物件導向編程精要

Essential Object-Oriented Programming

柯向上 Josh Ko

2007.03.11

前言

本文介紹物件導向編程（object-oriented programming）的核心脈絡。介紹一門學問有兩種可能途徑：循歷史進程娓娓道來，或依當代理路做系統化介紹，兩條途徑各有優劣，而且些許互斥。本文採納系統化途徑，以 C++或 Java 為例，並預設讀者對這兩個語言或類似語言的 OO 機制有基礎認識，例如 class 與 object 之間的關係。以這兩個語言為例，多少暗示本文所關注的是比較古典的 OOP，如 Ruby 的 duck typing 就不在討論範圍內。

物件導向分解

物理學家致力追尋物理世界的終極定律，他們相信這個定律是簡單而優雅的。與之相對，軟體系統的本質是複雜（complexity）。人類同時能夠處理的資訊量有限，因此當程式員面對一個複雜系統，必須將之分解為規模較小、功能較簡單的元件，一次專注於一個元件而選擇性忽略其餘元件的細節，藉著掌握各個元件而掌握整個系統。這個選擇性忽略的過程稱為 ***abstraction***（**抽象化**），是人類建造複雜系統的關鍵手法。物件導向範型（paradigm）指出，應該將系統分解為一個個的物件（objects），讓這些 objects 之間互動，如此模塑（model）整個系統。相對於 object-oriented decomposition，algorithmic decomposition 的拆解對象是系統的行為舉止，把整個系統的處理過程分解為較小的處理步驟。值得注意的一點是，這兩種範型並無先後或輕重之分，如 Booch 所說[1]：

> 哪一種是分解複雜系統的正確方式 — 以演算法為切入點，還是物件？這其實是個刁鑽的問題，因為正確答案是「兩種觀點都很重要」：演算法觀點突顯事件的次序，而物件導向觀點強調「做出行為或身為某操作的施用對象」的主體。然而，事實是我們不能同時以兩種方式建造一個複雜系統，因為此二者是正交的觀點。我們開始分解系統時，只能從演算法或物件觀點之一下手，然後用析出的結構作為表達其餘觀點的框架。

[1] [Booch94] sec. 1.3, “Algorithmic versus Object-Oriented Decomposition,” p. 17.

Booch 接著指出先套用物件導向觀點對於程式員較為自然。我們不繼續闡論兩種分解方式的區別和適用時機，而把焦點放在物件導向思維本身。

Data Abstraction

介面與實作

由上段可知，我們將複雜系統拆解為一個個物件後，構成整個系統的是這些物件的互動。當抽象層級在系統層次時，我們關注的就是這些物件的行為，此處的物件行為可定義為「接收到某個訊息時所做出的反應」。因此所謂物件互動，就是物件之間互傳聲息而有所反應。傳訊的概念體現於程式語言，就是喚起該物件的 public instance methods[2]。一個物件所能處理的全部訊息，便構成此物件的**介面**（***interface***）[3]，一般是所有 public instance methods 的簽名式（signatures）所構成的集合。欲與物件溝通，就應該透過此物件的介面，因為介面是這個物件所能處理的全部訊息。相對於介面，就是物件本身的**實作**內容（***implementation***）。介面是物件提供外界作為溝通橋樑的部份，而實作是物件內部運行所需的各式資料或機制，外界不應干涉。

至此，一個物件的成份就分為兩類，一類是外界可取用的介面，一類是外界不應干涉的內部實作。介面隱藏物件內部的複雜機制，抽取（abstract）出這個物件的外顯行為，提供較為高階的一組語意。請注意：我們稱介面是物件的外顯**行為**，因此絕大部份組成介面的成員都是函式而非資料。此事稍後在 design by contract 的脈絡下有更強力的證成。

Andrew Koenig 有句名言[4]：

> *Abstraction is selective ignorance.*

在 data abstraction 的脈絡下詮釋該句：外界選擇忽略物件的實作部份，只關心物件的介面，從而只關心物件的外顯行為。因此若遵守「與物件互動時僅存取其介面」的規定，那麼即使是 C 也能夠模擬物件。但 Stroustrup 說得好[5]：

> 此處有個重要區別。一個語言宣稱**支援**（*supports*）某種編程風格，意思是它提供設施使得運用該種風格很方便（合理地簡單、安全、有效率）。一個語言不支援某種技巧，意思是撰寫那種程式需要額外心力或不尋常的技術；它只是**允許**（*enables*）使用該種技巧。

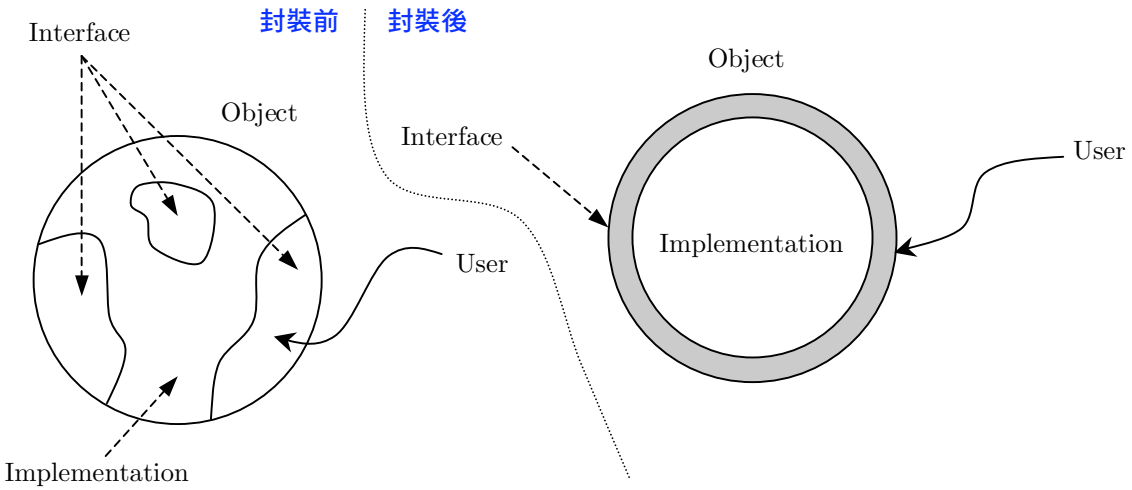
[2] 在 C++ 就是 public non-static member functions。
[3] 此處的 interface 採其廣泛意義，非指 Java 的 interface 構件。後者只是前者於 Java 語言的體現，這在本文稍後有所論述。
[4] [Koenig97].
[5] [Stroustrup91] sec. 2 “Programming Paradigms,” par. 2.

例如，你可以用 *Fortran* 寫結構化程式，用 *C* 寫型別安全的程式，在 *Modula-2* 裡使用 *data abstraction*，但這麼做會有非必要的困難，因為這些語言不支援那些技巧。

封裝

物件導向語言對 data abstraction 的回應（也就是此類語言宣稱**支援**物件所必須提供的設施）之一就是**封裝**（***encapsulation***）。既然外界僅應與物件的介面溝通，不應干涉實作，那麼就用介面把實作包藏在內，令外界僅能接觸介面而無法看見實作，強迫物件的使用者專注於介面，從而強制實現 data abstraction。一般支援封裝的方式是提供對成員的存取控制（access control），public members 就構成物件介面，private members 則是物件實作。

另一方面，封裝後使用者只能存取介面，如此只要介面不變，無論實作如何改變，使用者都不會受到波及。「使用者撰寫的程式碼」和「物件的實作程式碼」之間的耦合關係因為封裝而強制被解除，術語稱這個過程為**除耦**（***decoupling***）。介面就如同一道防火牆，使一側的變動不至於擴散到另一側。



Design by Contract

接著我們引入 Bertrand Meyer 的方法論 ***design by contract***（DBC），進一步論述介面與實作的角色，以及封裝的必要性。DBC 是個效力甚強的規範，無論在實務上檢驗程式正確性，或如本文內用以刻劃原則，都相當有效。Meyer 以「契約」形容物件介面的詳細規格，簽訂這份契約的雙方是分別在物件介面兩側進行編程的「物件設計者」和「物件使用者」。這份契約最顯眼的部份是介

面上各個函式的簽名式（signature）和語意。請注意：語意往往無法以程式語言描述，必須另外以文件闡明。Arnold、Gosling、和 Holmes 特別指出這一點[6]：

> 一個常見的假設是，*class* 宣告的函式就是 *class* 的完整契約。函式語意也是契約的一部份，即使它們可能只描述於文件內。兩個函式可能擁有相同的名稱和參數，並擲出相同的異常，但如果它們的語意不同，兩者就不等同。例如並不是每個名為 *print* 的函式都可被假設為印出一份 *object* 複本。也許有人將 `print()` 的語意定義為 *"process interval"* 或 *"prioritize nonterminals"*，雖然我們並不建議這麼做。函式的契約 — 簽名式（*signature*）和語意 — 定義出函式的意義。

Java documentation comments 的設計良好地反映這個觀點，提供一套標準機制，讓「描述函式語意的文字」能和函式一起出現在程式碼之中，並能以工具萃取出來，產出描述 class 介面的網頁。

除了函式明確指出的介面以外，較隱晦的一個條款是 ***class invariant***，指明此類物件的狀態（state）必須滿足的條件。最常見的情況是，物件狀態由 instance variables 組成，此時任意賦值給各個變數不一定是有意義的狀態。以下面的 C++ class `Stack` 為例：

```
template <typename T>
class Stack {

public:  // interface

  // copy-control members omitted

  void push(const T&);
  void pop();
  T& top() const;

  size_t size() const;
  bool empty() const;

private:  // implementation

  T* buf;
  size_t buf_size;
  size_t cur_size;

  void grow();
};
```

`Stack` objects 必須遵守的 class invariants 有：

1. `cur_size` 恆小於等於 `buf_size`。

[6] [Arnold05], chap. 2 "Classes and Objects," par. 3, p. 41.

2. `buf[0]`到`buf[cur_size - 1]`是 stack 內的所有元素，且依適當順序擺放。

只有在上述兩條 class invariants 成立的環境下，`Stack` objects 才處於有效狀態（valid state），其成員函式（member functions）方能正常運作。這裡引出「契約」裡面關於成員函式的細則：對每個成員函式的呼叫都帶有**前置條件**（***precondition***）和**後置條件**（***postcondition***）。呼叫某個成員函式之前，呼叫者必須確保該成員函式所需的前置條件都已滿足。另一方面，每個成員函式回返前都必須確保其後置條件成立。Class invariants 即自動蘊含於每個成員函式的前置條件和後置條件之內。物件在一個個成員函式呼叫之前與之後維護其 class invariants，從而持續處於有效狀態而得以持續有效運作。請注意：成員函式執行期間，class invariants 可以暫時被摧毀。例如上例的 private 成員函式 `grow` 負責增長儲存空間，過程中會摧毀 `Stack` 的 class invariants，但 `grow` 回返之前，必須重新建立 class invariants。

由 class invariants 的觀點，可輕易證成封裝的必要性。若允許外界任意修改物件內容，那麼 class invariants 無法保證恆成立。相反地，成員函式是物件的一部份，物件設計者能夠謹慎撰寫成員函式維護 class invariants，最後經由封裝機制，便能確保 class invariants 恆成立。（因為唯一可能破壞 class invariants 的外部使用者僅能存取沒有破壞效果的成員函式。）第二序原則「絕大多數情況不應將 member variables 暴露於介面」的證成可直接由上段論證獲得。一個例外的經典例子是 C++的 `std::pair`，此類物件單純把數個變數打包在一起，因此 `std::pair` 的 member variables `first` 和 `second` 無需特別的 invariants 束縛，而不受前述原則約束。

一個成員函式呼叫所需的 class invariants 係由前次成員函式呼叫保證成立。那第一次成員函式呼叫所需的 class invariants 呢？從無到有建構 class invariants 的任務就落在建構式（constructor）身上。建構式接收「生成一個物件」所需的所有參數，設立物件存活所需的環境，這在 design by contract 的脈絡下就是 class invariants。物件生成必定伴隨建構式呼叫，從而確保物件「出社會」時，所有的 class invariants 已然齊備。

論 C++ Friend 函式

一個常見的說法是「C++ friend functions 破壞封裝性」，但其實 friend functions 對封裝性的破壞力和 member functions 相當[7]。這麼說的原因很簡單 — 與 class 處於同一個 namespace 的 friend functions 也是 class 介面的一部份，只是呼叫形式不同。Sutter 以 C++的 Argument-Dependent Lookup（ADL，又稱 Koenig Lookup）規則支持此一事實[8]。當喚起一個非成員函式，ADL 要求 overload resolution 的第一步 name lookup 必須把「引數型別所在的 namespace(s)」納入搜尋範圍，

[7] 依[Meyers00]的說法。
[8] [Sutter00], “Name Lookup, Namespaces, and the Interface Principle.”

因此喚起 friend functions 和喚起 member functions 除了語法以外沒什麼差異。至於「所在 namespace 與參數型別棲身之 namespace 相異」的 friend functions 違反的是模組化原則，算是濫用（abuse），不在討論範圍內。

Abstract vs. Concrete

嚴格地講，僅用上 data abstraction 的程式不能稱為物件導向程式。物件導向編程奠基於 data abstraction 繼續延伸，但 data abstraction 並不必然發展為 object-orientation。一個重要的反例是 C++ STL，它的基石之一就是 C++的 data abstraction 機制，但其設計思維 generic programming 的旨趣、實作和 C++ OOP 無顯著交集。Data abstraction 的產物一般稱為 abstract data types（ADT），Doug McIlroy 曾如此評論：

> *Those types are not "abstract," they are as real as* `int` *and* `float`*.*

此句中的 "real" 也可代換為 "concrete"。表面上，abstract 與 concrete 兩相矛盾，但此時卻用來形容同一個概念。解釋這層（表面上的）矛盾有助於釐清 data abstraction 的真意。

"Concrete types" 是 Stroustrup 偏好用來稱呼 ADT 的辭彙[9]：

> ……相對簡單的「具體」使用者自定型別，邏輯上和內建型別沒什麼不同。理想上，此類型別和內建型別在使用方式上不應有差異，只在產生方式上有所不同。……我稱呼單純的使用者自定型別為 *concrete types*，以和 *abstract classes* 與 *class hierarchies* 有所區別，同時強調它們與內建型別如 `int`、`char` 的相似之處。……它們的使用模式和其設計背後的「哲學」相當不同於常被宣揚的 *object-oriented programming*。

C++發展出一套完整機制支援 data abstraction，其中值得一提的是 operator overloading，這使 concrete types 的介面能夠和內建型別極為相近，戲劇性地縮短兩者之間的差距。封裝讓我們只看到 concrete type 的介面而不見實作，正如我們操作 `int` 時任意使用整數算術而不理會機器如何處理加法一般。我們甚至可以從字面意義詮釋 "concrete" types：concrete objects 硬實不變，就如同 `int` 的表述（representation）與操作烙在機器上堅不可變一樣。介面和實作是一體的，只是我們選擇從介面外殼存取它們而已。Data abstraction 的硬實特質稍後將被 OOP 軟化鬆動，創造出執行期多型（runtime polymorphism）。

[9] [Stroustrup00], sec. 10.1, p. 224, par. 2; sec. 10.3.4 "The Significance of Concrete Class," p. 241.

至於 ADT 中的 abstract，自然是源於 data abstraction，描述的是 ADT 暴露在外的抽象介面。這裡的抽象和先前的具體並不相斥，因為此處的抽象並不是說物件虛無飄渺難以捉摸，而是 Koenig 所稱的選擇性忽略 — 把使用者真正關心的特質抽取出來，忽略其他非關注焦點的部份。

繼承與多型

差異編程

繼承（inheritance）的第一種使用模式是差異編程（program by difference），精神上是 data abstraction 的直接擴充。給予一個 ADT，若要強化或增加其功能，可針對需要修改的部份撰寫程式，不需修改的部份則直接繼承，如此獲得一個新的 ADT。我們稱原本的 ADT 為 superclass，新生的 ADT 為 subclass，在 C++的術語是 base class 和 derived class。由此類繼承的目的立刻能推得 name hiding 的高階意義 — 如果在 derived class 裡面出現一個同名函式，其目的是替換掉 base class 的對應函式，因此名稱應該決議至 derived class 所定義的版本。一個式子就能解釋這種繼承的絕大部份行為：

A subclass object = a superclass (sub)object + subclass extension.

一個直接推論就是，subclass 建構式必須（隱寓或明白地）喚起 superclass 建構式。請注意，subclass object 之於「其內含的 superclass object」仍是使用者，所以 derived object 使用 base object 的方式和一般使用者完全一致，base class 的實作部份（存取層級為 private 的部份）也不為 derived object 所見。然而，繼承的目的是精煉或修改 superclass 的行為，這可能需要存取 superclass 實作部份。可能的解決方案包括解除封裝或把部份實作放到介面，但這將增加與一般使用者的耦合性。C++/Java 的解法是引進另一個存取層級 protected，實質意義是造出另一個針對 derived class 的介面，透過這個介面能存取 base class 不應公開給一般使用者但應提供給 derived class 使用的部份。請注意，這暗示 derived class 和 base class 之間有強烈的耦合性（base class 的變動容易波及 derived class），因此繼承常被形容為兩物體（entities）間最強的關係[10]。

在 C++，若以傳值（by value）方式將 derived object 賦值給 base class variable，發生的是 slicing — 被複製過去的只有 derived object 裡面的 base object。（差異編程模式的繼承之下）若讓一個 base pointer/reference 指向一個 derived object，效果就相當於指向該物件內含的 base (sub)object — 透過這個 pointer/reference 喚起的成員函式一定是 base class 定義的版本。換句話說，一個變

[10] 在 C++有個例外 — friend functions。從前面的討論可知，friend functions 其實是 object（介面）的一部份。

數的 declared type 和此變數所指涉 object 的 actual type 永遠符合。因此在差異編程下的繼承，介面（declared type）和實作（actual type）之間仍未鬆動，並未跳脫 data abstraction 的思維。

介面編程

往 OOP 最重要的一步跳躍，在於鬆綁 ADT 介面與實作間的緊密連結關係，引出動態多型 — 介面後的實作能夠在執行期變動。GoF 提出一個核心的 OO 原則[11]：

> *Program to an interface, not an implementation.*

乍看之下這和 data abstraction 沒太大差異 — 封裝不就強迫我們針對介面編程了嗎？關鍵在於，ADT 的介面與實作是一體而且固定的，使用者只是選擇把關注焦點放在介面，而多型編程的對象完全是介面本身，介面後的實作可動態抽換，甚至可能是使用端未知的物件。但並非任意物件都能被換上、接在介面之後，這個實作品必須遵守介面所定的契約。而聲稱「某個實作品遵守介面之契約」的方式，就是繼承。

在介面編程模式的繼承下，subclass 仍然是個擁有自己的介面與實作的 ADT，但它同時也繼承 superclass 的介面，並適當替換 superclass 某些函式的實作。請注意，替換後的實作仍須遵守 superclass 介面的契約，因為介面編程的精髓在於「透過 superclass 的介面使用 subclass objects」的能力。這要求 subclass 和 superclass 之間具有所謂的 ***IsA*** 關係，例如 class `Rectangle` 繼承 class `Shape`，語意即矩形「是一種」形狀。更精確的定義由 ***Liskov Substitution Principle***（LSP）給出[12]：

> 任何可使用 *superclass objects* 的地方，都必須能以 *subclass objects* 代換。

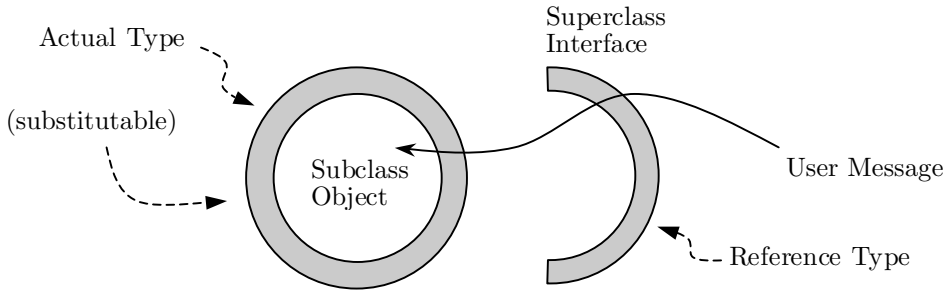
這個原則所指示的可代換（substitutable）關係是介面繼承的核心。Liskov 的原文是：

> 此處所要的是類似以下的代換性質：若對於每個型別 S 的物件 o_1，都有個型別 T 的物件 o_2，對於所有以 T 定義的程式 P，以 o_1 代換 o_2 時 P 的行為不變，則 S 就是 T 的 *subtype*。

我們改由傳訊的觀點詮釋介面編程：在 data abstraction 下，當我們傳遞訊息給一個 class 的介面，處理訊息的就是那個 class 本身的實作。程式執行期間，每個訊息會被哪份實作處理都已確定。但在介面繼承下，當我們傳訊給 superclass 介面，我們不能確定處理訊息的是 superclass object 或哪一個 subclass object — 訊息會穿透介面抵達彼端的未知物件，由那個物件依照自己設定的方式處

[11] [Gamma95], sec. 1.6, "Programming to an Interface, not an Implementation," p. 17.
[12] [Martin02], chap. 10, "LSP: The Liskov Substitution Principle."

理那個訊息。只要彼端物件不違反 LSP，這個訊息就能被妥善處理，因為介面契約讓雙邊能夠在不認識對方的情況下無隙合作。



這樣的函式呼叫機制顯然和差異編程模式下的繼承相當不同，靜態繫結（static binding）無法達成這樣的效果。針對 superclass 介面撰寫程式，若採用靜態繫結，處理訊息的永遠是 superclass object。此處需要的機制不能在編譯期就決定函式名稱的繫結對象，必須等到執行期，真的有個物件在介面之後，才喚起那個物件的對應函式。這個關鍵機制就是**動態繫結**（***dynamic binding***），也稱晚期繫結（late binding）或虛擬繫結（virtual binding）。在 Java，所有 instance methods 預設採用動態繫結，而在 C++需明確以 `virtual` 修飾之。動態繫結係根據物件的 actual type 決定繫結對象，而非物件的 reference type — 前者是實際用以生成該物件的 class，而後者是用以指涉該物件的 class，也就是用以存取該物件的介面。當一個函式採用動態繫結，就能保證無論透過哪個介面存取物件，都能喚起正確的函式。從 superclass 介面觀之，原本應該喚起 superclass 函式，現在因為動態繫結改而喚起 subclass 函式，猶如後者蓋過前者一般，因此我們稱 subclass 函式覆寫了（overrides）superclass 函式。

動態繫結與介面繼承攜手登場，就帶來 OOP 核心的核心 — **多型**（***polymorphism***）[13]。多型的意義在於，我們針對介面撰寫程式，介面之後的物件形態可能活跳亂蹦，但一定遵守介面契約，如此我們寫的程式毋需修改（甚至毋需重新編譯）就能與未知的程式協同運作。這符合 Bertrand Meyer 深具洞見的 ***Open-Closed Principle***（OCP）[14]：

> *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

針對 superclass 介面撰寫的程式不再更動，謂之封閉；藉著提供適當的 subclass 實作品而得以產生不同的擴充行為，謂之開放。重新省視 GoF 所提出的原則，若針對實作品撰碼，正違反了 OCP — 使用者的程式碼與實作耦合，擴充程式功能時就可能得修改既有程式碼。特別值得一提的是，在

[13] 本文中指涉「多型」均採狹義觀點，侷限於 OOP 的動態多型，而不考慮其餘種類的多型。
[14] [Meyer97]; [Martin02], chap. 9, "OCP: The Open-Closed Principle."

執行期，ADT 就是一種實作品，因為傳遞訊息給 ADT，這個訊息一定由 ADT 本身的實作加以處理，所以意義上是 "program to an implementation"[15]。OCP 也可證成 LSP：如果有某個 subclass object 無法代換掉 superclass object，定然是這個 subclass object 有一些不符合契約的特性，那麼當初針對 superclass 撰寫的程式不一定能原封不動而適應這個 subclass object 的特性。

運用多型最經典的例子就是 application framework，應用程式的框架都已準備妥當不需修改，程式員只需撰寫適當的 subclasses 提供自定行為。執行時，框架程式碼負責驅動所有基礎設施，並在適當時機透過多型機制喚起那些自定行為。這就使框架得以**復用**（***reuse***），而讓應用程式撰寫者得以專注於應用程式的特化行為，毋需每次從輪子重新打造（reinvent the wheel）。

從 DBC 看 LSP

我們再次請出 Design by Contract 方法論，更精確地刻劃 LSP 所描述的可代換關係。DBC 指出每個成員函式的呼叫前後分別有前置條件和後置條件必須滿足，這就限制了 subclass 成員函式的實作方式，因為根據 LSP，subclass 提供的函式實作必須遵守 superclass 介面所訂的契約。一般化的說法是：

> *Subclass 函式所需的前置條件不得強於 superclass 函式，而 subclass 函式所建立的後置條件不得弱於 superclass 函式*[16]。

這個定理除了指引 subclass 函式的寫作方式以外，也可用來解釋某些語言構件的設計。例如，Java 的 overriding method 之存取層級只能比 overridden method 來得寬鬆，這來自前置條件的限制 — 如果 overridden method 已經是 public，就代表此函式在契約上聲稱眾生皆可存取，那麼 overriding method 就不應該是 protected（或其他層級），因為這樣的函式僅限 subclasses 存取，使前置條件變強。又例如，Java/C++都支援 covariant return types，這個特性允許 overriding method 的回返型別不須與 overridden method 的回返型別完全相同，而可以是後者的 subtype。使用端只預期回返的是 supertype，但實際上給的卻是較特化的 subtype，後置條件變強，不違反 LSP。

Abstract Classes & Java Interfaces

Abstract class 把介面編程的概念再往前推一步，允許程式員省略部份函式的實作，只留下介面。因為實作不全，abstract class 無法具現化（instantiated），subclass 必須填補所有遺漏的實作細節

---

[15] 請注意，這是假設「執行期需要變動」（介面編程的威力所在）所得到的結論。如果執行期不需變動行為且允許重新編譯，ADT 本身的介面即足以隔離使用端和實作部份。

[16] 以反證法即可輕易推得這個結論：若 subclass 函式的前置條件強於契約上的前置條件，使用端未必會提供足夠強的前置條件，那麼 subclass 函式無法保證正確運作；若 subclass 函式的後置條件弱於契約上的後置條件，那麼使用端假設成立的事實可能不為真。

方能具現出物件。繼續純化，就抵達 Java interface 構件 — 所有實作皆不列，純粹指定介面，作為編程的對象。因為介面的概念在 Java 被提煉成 interface 構件，Java class 就不像 C++ class 那樣兼任「作為編程對象的介面」的角色[17]，而純粹是實作品，因為帶有實作的程式碼都放在 Java class。經先前一番討論，Java interface 的設計應該相當直覺（例如所有 members 自動成為 public、變數自動成為 static non-blank final）。Arnold、Gosling、和 Holmes 如此評論：

> 任何你預期將被擴充的主要 *class*，無論為 *abstract* 與否，都應該實作某個 *interface*。

此句的證成相當簡單：句中所描述的 class 顯然意圖用於介面編程，而 Java 又已把「介面」焠鍊成 interface 構件，因此在 Java 最恰當的做法是明確以 class 代表實作、interface 構件代表介面。

若要使一個物件契合多個介面，C++提供多繼承（multiple inheritance），但因 C++ class 未將介面與實作的概念分開來，多繼承可能使繼承得來的多份 base class 實作發生衝突。Java class extension 採單繼承（single inheritance），但允許繼承（實作）多個 interfaces，避開多份實作的衝突問題。程式員可能用多個介面指涉同一個物件，例如可以透過該物件自己的 class、它的 superclasses、或它所實作的 interfaces。這或可稱為物件觀點的多型：一個物件有多個介面，因為動態繫結而總是能親身處理自各個介面傳來的訊息。選擇從其中一個介面觀察此物件時，就如同戴了一副有色眼鏡，只看得到物件契合這個介面的部份，而透過這個介面傳送過去的訊息總是能被物件妥當處理，無論這個介面和物件的 actual type 距離有多遠。

總結

以下是我們從最簡單的物件概念走到純粹介面編程的諸里程碑：

1. 物件導向的精神在於將複雜系統分解為彼此互動的物件。
2. 區分物件介面與實作，前者是外界與物件的溝通管道，後者是物件內部的運行機制。
3. 封裝把實作隱藏在介面之後，強制實現 data abstraction 並降低耦合性。
4. 差異編程的繼承機制讓程式員能方便地從既有的 ADT 產生新的 ADT。
5. 介面編程把介面的概念獨立出來，運用動態繫結和繼承造出執行期多型，讓介面後的實作品能夠極具彈性地任意抽換。至此我們才踏進了 OOP 的大門。

[17] Java interface 在 C++的對偶角色是「成員全為 pure virtual functions」的 abstract class。

致謝

感謝白夜（Thirddawn）同學予我動機擬出本文骨幹 ☺。

參考資料

[Arnold05] Arnold, K., Gosling, J., Holmes, D. *The Java Programming Language*. Fourth edition, Addison-Wesley, 2005.

[Booch94] Booch, G. *Object-Oriented Analysis and Design with Applications*. Second edition, Addison-Wesley, 1994.

[Gamma95] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns*. Addison-Wesley, 1995.

[Koenig97] Koenig, A., Moo, B. E. *Ruminations on C++*. Addison-Wesley, 1997.

[Martin02] Martin, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[Meyer97] Meyer, B. *Object-Oriented Software Construction*. Second edition, Prentice Hall, 1997.

[Meyers00] Meyers, S. *How Non-Member Functions Improve Encapsulation*. *C/C++ Users Journal*, February 2000.

[Stroustrup91] Stroustrup, B. *What is Object-Oriented Programming? (1991 revised version)*. Proc. 1st European Software Festival. February, 1991.

[Stroustrup00] Stroustrup, B. *The C++ Programming Language*. Special third edition, Addison-Wesley, 2000.

[Sutter00] Sutter, H. *Exceptional C++*. Addison-Wesley, 2000.