的許多圖型，都是為了要讓複雜的軟體系統架構和行為更容易瞭解，循序圖型就是用來顯示軟體系統行為的動態圖型。

在傳統的軟體系統中，用來表示系統運作的圖型只有類似「活動圖型、Activity Diagrams」的流程圖。但是在物件導向的軟體系統中，傳統的流程圖已經無法表達系統的運作，尤其是在分散式的環境中，例如 Java 企業元件 (Enterprise JavaBean) 中的 Stateful Session Bean，使用循序圖型可以清楚的呈現出從建立元件到使用的過程：



建立循序圖型主要的基礎在「時間順序」和「合作物件」的概念上，你可以從圖型中清楚的瞭解有哪些物件在完成某項任務，更重要的是，圖型中可以顯示出完成這樣任務的先後順序：



元素

循序圖型由下列的基本元素構成：

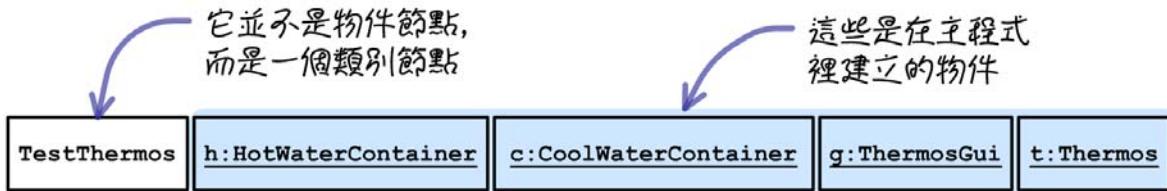
- 物件節點 (Object node)
- 生命線 (Lifeline)
- 活化區塊 (Activation box)
- 訊息 (Message)
- 內部訊息
- 解構物件
- 迴圈

物件節點

當你決定建立某項任務的循序圖型之後，要先將參與這項任務的物件找出來，這些物件就是你要放在循序圖型上的物件節點。以下列的範例程式來說，它是一個模擬開飲機的主程式：

```
public class TestThermos {  
    public static void main(String[] args) {  
        HotWaterContainer h = new HotWaterContainer(2);  
        CoolWaterContainer c = new CoolWaterContainer(50);  
        ThermosGui g = new ThermosGui();  
        Thermos t = new Thermos(h, c, g);  
    }  
}
```

這個包含有程式進入點的類別，建立了四個主要的開飲機類別物件，包含這個主程式，這些就是你要在循序圖型上呈現出來的物件節點：



在上圖的範例中，除了「`TestThermos`」以外，全部都是物件節點，這也是一般比較常見的情況，在循序圖型裡大都是使用物件節點。但是依照上列範例程式來看，「`TestThermos`」只是一個主程式，並沒有建立物件，所以使用類別節點來表示。在這種情況下，如果類別名稱相對來說不是很重要的話，會用「使用案例圖型」中的行爲者來代替：



不過在下列的情形下，你還是得使用類別節點。在模擬開飲機的程式中，有一個將冷水加熱為開水的方法：

```

public class Heater {
    public static int brew(int water) {
        try {
            Thread.sleep(100);
        } catch (Exception ex) { }
        return water;
    }
}

```

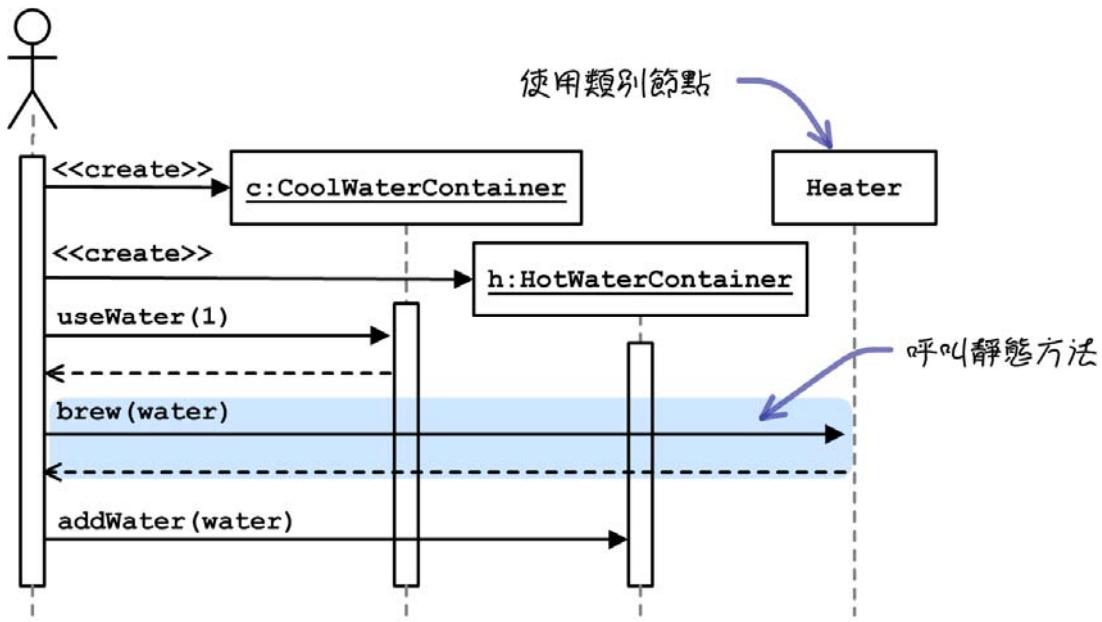
如果你需要在循序圖型裡加入這個動作的話，因為不必建立「Heater」類別的物件就可以呼叫「brew」方法，所以會在循序圖型裡使用類別節點。以下列的敘述來說：

```

HotWaterContainer h = new HotWaterContainer(2);
CoolWaterContainer c = new
CoolWaterContainer(50);
int water = c.useWater(1);
h.addWater( Heater.brew(water) );
    
```

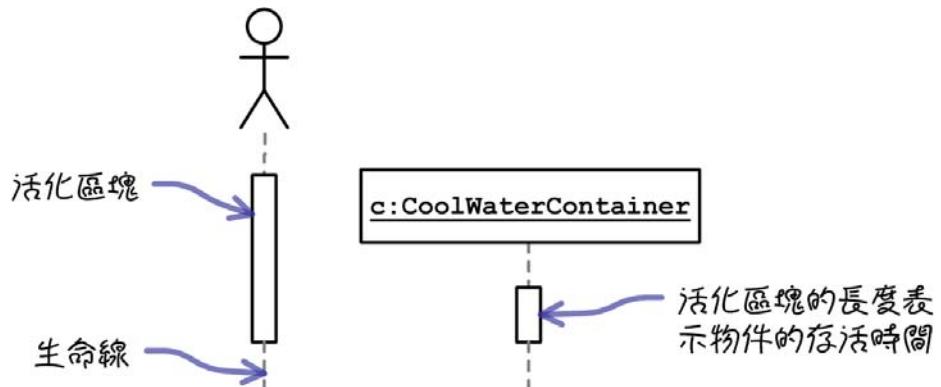
呼叫「Heater」類別
的靜態方法「brew」

顯示將冷水燒成開水任務的循序圖型會像這樣：



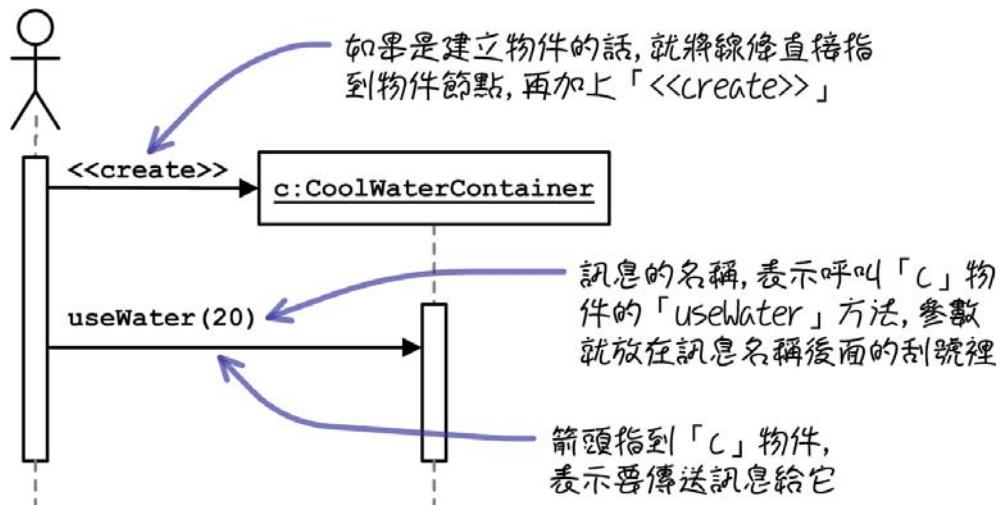
生命線與活化區塊

循序圖型裡使用生命線來表示物件的時間順序，它的樣式是一條連接物件節點或使用者的虛線。活化區塊指的是物件存活的時間，使用一個條狀的方塊放在生命線上，表示這個物件存在的時間範圍：



訊息

循序圖型裡使用「訊息」來表示物件之間的互動，訊息是一個從一個物件到另一個物件的箭頭，在 Java 的程式碼中，通常是代表方法的呼叫或使用建構式：



上列圖型對應的程式碼會像這樣：

```
CoolWaterContainer c = new CoolWaterContainer(50);
c.addWater(20);
```

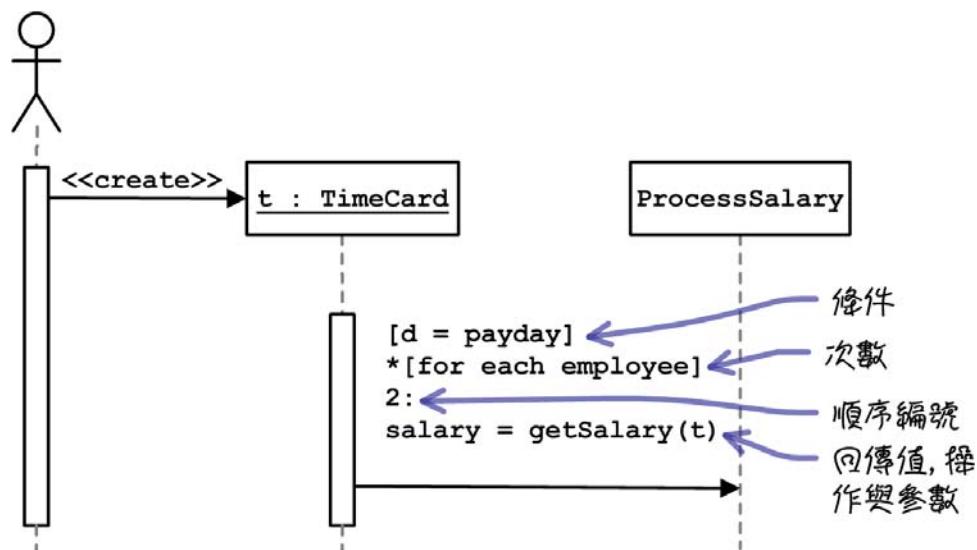
在物件之間的訊息傳遞，使用下列的語法來宣告：

[[<條件>]] [[*<重複>]] [<序號> :] <回傳值> := <操作名稱>(<參數>)

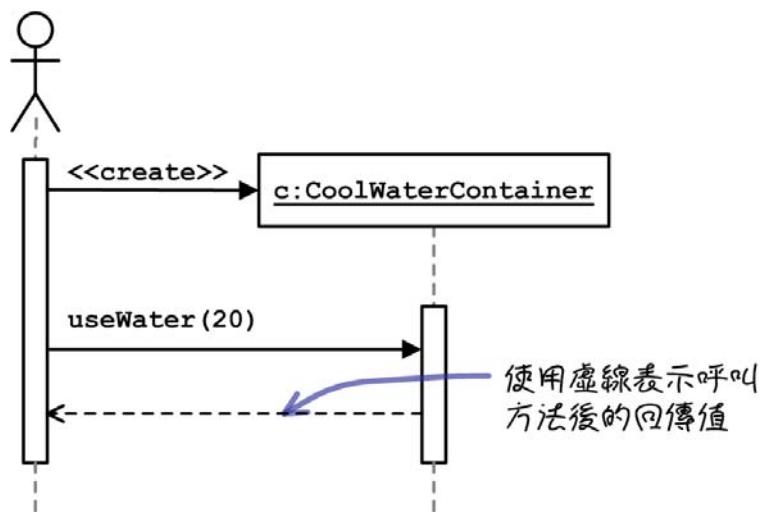
在上列的語法中：

- ▣ 條件：可選擇性的宣告。表示必需滿足指定的條件才會送出這個訊息，如果要填入指定的條件，必須放在「[]」裡面。
- ▣ 重複：可選擇性的宣告。表示送出訊息的次數。
- ▣ 序號：可選擇性的宣告。表示在任務中傳送訊息的順序編號，如果指定了順序編號，要在編號後面加上冒號「:」。
- ▣ 回傳值：可選擇性的宣告。表示接收操作的回傳值。
- ▣ 操作名稱：通常是方法的名稱。
- ▣ 參數：可選擇性的宣告。傳遞給操作的參數，有多個參數時，使用逗號分開。

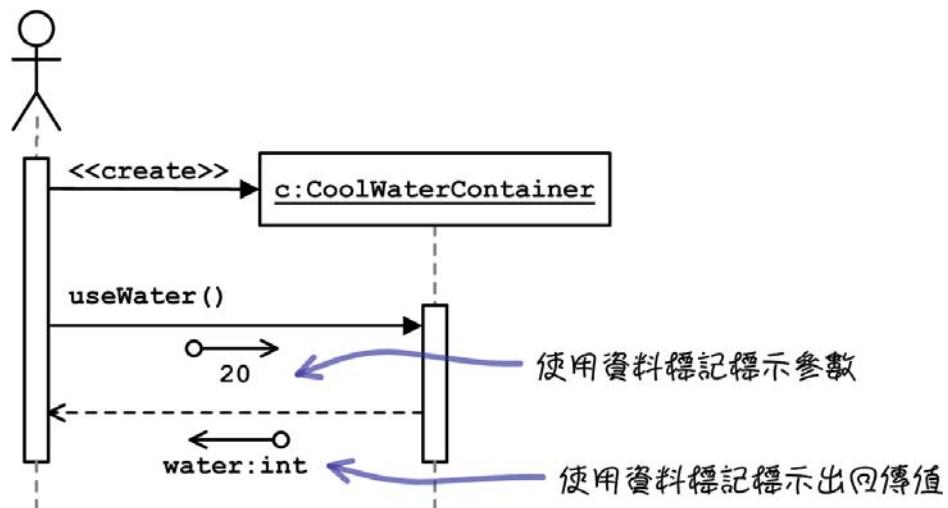
在循序圖型裡通常只會使用操作名稱和參數。下列的圖型是使用其它選項的範例：



如果不在訊息傳遞時指定操作的回傳值，也可以使用下列圖型中的虛線線條來表示執行某項操作後的回傳值。這種表示方法的好處是從圖型上來看會比在訊息傳遞時指定操作的回傳值清楚一些：



在訊息參數和回傳值方面，除了上列的表示方法外，還可以使用「資料標記、Data tokens」的表示方法：



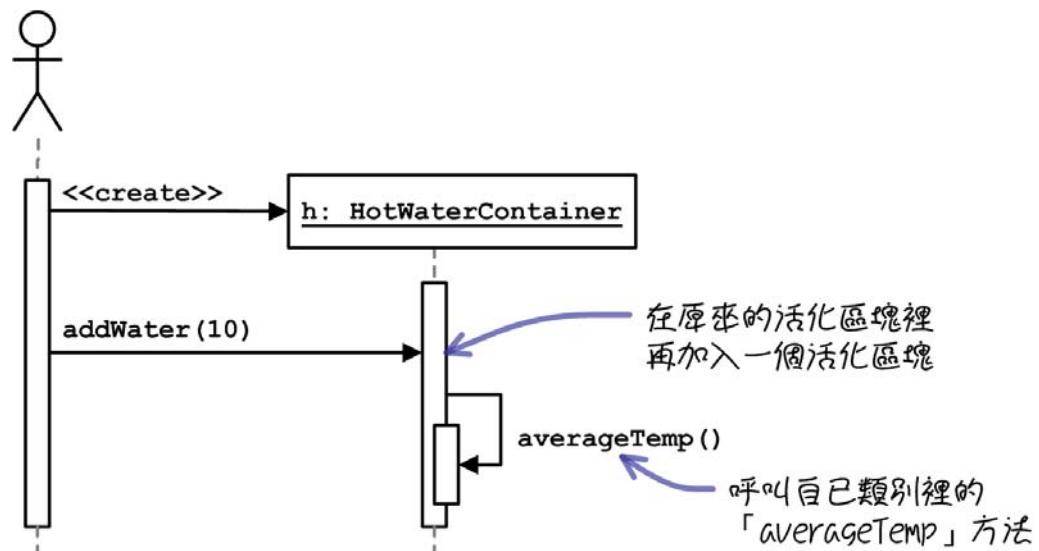
內部訊息

上列的討論中，都是不同物件之間的互動，在很多情況下也會呼叫同一個物件的方法，這在循序圖型裡稱為「內部訊息」。例如下列的敘述：

```
HotWaterContainer h = new HotWaterContainer(2);
h.addWater(10);

public class HotWaterContainer extends WaterContainer {
    ...
    public synchronized void addWater(int water) {
        temperature = averageTemp(super.getCapacity(), temperature,
                                   water, 100);
        super.addWater(water);
    }
    private int averageTemp(int oldCap, int oldTemp,
                           int newCap, int newTemp) {
        ...
    }
    ...
}
```

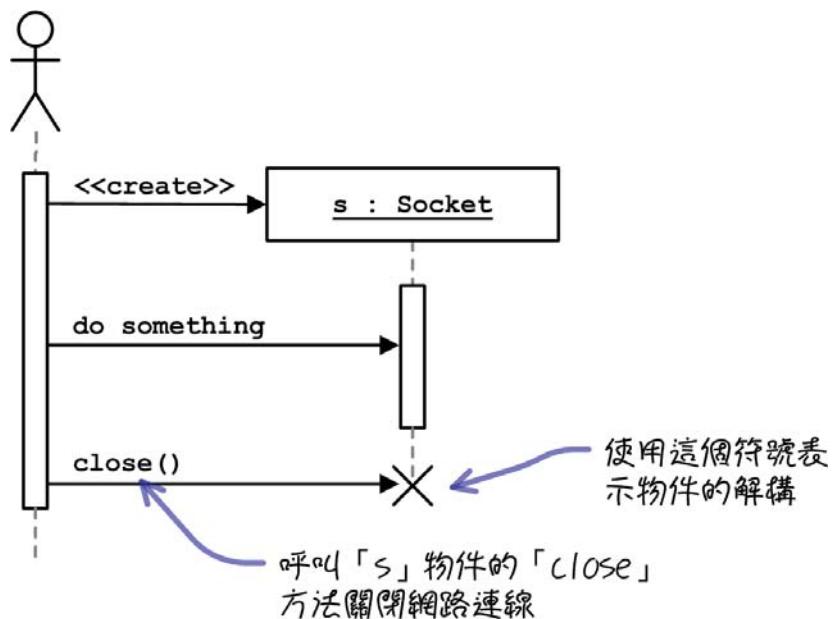
循序圖型使用下列的表示方法來顯示這類的實作：



解構物件

一般來說，使用 Java 程式語言，並不會特別去處理物件的「解構、deconstruct」；在循序圖型裡使用一個「X」符號來表示物件的解構，這對其它的程式語言可能很重要，從圖型裡可以讓程式設計師知道什麼時候該把物件給釋放掉。

Java 程式語言裡雖然不會特別去使用這個特性，可是你可以把這個特性應用在資源的釋放，例如串流或網路連線的關閉，這對 Java 程式設計師來說，與其它的程式語言相對於物件解構是一樣重要的：



上列圖型對應的程式碼會像這樣：

```

Socket s = new Socket(remoteHost, 8000);
// do something
s.close();

```

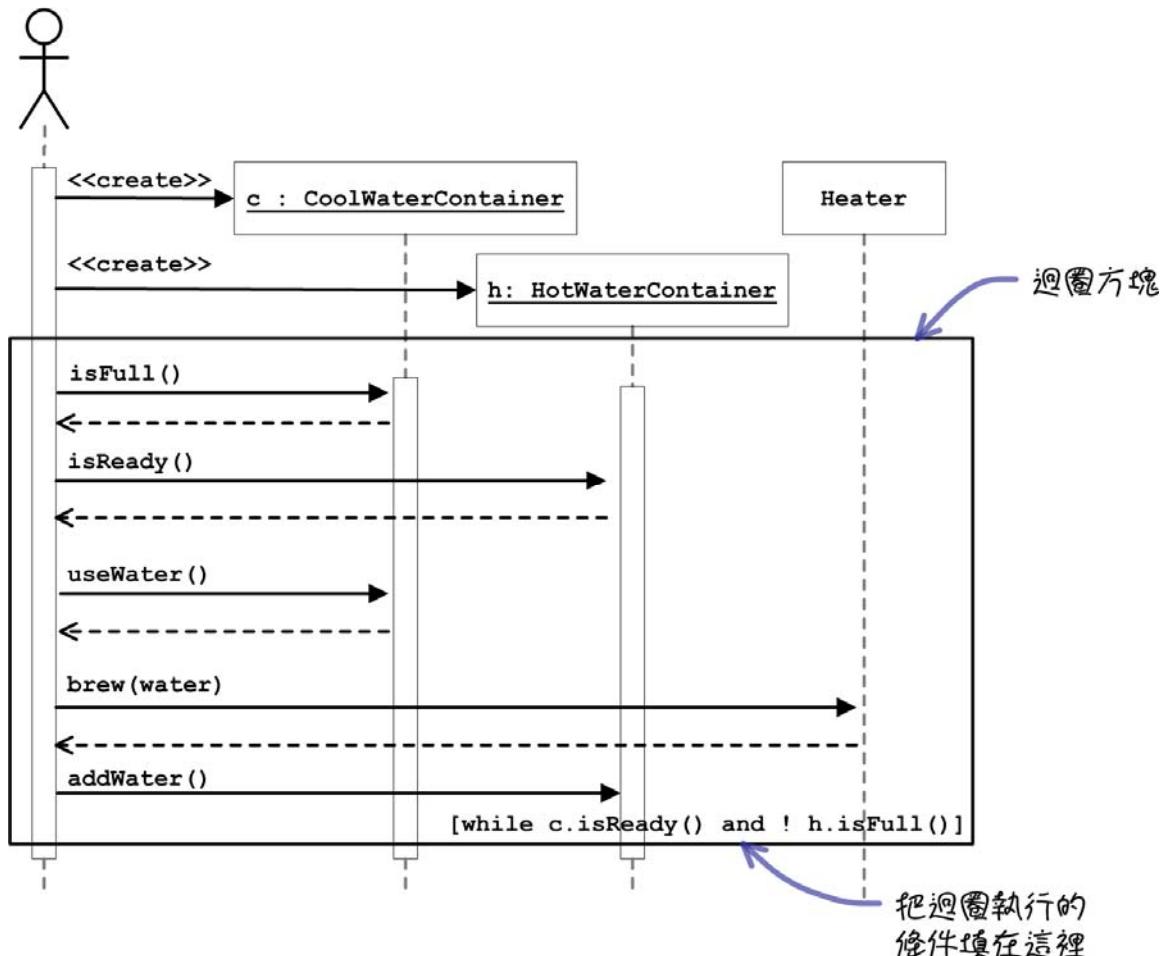
迴圈

在程式的實作上，反覆執行某一些工作的情形非常多，以下列的敘述來說：

```
HotWaterContainer h = new HotWaterContainer(2);
CoolWaterContainer c = new CoolWaterContainer(50);
while (c.isReady() && !h.isFull()) {
    int water = c.useWater(1);
    h.addWater( Heater.brew(water) );
}
```

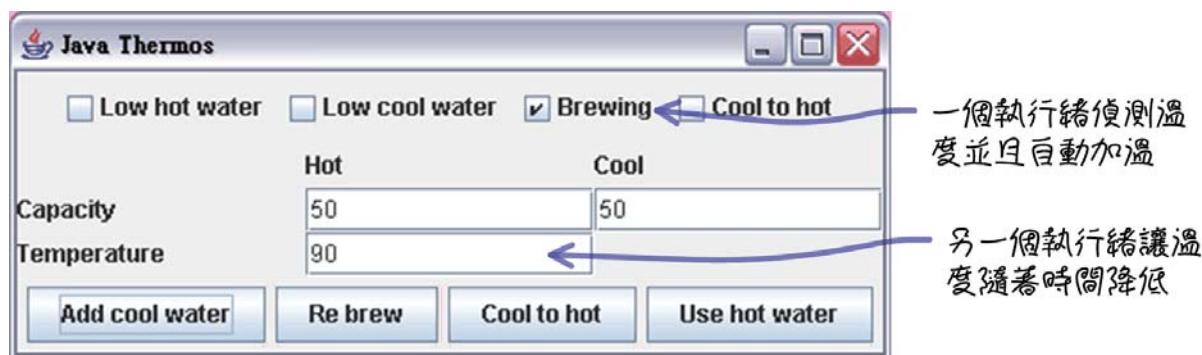
使用迴圈把冷水煮成開水，一直到冷水沒了或是開水滿了

上列的範例程式，包含了一組重複執行的訊息，所以不適合使用訊息宣告語法中的重複選項。不過你可以使用循序圖型中的「迴圈方塊」表示方法：

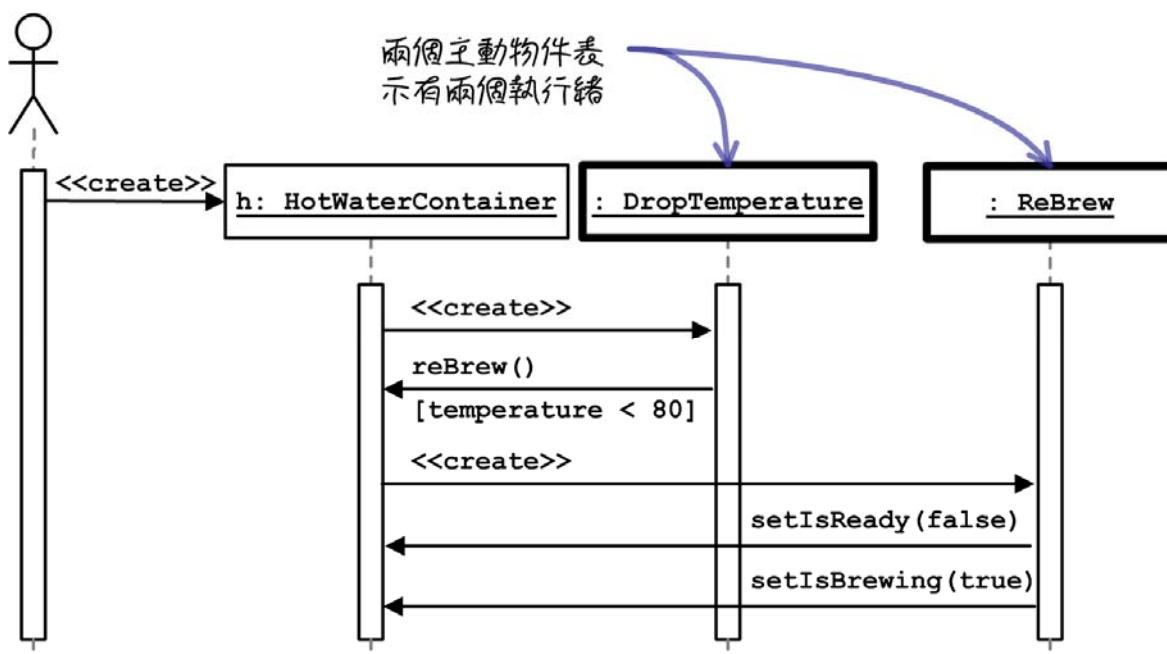


多執行緒

多執行緒的應用程式一向是最難以理解的，尤其是在發生錯誤的時候，想要從程式碼去排除錯誤會是比較困難的。除了在靜態圖型中可以看出誰是具有多執行緒特性的物件以外，循序圖型使用「主動物件、Active objects」來加入圖型中，讓執行緒與其它物件的互動可以更清楚。在開飲機的模擬程式裡，使用了好幾個多執行緒物件來模擬真實的開飲機。例如開水容器中的溫度會隨著時間慢慢降低(每秒中降低一度)，當溫度降到 80 度以下時，開飲機會自動將開水加溫到 100 度，這些動作都是使用執行緒在背景中偵測和執行：



你可以使用下列的圖型來表現上列圖中的動作：



上列圖形中的「DropTemperature」和「ReBrew」都是實作「Runnable」的類別，由「HotWaterContainer」物件來建立它們，一旦它們建立起來以後，就由自己控制執行緒。

```
public class DropTemperature implements Runnable {  
    public void run() {  
        ...  
    }  
}
```

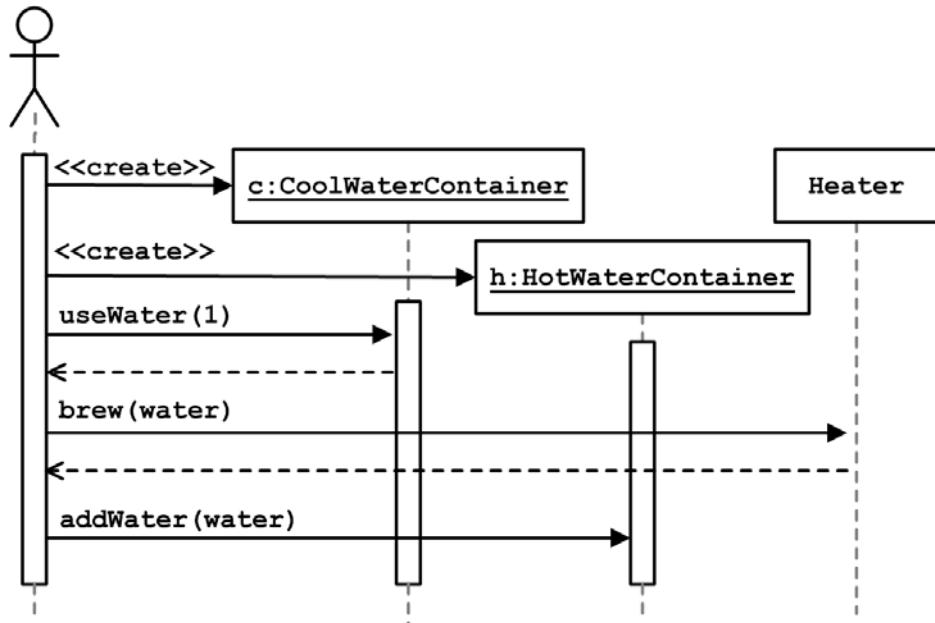
```
public class ReBrew implements Runnable {  
    public void run() {  
        ...  
    }  
}
```

Memo

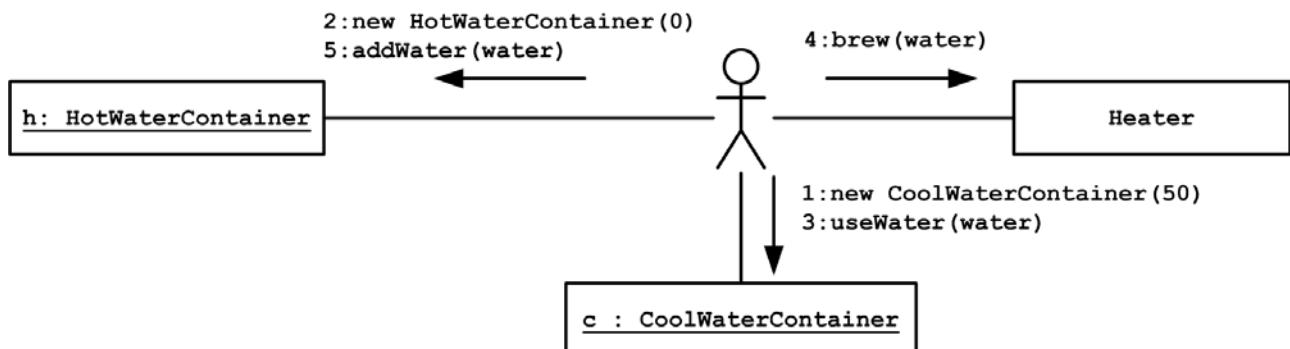
8. 合作圖型

「合作圖型、Collaboration Diagrams」與循序圖型所顯示的資訊，有部份是重複的，都是用來呈現物件在時間上的互動。但是循序圖型以時間順序來顯示物件的互動，而合作圖型則把焦點又拉回物件，以物件來顯示互動的順序。當你想要瞭解軟體系統中的關鍵物件，與其它物件一起完成一件任務的順序時，就要使用合作圖型。

下列是開飲機模擬程式中，將冷水燒為開水的循序圖型：



使用合作圖型來顯示的話，會像這樣：



元素

合作圖型由下列的基本元素構成：

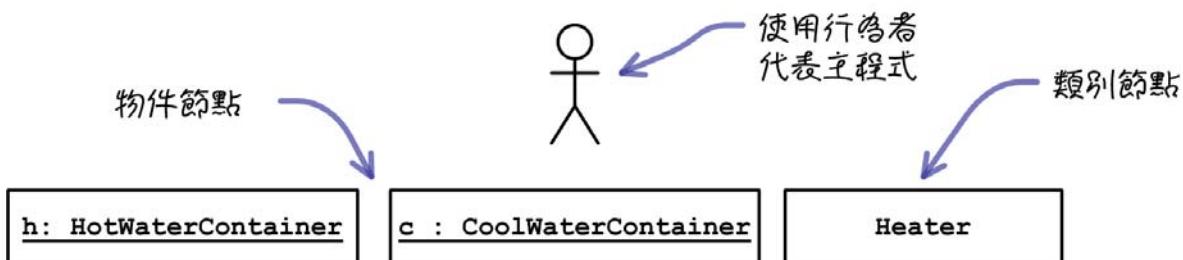
- 物件節點 (Object node)
- 連結 (Link)
- 訊息 (Message)

物件節點

與循序圖型一樣，當你決定建立某項任務的合作圖型之後，要先將參與這項任務的物件找出來。以下列的範例程式來說：

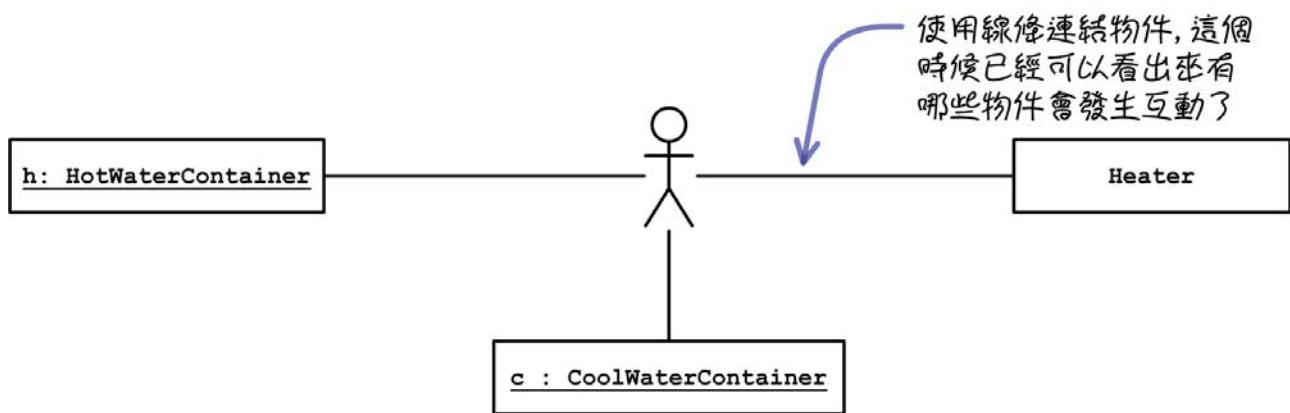
```
public class TestHeater {
    public static void main(String args[]) {
        CoolWaterContainer c = new CoolWaterContainer(50);
        HotWaterContainer h = new HotWaterContainer(2);
        int water = c.useWater(1);
        h.addWater(water);
    }
}
```

要使用合作圖型來顯示這個程式物件的互動，需要下列的元素：



連結

找出合作圖型中需要的物件節點後，將有互動的物件使用線條連結起來：



訊息

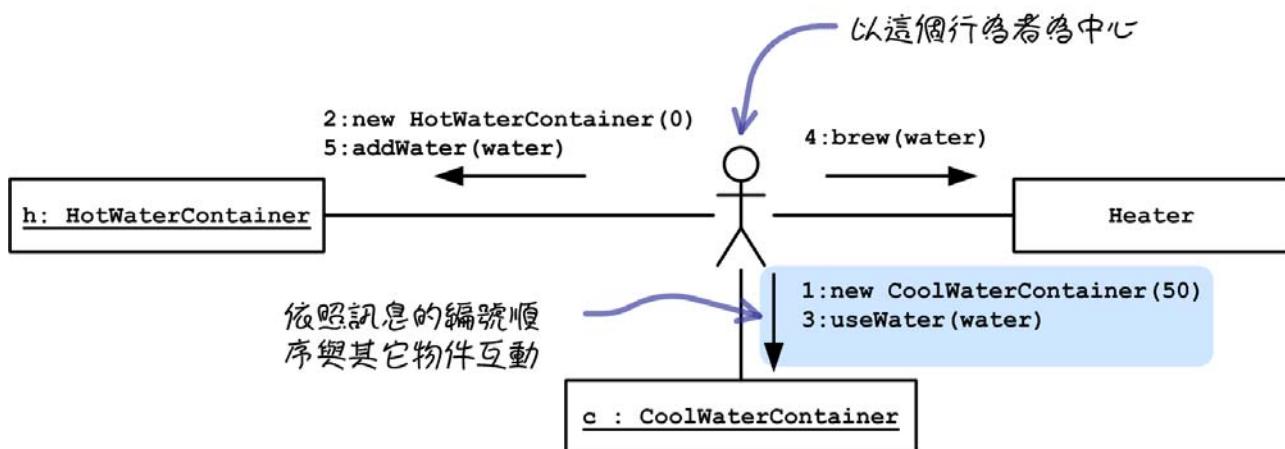
在物件之間的訊息傳遞，使用的宣告語法與循序圖型中的訊息一樣：

`[[<條件>]] [[*<重複>]] [<序號> :] <回傳值> := <操作名稱>(<參數>)`

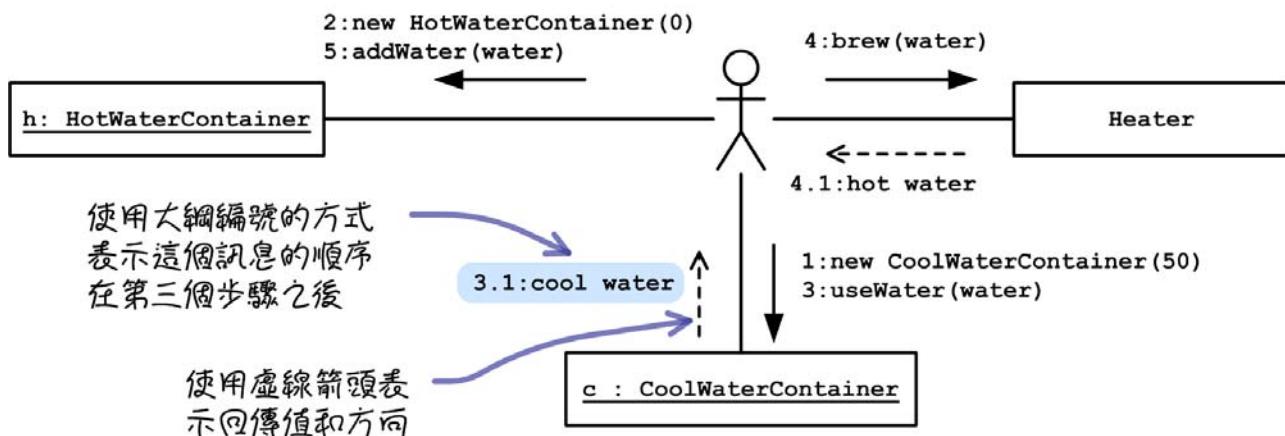
在上列的語法中：

- ▣ 條件：可選擇性的宣告。表示必需滿足指定的條件才會送出這個訊息，如果要填入指定的條件，必須放在「[]」裡面。
- ▣ 重複：可選擇性的宣告。表示送出訊息的次數。
- ▣ 序號：表示在任務中傳送訊息的順序編號，在順序編號的後面加上冒號「:」。
- ▣ 回傳值：可選擇性的宣告。表示接收操作的回傳值。
- ▣ 操作名稱：通常是方法的名稱。
- ▣ 參數：可選擇性的宣告。傳遞給操作的參數，有多個參數時，使用逗號分開。

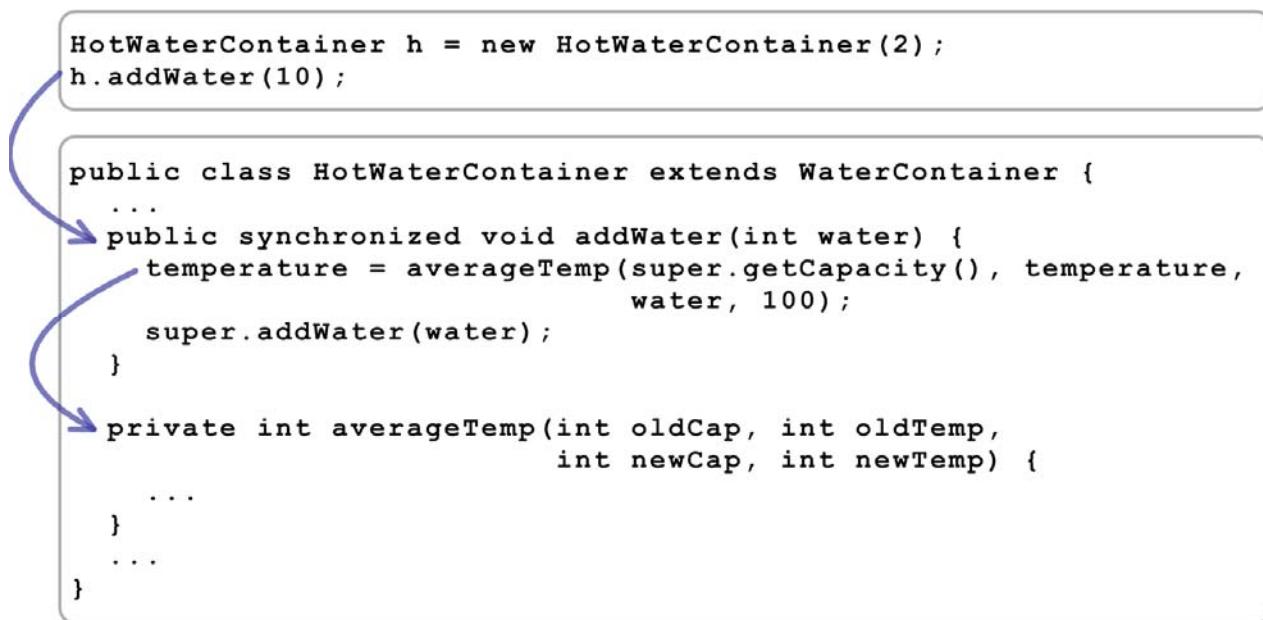
下列是加上訊息後的合作圖型：



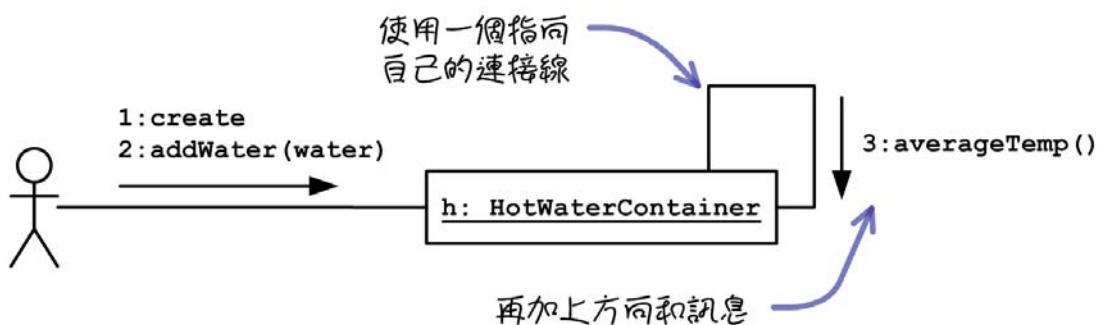
在上列的圖型中，第三個步驟的使用冷水和第四個步驟的從冷水燒成開水，這兩個互動都具有回傳值，可是在圖型裡並沒有顯示出來。你可以使用虛線箭頭來在合作圖型裡表示互動以後的回傳值：



上列的討論都是不同物件之間的互動，在很多情況下也會呼叫同一個物件內的方法，這在合作圖型裡稱為「內部訊息」。例如下列的敘述：



合作圖型使用下列的表示方法來顯示這類的實作：



上列的圖型顯示將開水加入開水的容器時，重新計算溫度的合作圖型。

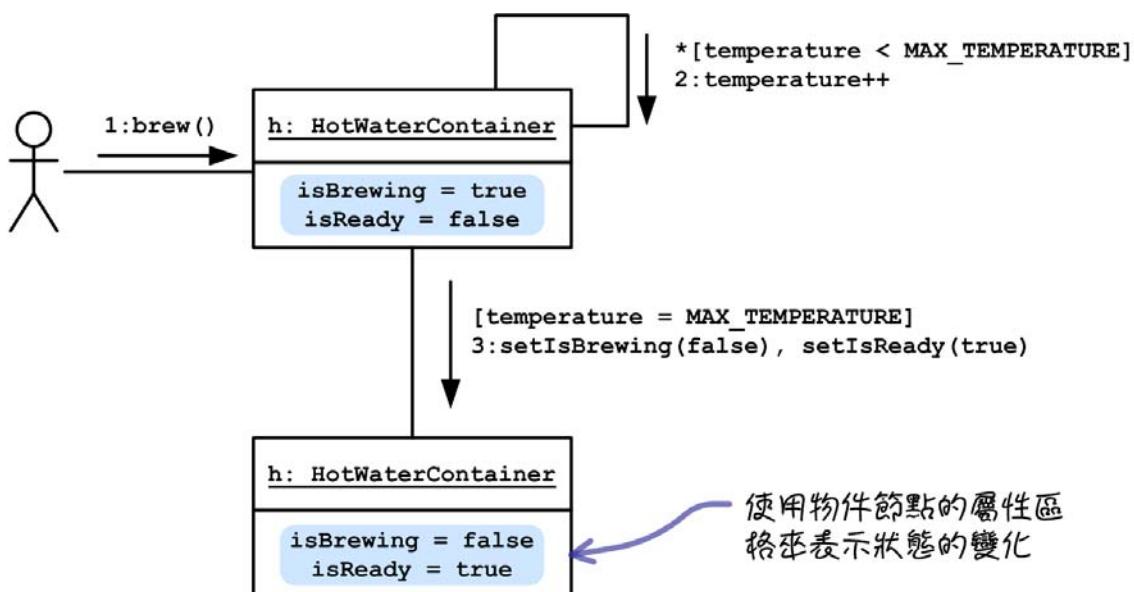
物件狀態的改變

在開飲機模擬程式中，提供了開水重新加熱的功能，如果使用這個功能，開飲機的狀態會設定為「加熱中」，而且開水在加熱中時不能使用。例如下列的程式敘述：

```
HotWaterContainer h = new HotWaterContainer(50);

h.reBrew(); ← 呼叫重新沸騰的方法
```

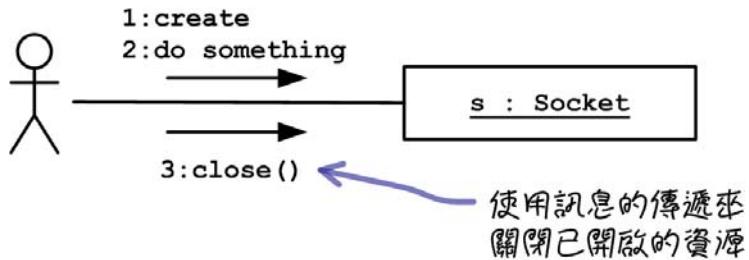
你可以使用下列的表示方法，來表示從開始加熱到加熱結束的合作圖型：



解構物件

一般來說，使用 Java 程式語言，並不會特別去處理物件的「解構、deconstruct」；你可在合作圖型裡使用訊息的傳遞來表示物件的解構，這對其它的程式語言可能很重要，從圖型裡可以讓程式設計師知道什麼時候該把物件給釋放掉。

Java 程式語言裡雖然不會特別去使用這個特性，可是你可以把這個特性應用在資源的釋放，例如串流或網路連線的關閉，這對 Java 程式設計師來說，與其它的程式語言相對於物件解構是一樣重要的：



上列圖型對應的程式碼會像這樣：

```
Socket s = new Socket(remoteHost, 8000);
// do something
s.close();
```

迴圈

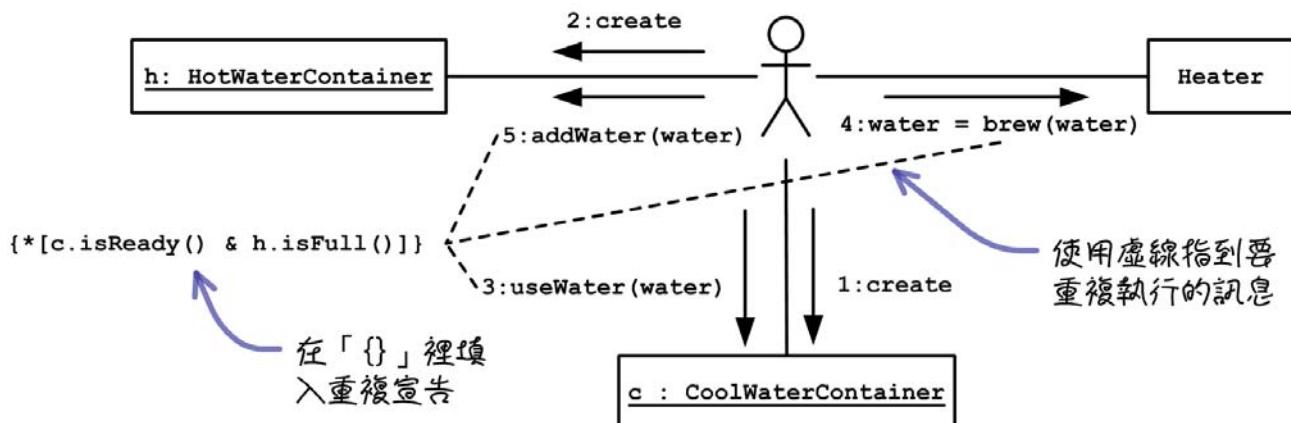
在程式的實作上，反覆執行某一些工作的情形非常多，以下列的敘述來說：

```
HotWaterContainer h = new HotWaterContainer(2);
CoolWaterContainer c = new CoolWaterContainer(50);

while (c.isReady() && !h.isFull()) {
    int water = c.useWater(1);
    h.addWater( Heater.brew(water) );
}
```

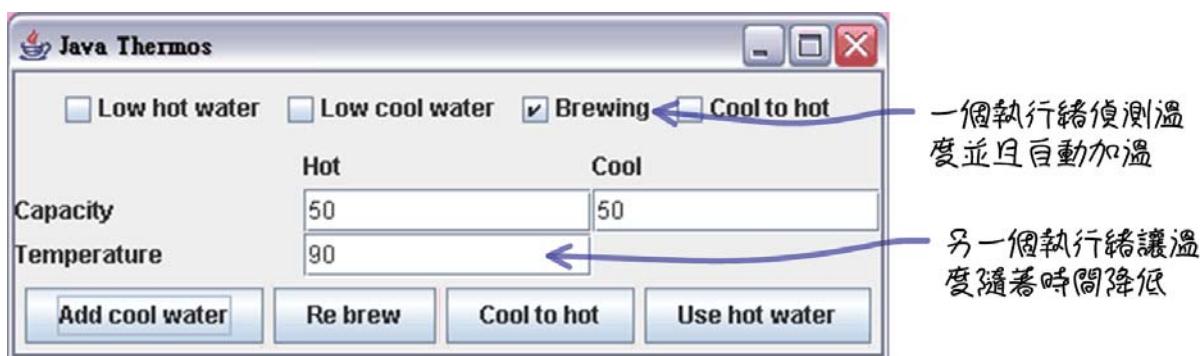
← 使用迴圈把冷水煮成開水，一直到冷水沒了或是開水滿了

上列的範例程式，包含了一組重複執行的訊息，所以不適合使用訊息宣告語法中的重複選項。不過你可以使用下列的表示方法：

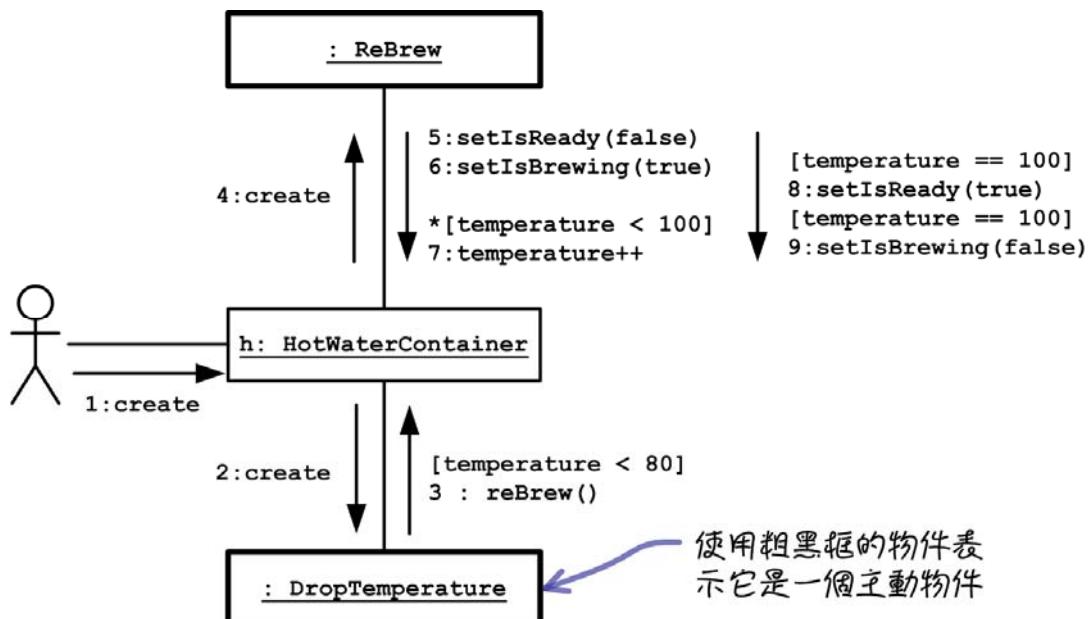


多執行緒

多執行緒的應用程式一向是最難以理解的，尤其是在發生錯誤的時候，想要從程式碼去排除錯誤會是比較困難的。除了在靜態圖型中可以看出誰是具有多執行緒特性的物件以外，合作圖型使用「主動物件、Active objects」來加入圖型中，讓執行緒與其它物件的互動可以更清楚。在開飲機的模擬程式裡，使用了好幾個多執行緒物件來模擬真實的開飲機。例如開水容器中的溫度會隨著時間慢慢降低(每秒中降低一度)，當溫度降到 80 度以下時，開飲機會自動將開水加溫到 100 度，這些動作都是使用執行緒在背景中偵測和執行：



你可以使用下列的圖型來表現上列圖中的動作：



上列圖形中的「DropTemperature」和「ReBrew」都是實作「Runnable」的類別，由「HotWaterContainer」物件來建立它們，一旦它們建立起來以後，就由自己控制執行緒。

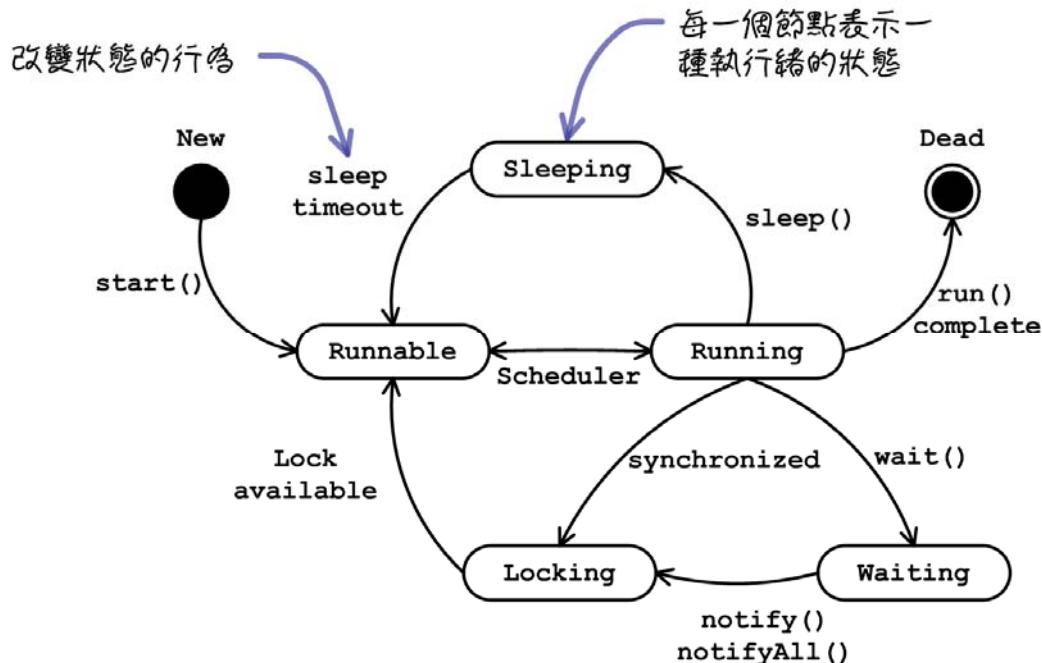
```
public class DropTemperature implements Runnable {  
    public void run() {  
        ...  
    }  
}
```

```
public class ReBrew implements Runnable {  
    public void run() {  
        ...  
    }  
}
```

9. 狀態圖型

「狀態圖型、Statechart diagrams」用來顯示軟體系統中特定的狀態情形，如果軟體系統中某項作業的生命週期是非常重要的，而且在生命週期當中，會變換不同的狀態時，你就需要使用狀態圖型，它可以確認邏輯的正確性和是不是有未考慮到的情況。

這裡提到的「狀態」，與物件圖型裡的「物件狀態」是不一樣的。在物件圖型的物件狀態指的是物件屬性的值，經由物件的屬性值來瞭解物件在某一個特定時間的狀態；而狀態圖型中的物件狀態有可能是物件屬性的值，但是在很多情況下，它就不是單純的物件屬性值。執行緒的狀態就是最好的例子，一個執行緒可能包含了許多共同合作的類別，它們的狀態是由執行緒的機制來控管生命週期：

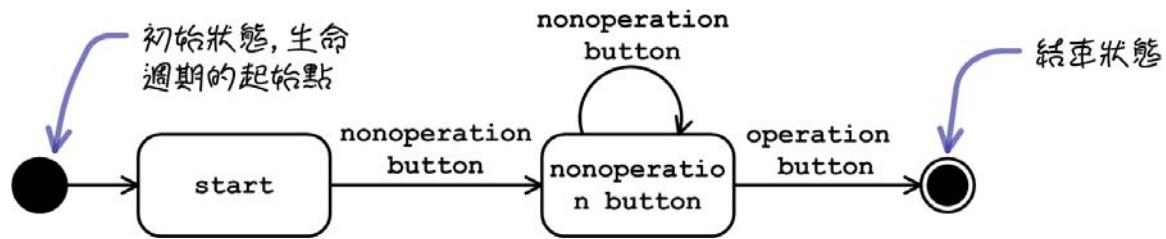


不過在使用狀態圖型前，有一個前提是一定要考慮的，就是這些生命週期必須是「有限的狀態」。這一點非常重要，因為你要排除所有具備不確定生命週期的情況，再根據實際的需要，畫出狀態圖型。

狀態節點

狀態圖型使用兩個特定的符號來表示生命週期的開始和結束：

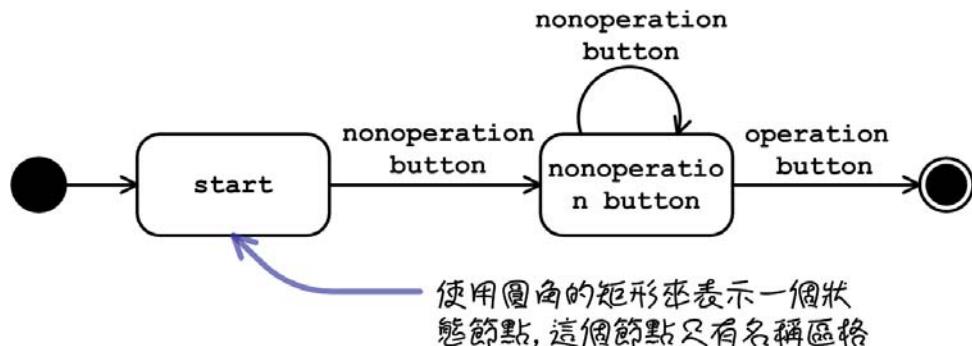
- 初始狀態 (Initial state)，使用實心黑色的圓型
- 結束狀態 (Final state)，使用實心黑色的圓型，外層包圍著空心的圓型



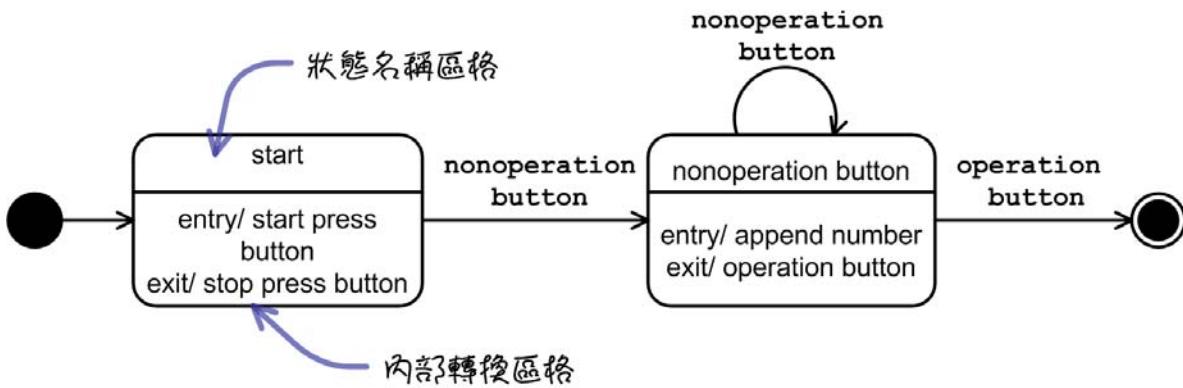
除了初始和結束節點外，狀態節點用來表示生命週期中的某一種狀態，它分成上、下兩個區格：

- 名稱區格 (Name compartment)
- 內部轉換區格 (Internal transitions compartment)

下列的狀態圖型中，都是省略內部轉換區格的狀態節點：



下列的狀態圖型中，是包含名稱區格和內部轉換區格的狀態節點：



名稱區格

名稱區格內的文字表示生命週期中的一個狀態，在 UML 的規格中並沒有規定一定要填寫，如果一個狀態節點沒有狀態名稱，就稱為「匿名狀態」，一般來說比較不會遇到這樣的情形。

內部轉換區格

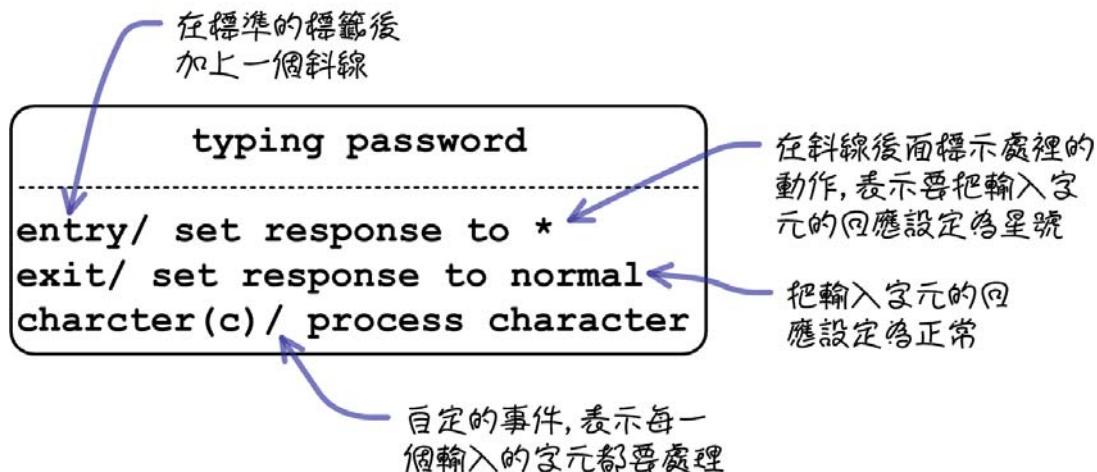
內部轉換區格用來表示狀態節點內部的轉換狀況，使用四種標籤來表示從進入到離開狀態節點之間作了哪些動作：

標籤	說明
entry	進入狀態節點時的動作
exit	離開狀態節點時的動作
do	停留在這個狀態節點時執行的動作

除了這三種 UML 規定的標籤外，你可以使用下列的格式定義自己的動作：

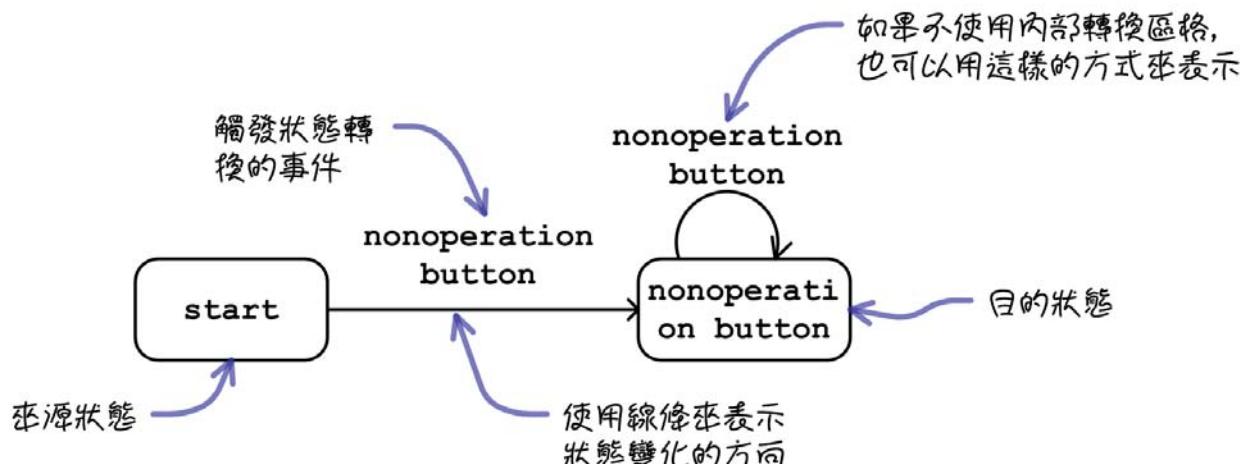
自定的事件名稱 → <事件名稱>(<參數>) / <動作>
 執行這個事件需要的參數 ↗ 這個事件執行的動作

下列的狀態節點，用來表示在輸入密碼欄位時的狀態，使用內部轉換區格來顯示使用者在輸入密碼時的動作：



轉換

在狀態圖型中，兩個狀態節點之間的標示稱為「轉換、transition」，用來表示原始狀態如何轉換到目的狀態：



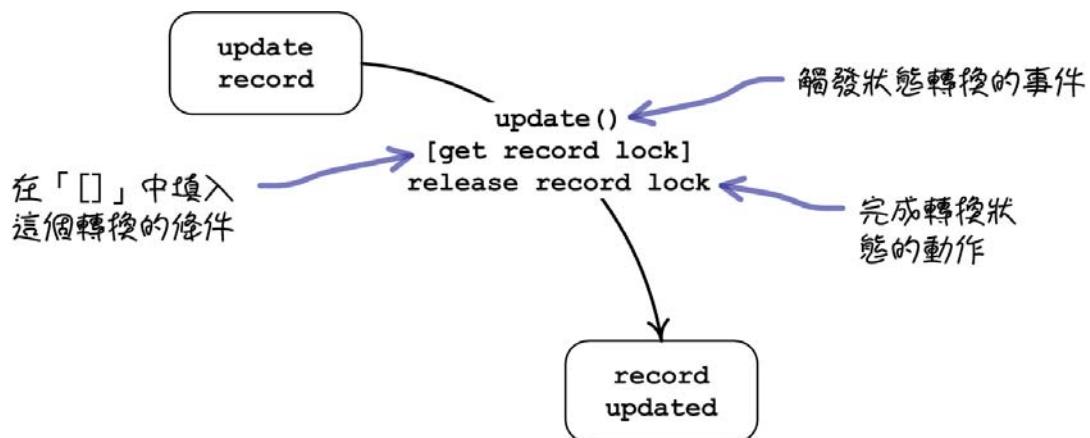
導致狀態轉換的事件，使用下列的語法來宣告：

<事件名稱> ([<參數>]) [[條件]]

在上列宣告事件的語法中：

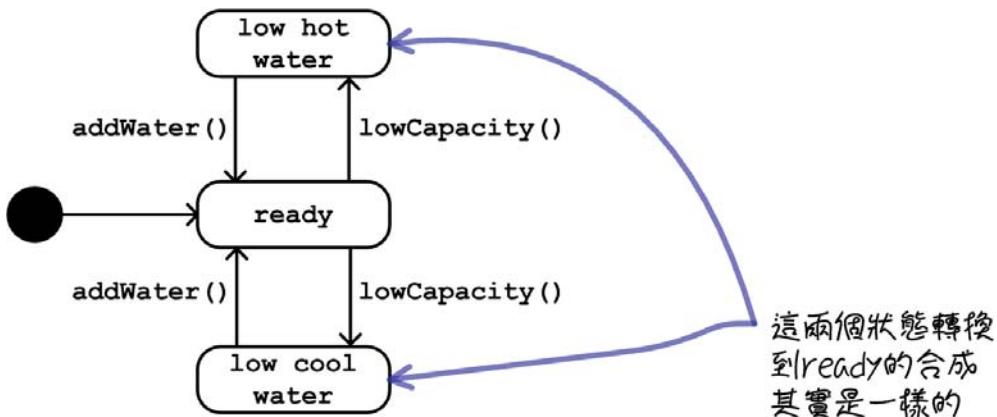
- 事件名稱：通常會是方法的名稱。
- 參數：可選擇性的宣告。傳遞給事件的參數。
- 條件：可選擇性的宣告。表示導致狀態轉換的條件。

一個造成狀態改變的事件，可能會非常複雜，例如修改資料表格中的某一筆資料，在事件裡還會有資料鎖定、修改與釋放鎖定的動作，你可以使用條件和動作來說明比較複雜的事件：

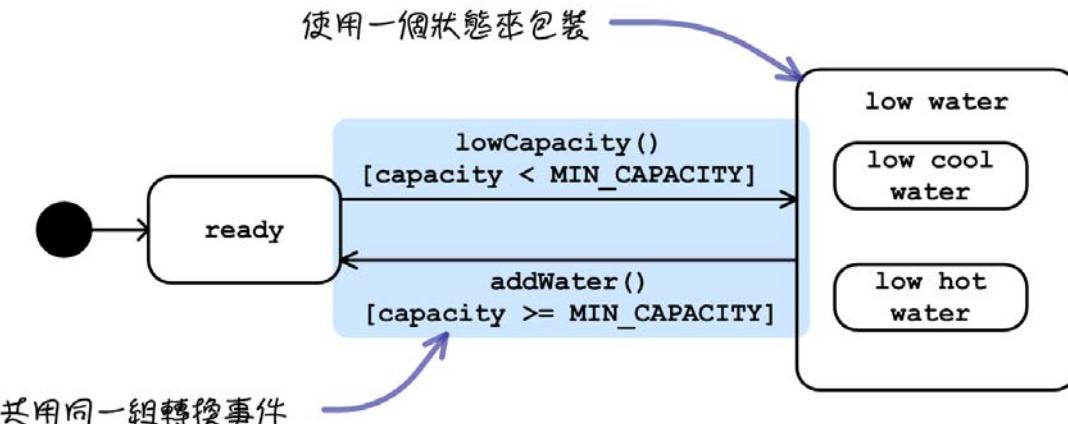


子狀態

在狀態圖型中，可能會出現下列的狀況：



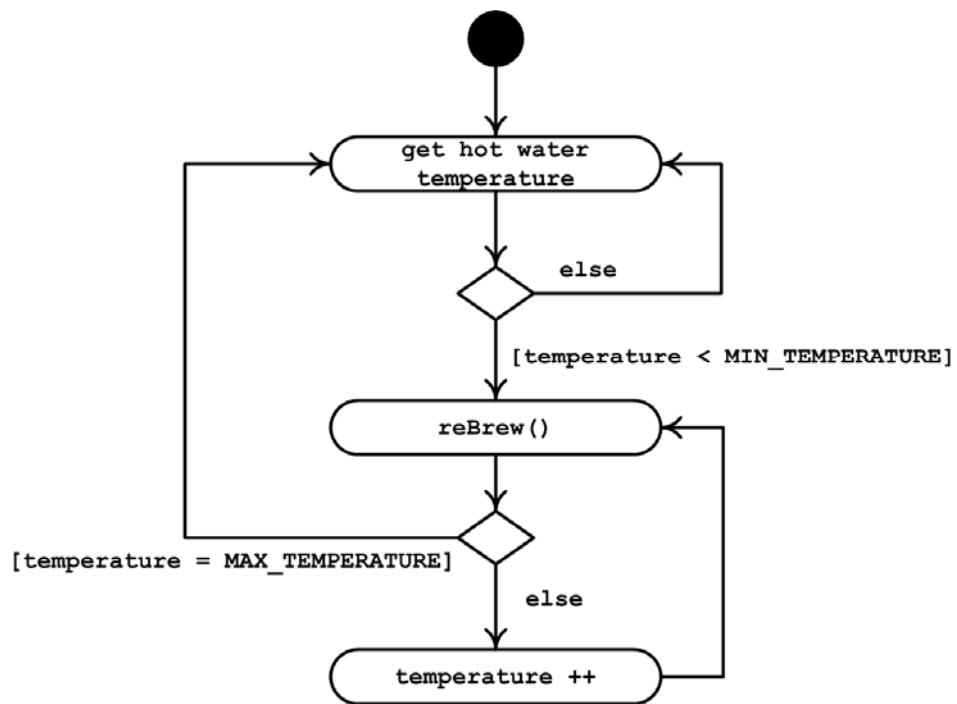
為了簡化上列的圖型，你可以使用一個狀態節點把「low hot water」和「low cool water」包裝起來：



使用這樣的方式來繪製狀態圖型，可以讓圖型更加清楚，「low hot water」和「low cool water」這兩個狀態節點，會稱為「子狀態、Substate」。

10. 活動圖型

「活動圖型、Activity diagrams」用來顯示軟體系統中特定的活動情形，活動圖型與其它圖型有一個最大的差異，就是在顯示活動情形的時候，通常不會考慮與物件或類別相關的問題。所以活動圖型是用來呈現軟體系統中某一些特定的活動，使用一般的流程圖概念來繪製就可以了。下列的活動圖型顯示從偵測溫度到自動加溫的活動流程，在圖型中沒有涉及到任何的物件，只是表現一個活動的過程：



活動圖型主要有下列的用途：

- 針對循序圖型中比較複雜的訊息傳遞加以說明
- 顯示在狀態圖型裡比較複雜的轉換事件
- 說明合作圖型中的訊息

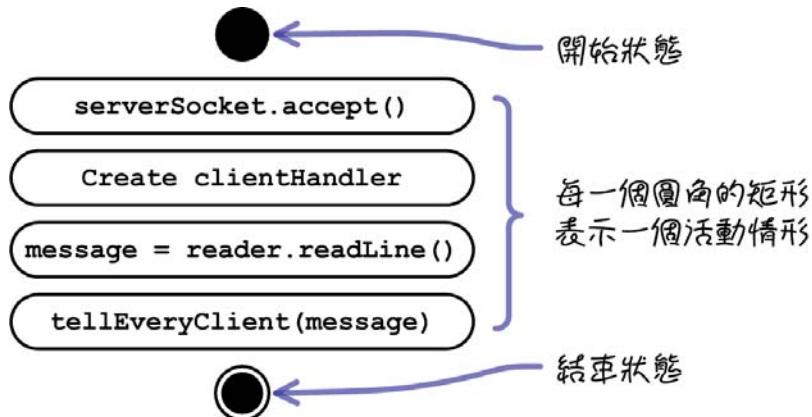
圖型元素

活動圖型由下列的基本元素組成：

- 狀態與活動 (State and Activity)
- 轉換 (Transition)
- 分支 (Branch)
- 分岐與結合 (Fork and Join)
- 水道 (Swimlane)

狀態與活動

活動圖型使用與狀態圖型相同符號的「開始狀態、Start state」和「結束狀態、Stop state」，表示活動圖型的開始和結束；而過程中的每一個活動使用圓角的矩形來表示：



上列圖形的狀態與活動對應的程式敘述像這樣：

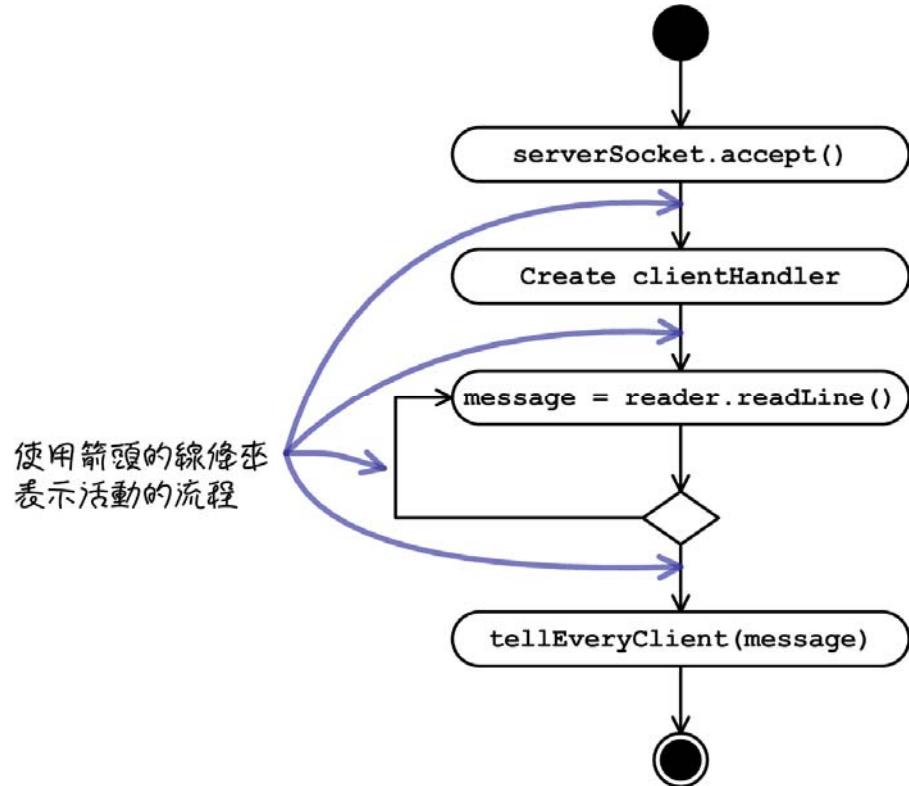
```

Socket clientSocket = serverSocket.accept();
ClientHandler clientHandler = new ClientHandler(clientSocket);
new Thread(clientHandler).start();
while ( (message = reader.readLine()) != null ) {
    tellEveryClient(message);
}

```

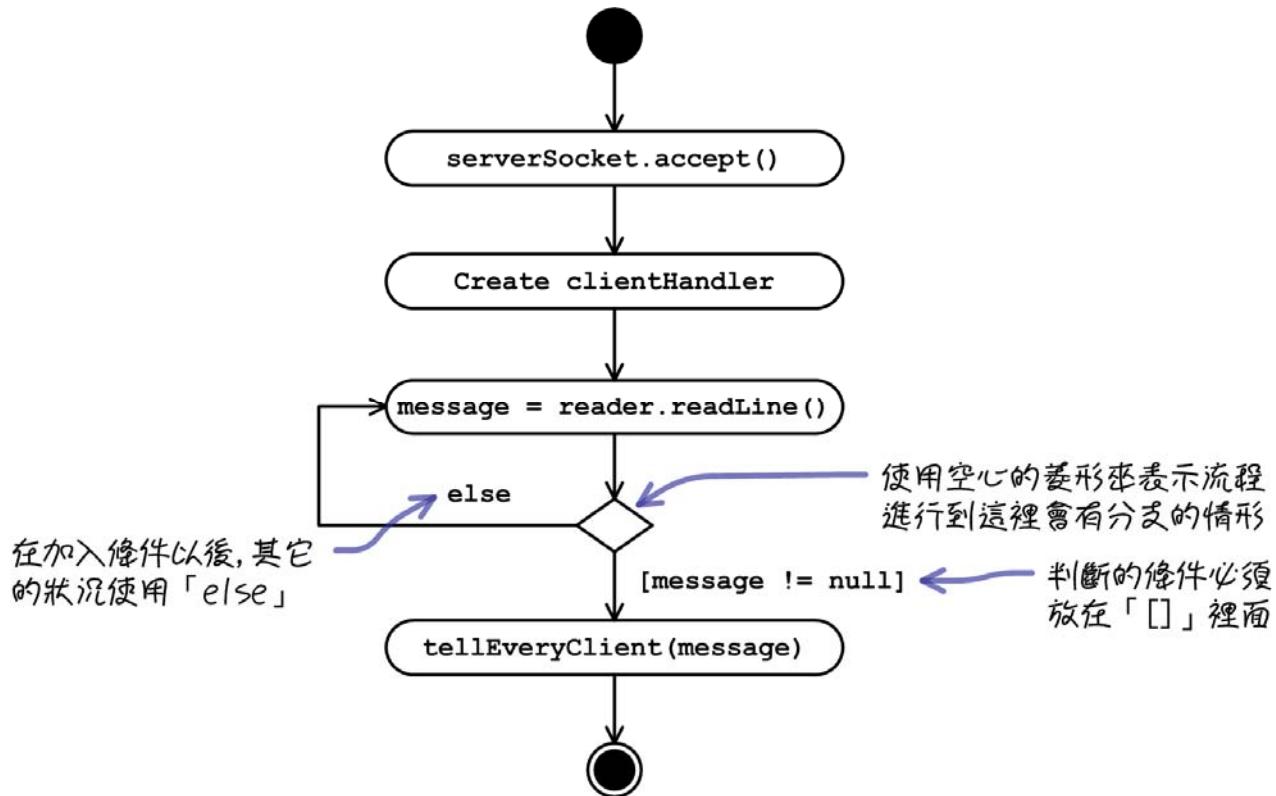
轉換

轉換指的是活動圖型中使用箭頭的線條來表示活動的流程。一般來說，都不會為活動到活動之間的轉換加上說明或條件，只有在分支的時候才會使用：



分支

分支表示依據判斷式而進行不同的活動，分支以後的轉換，就必需加上說明或條件：

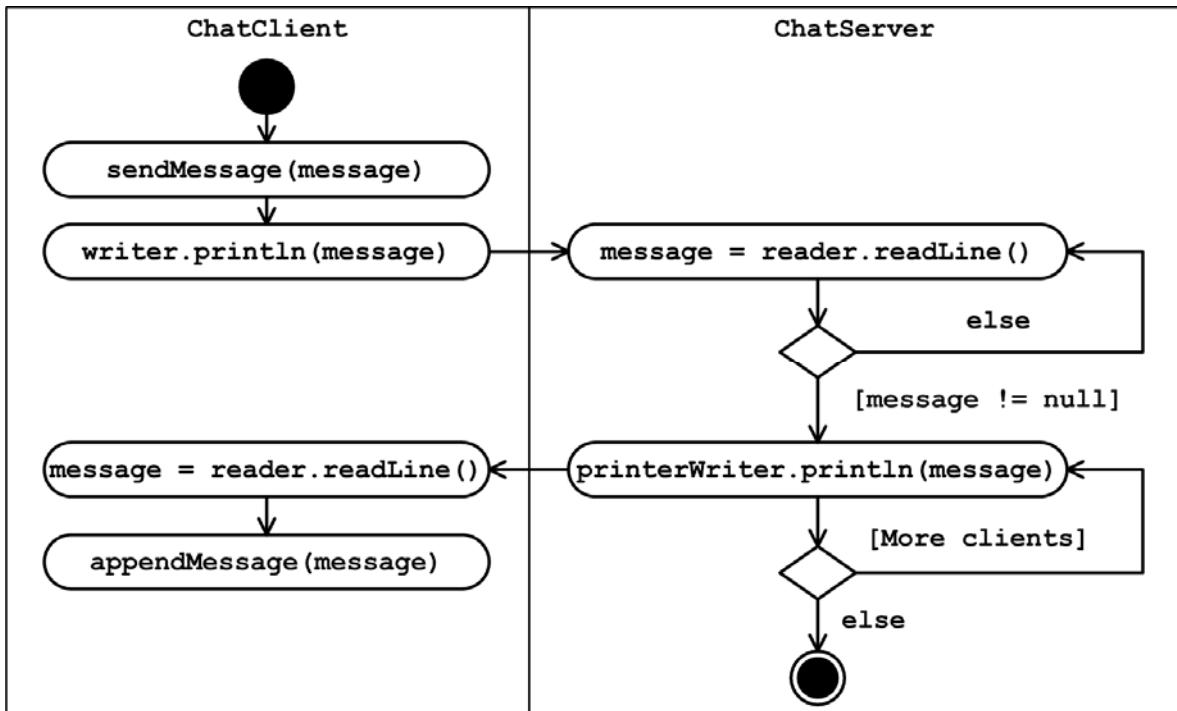


水道

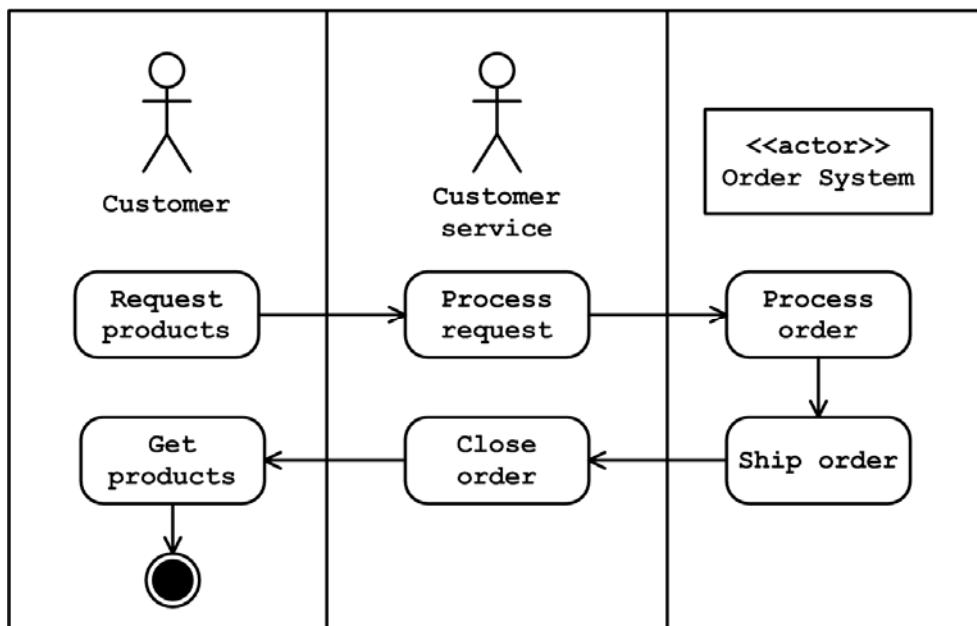
活動圖型中所顯示的活動流程，有可能很單純的在一個方法或類別中就完成了，但是也可能會由不同的角色一起完成流程。以下列的應用程式來說：



從「Mary」發送訊息，到「Chat server」將訊息傳遞給所有用戶的流程，是由兩個角色一起完成這個流程。你可以使用水道來區分不同的角色和所負責的活動：



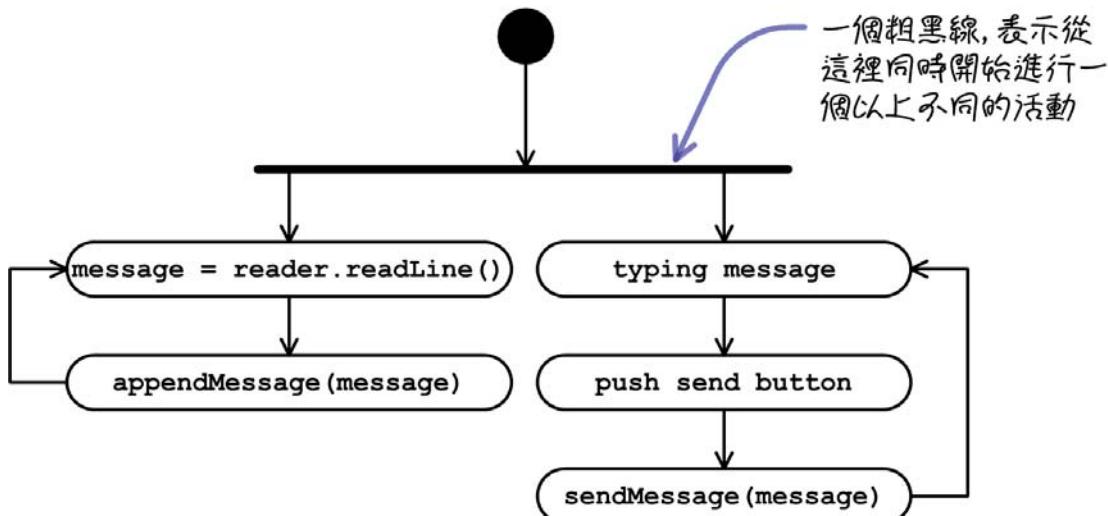
水道也可以使用行爲者標記來表示某一個角色，這樣的表示方法可以讓活動圖型更加清楚：



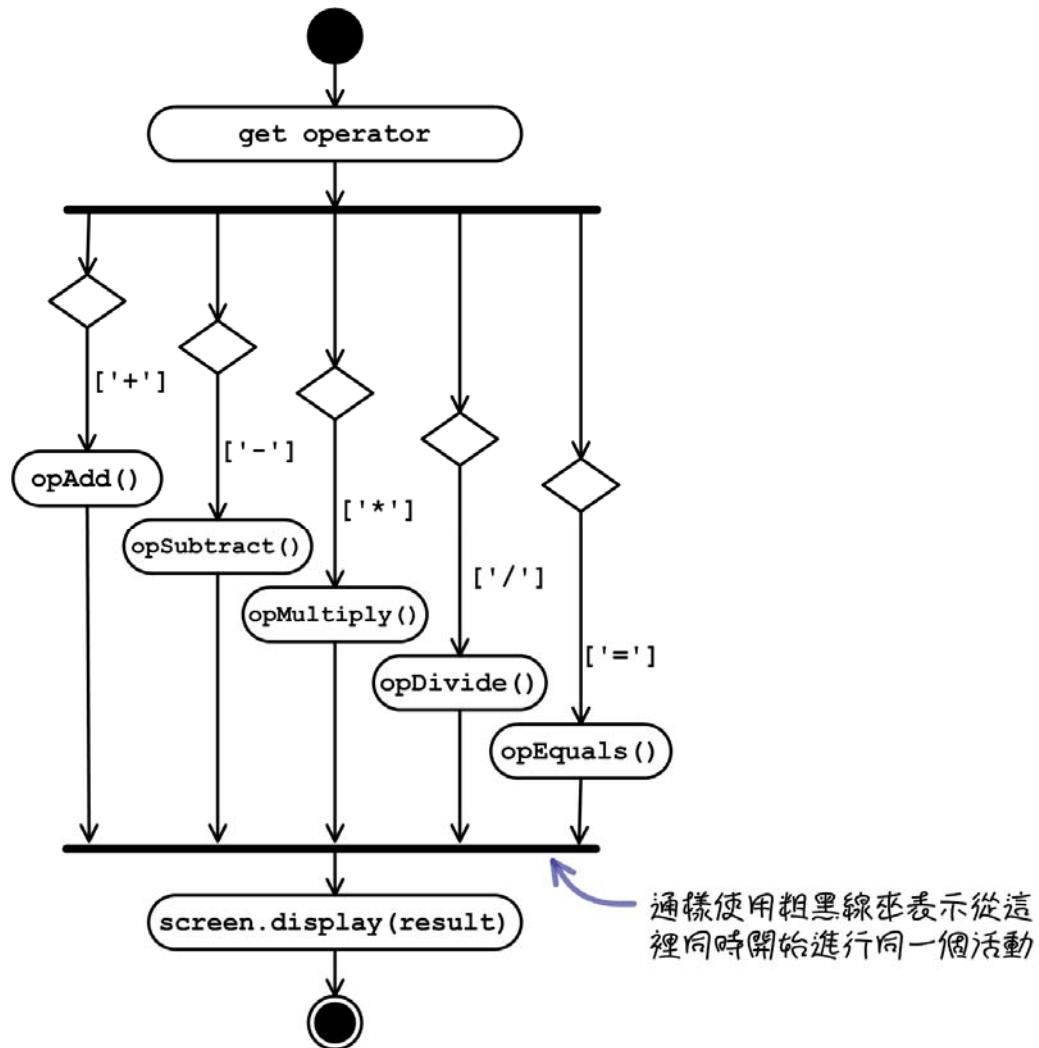
分歧與結合

上列討論的「Chat server」應用程式中，在用戶端的部分，它有兩個活動同時在進行，除了使用者可以隨時輸入並發送訊息外，還一個執行緒在背景執行，隨時接受由「Chat server」傳送過來的訊息。

你可以使用「分歧、Fork」來顯示這類同時進行的活動：



分歧通常用來表示同時進行的活動或是無關前後順序的活動，它也可用來表示「switch」敘述的行為。在下列的圖型中也使用了「結合、Join」標記，把多個活動的流程集合到一個活動：



上列的範例圖型是計算模擬程式中，負責處理功能按鍵的方法，程式敘述會像這樣：

```
private class OperationHanlder implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        char operator = event.getActionCommand().charAt(0);
        String result = "";
        switch (operator) {
            case '+':
                result = calculator.opAdd(screen.getScreen());
                break;
            case '-':
                result = calculator.opSubtract(screen.getScreen());
                break;
            case '*':
                result = calculator.opMultiply(screen.getScreen());
                break;
            case '/':
                result = calculator.opDivide(screen.getScreen());
                break;
            case '=':
                result = calculator.opEquals(screen.getScreen());
                break;
        }
        screen.display(result);
        isReady = true;
    }
}
```