

# Object Oriented Programming

## Lec01: OOP Conception

Sun Chin-Yu (孫勤昱)

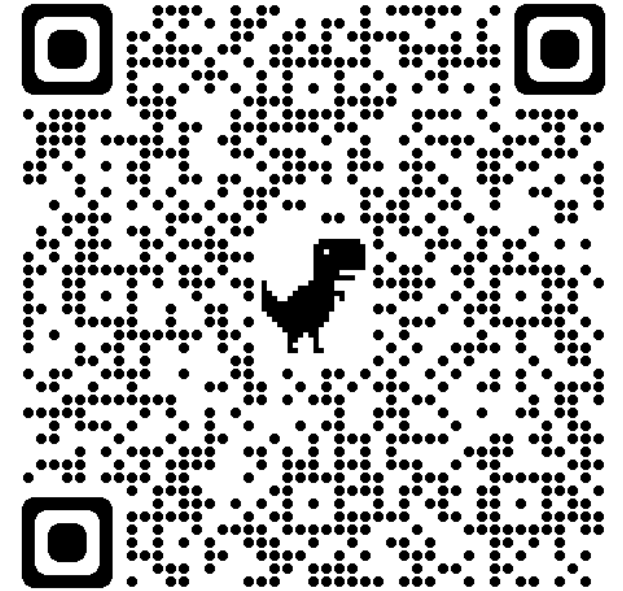
[cysun@ntut.edu.tw](mailto:cysun@ntut.edu.tw)

2023/09/11



# 課前準備

- <http://gitlab.is1ab.com/OOP2023f/announcement>
- <http://gitlab.is1ab.com/>
- <http://jenkins.is1ab.com/>
- Survey of the examination participation:
  - Please fill it to report the participation
  - If you do, you will get +1 point on the final grade :D
  - If you do not, you will lose -3 point on the final grade >:(



# Outline

- Schedule Review
- 為什麼我們需要 OOP
- 什麼是 OOP
- OOP 的幾個主題
  - 封裝
  - 繼承
  - 多型
  - 組合
  - 依賴注入
  - 介面



# Schedule Review

W	Date	Lecture	Homework
1	09/11, 09/15	Lec01: Course Information, Environment Introduction, and OOP Conception	Homework 00 (Fri.)
2	09/18, 09/22	Lec02: Class Introduction, and Essential STL Introduction	
3	09/25, 09/29	Lec02: Class Introduction, and Essential STL Introduction / No class due to Moon Festival	Homework 01 (Mon.)
4	10/02, 10/06	Lec03: Encapsulation	
5	10/09, 10/13	No class due to the bridge holiday of Nation day / Lec03: Encapsulation	Homework 02 (Fri.)
6	10/16, 10/20	Lec04: Inheritance	
7	10/23, 10/27	Lec04: Inheritance	Homework 03 (Mon.)
8	10/30, 11/03	Lec05: Polymorphism	Homework 04 (Mon.)
9	11/06, 11/10	Physical Hand-Written Midterm / Physical Computer-based Midterm	
10	11/13, 11/17	Lec05: Polymorphism	
11	11/20, 11/24	Lec06: Composition & Interface	Homework 05 (Fri.)
12	11/27, 12/01	Lec07: Factory	Homework 06 (Fri.)
13	12/04, 12/08	Lec08: Dependency Injection	
14	12/11, 12/15	Lec09: Efficiency	
15	12/18, 12/22	Lec10: RAI	Homework 07 (Mon.)
16	12/25, 12/29	Physical Hand-Written Final, Flexible time	
17	01/01, 01/05	No class / Physical Computer-based Final	
18	01/08, 01/12	No class & Let's you guys to get prepared for other finals	

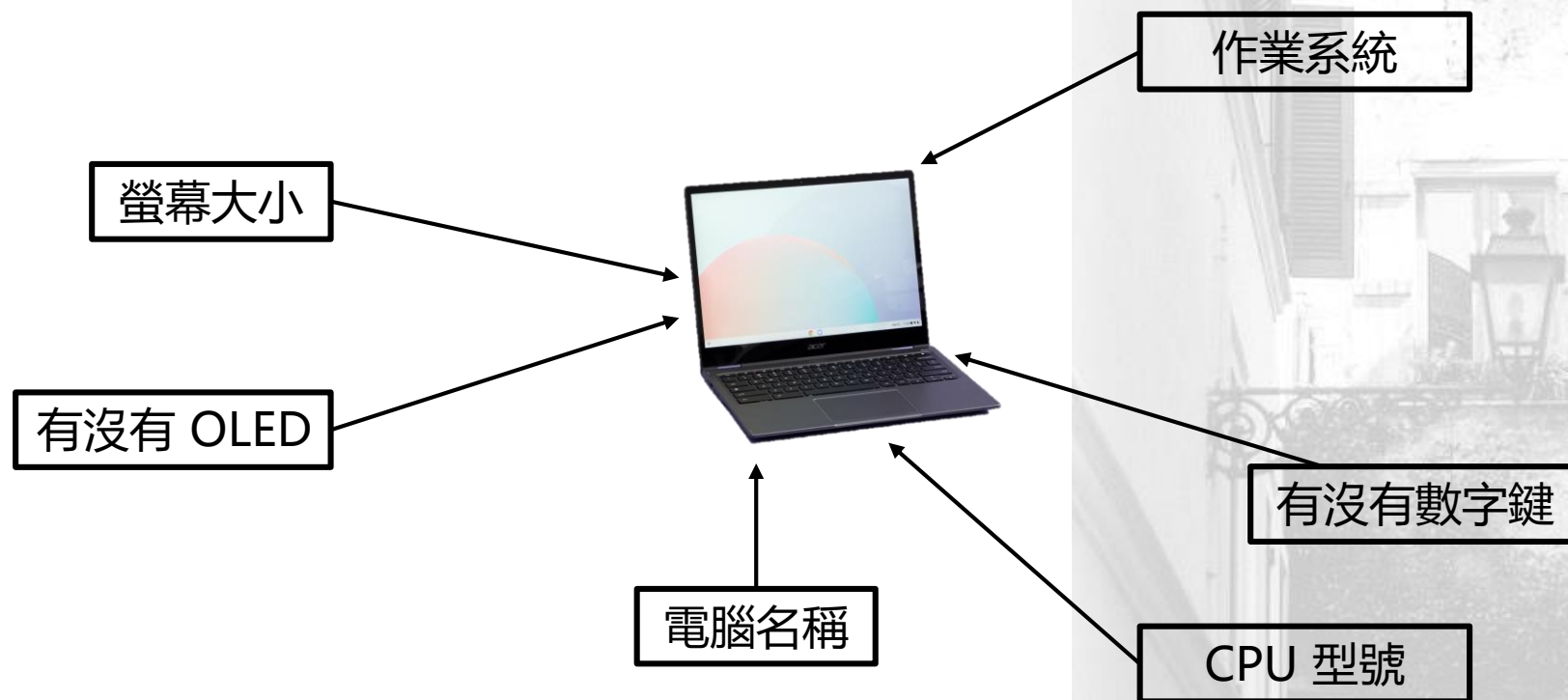


# 為什麼我們需要 OOP ?



# 為什麼我們需要 OOP (1/8)

- 試想，你想要用程式描述這台筆電 ...



# 為什麼我們需要 OOP (2/8)

- 如果使用程式碼描述這台筆電，會長得像這樣 ...

作業系統 = Windows

螢幕大小 (吋) = 14

有沒有 OLED = 有

有沒有數字鍵 = 有

電腦名稱 = 這是一台酷電腦

CPU 型號 = Letni i9-48763 4.87GHz

# 在這裡實作輸出的細節

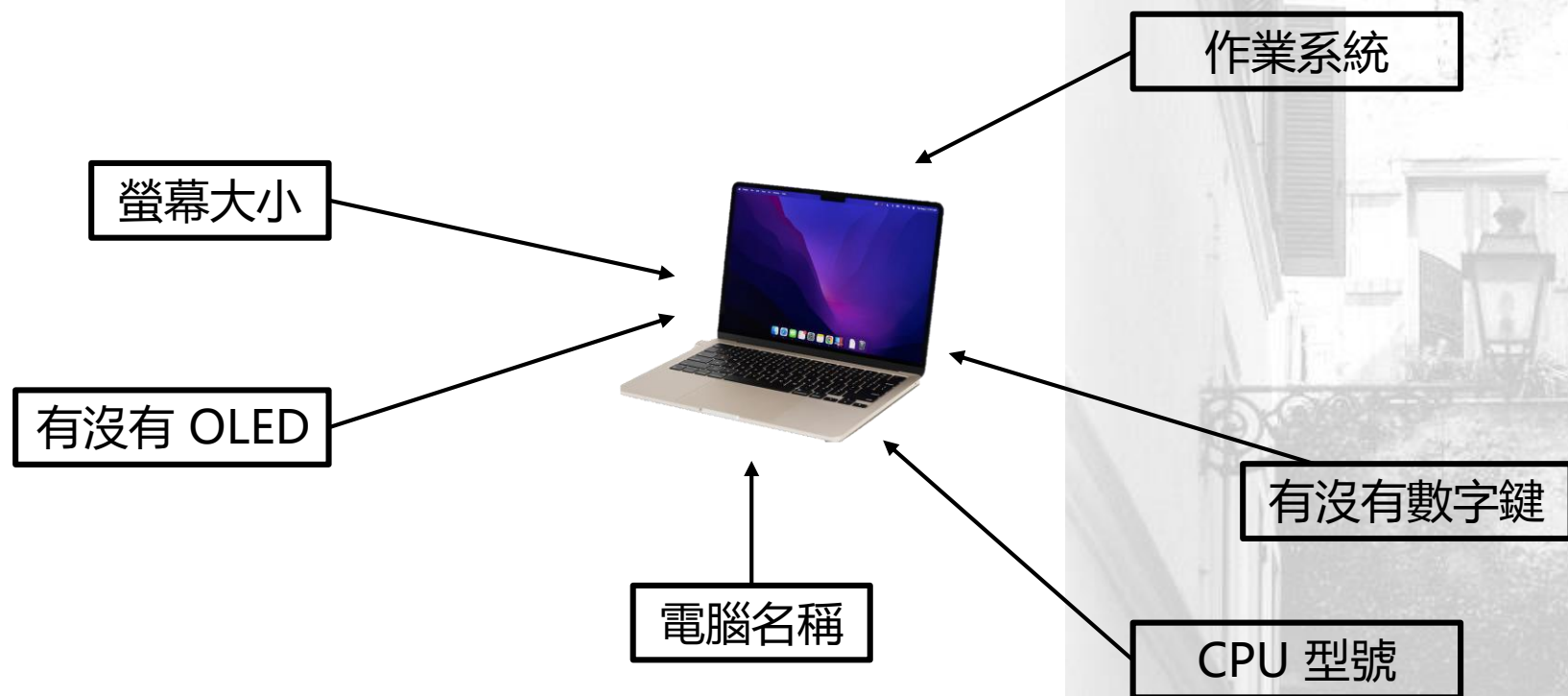
...

...



# 為什麼我們需要 OOP (3/8)

- 假設多了一台新的電腦





# 為什麼我們需要 OOP (4/8)

- 然後你要描述這兩台電腦...

電腦1\_作業系統 = Windows

電腦1\_螢幕大小 (吋) = 14

電腦1\_有沒有 OLED = 有

電腦1\_有沒有數字鍵 = 沒有

電腦1\_電腦名稱 = 這是一台酷電腦

電腦1\_CPU 型號 = Letni i9-48763 4.87GHz

電腦2\_作業系統 = MacOS

電腦2\_螢幕大小 (吋) = 14

電腦2\_有沒有 OLED = 有

電腦2\_有沒有數字鍵 = 沒有

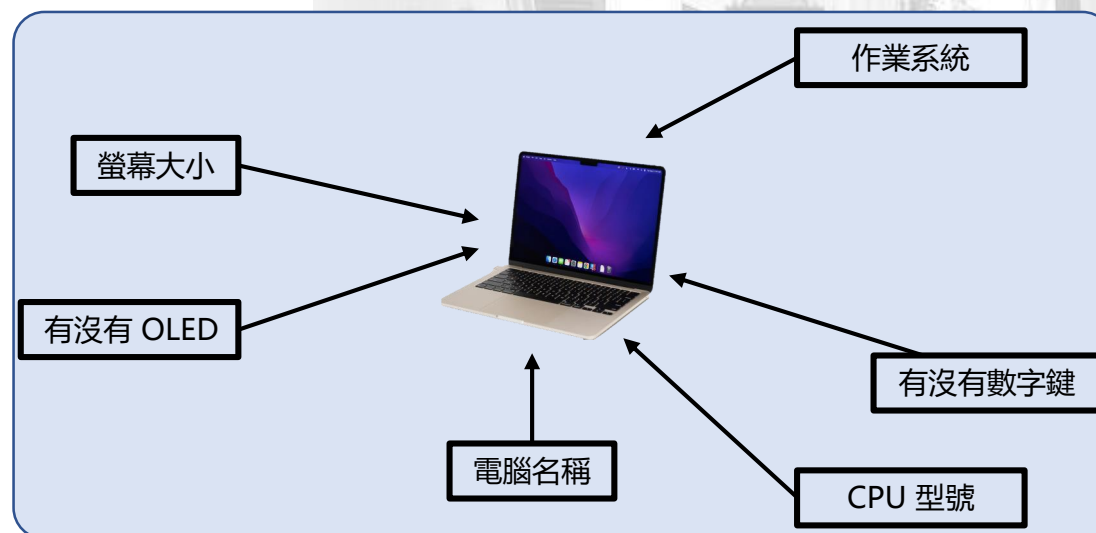
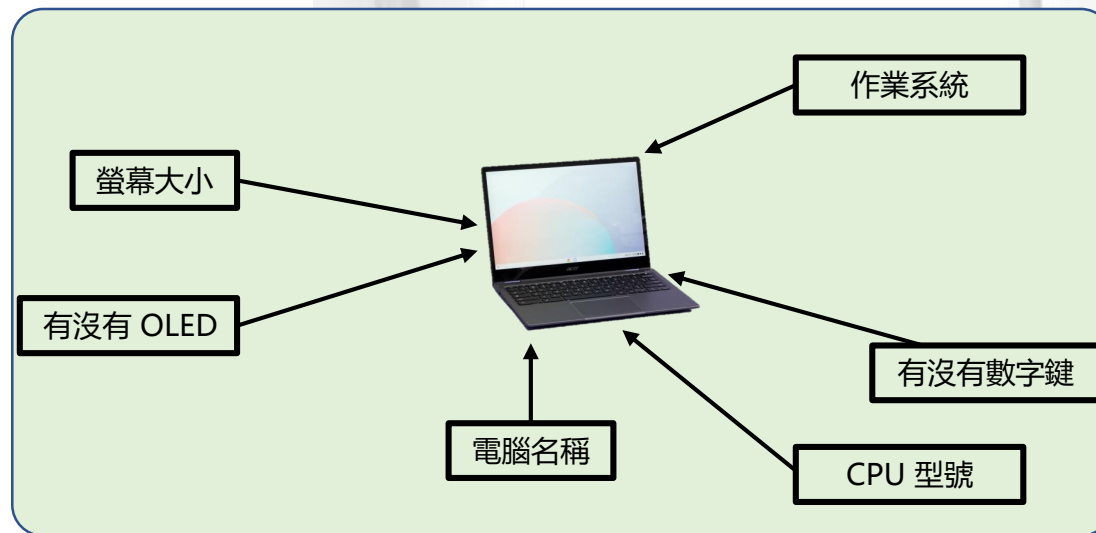
電腦2\_電腦名稱 = 這是另一台酷電腦

電腦2\_CPU 型號 = M9

# 在這裡實作輸出的細節

...

...



# 為什麼我們需要 OOP (4/8)

- 然後你要描述更多台電腦...

這樣的寫法沒有不行，但：

1. 難以維護
2. 每多一台電腦就要多個 6 行
3. 看起來是可以被簡化的，因為許多的屬性都是相似的

```
電腦1_作業系統 = Windows
電腦1_螢幕大小 (吋) = 14
電腦1_有沒有 OLED = 有
電腦1_有沒有數字鍵 = 沒有
電腦1_電腦名稱 = 這是一台酷電腦
電腦1_CPU 型號 = Letni i9-48763 4.87GHz
電腦2_作業系統 = MacOS
電腦2_螢幕大小 (吋) = 14
電腦2_有沒有 OLED = 有
電腦2_有沒有數字鍵 = 沒有
電腦2_電腦名稱 = 這是另一台酷電腦
電腦3_作業系統 = Windows
電腦3_螢幕大小 (吋) = 14
電腦3_有沒有 OLED = 有
電腦3_有沒有數字鍵 = 沒有
電腦3_電腦名稱 = 這是另一台酷電腦
電腦3_CPU 型號 = M9
電腦4_作業系統 = Windows
電腦4_螢幕大小 (吋) = 14
電腦4_有沒有 OLED = 有
電腦4_有沒有數字鍵 = 沒有
電腦4_電腦名稱 = 這是另一台酷電腦
電腦4_CPU 型號 = M9
...
```

```
...
電腦5_作業系統 = Windows
電腦5_螢幕大小 (吋) = 14
電腦5_有沒有 OLED = 有
電腦5_有沒有數字鍵 = 沒有
電腦5_電腦名稱 = 這是一台酷電腦
電腦5_CPU 型號 = Letni i9-48763 4.87GHz
電腦6_作業系統 = Windows
電腦6_螢幕大小 (吋) = 14
電腦6_有沒有 OLED = 有
電腦6_有沒有數字鍵 = 沒有
電腦6_電腦名稱 = 這是另一台酷電腦
電腦7_作業系統 = Windows
電腦7_螢幕大小 (吋) = 14
電腦7_有沒有 OLED = 有
電腦7_有沒有數字鍵 = 沒有
電腦7_電腦名稱 = 這是另一台酷電腦
電腦7_CPU 型號 = M9
電腦8_作業系統 = Windows
電腦8_螢幕大小 (吋) = 14
電腦8_有沒有 OLED = 有
電腦8_有沒有數字鍵 = 沒有
電腦8_電腦名稱 = 這是另一台酷電腦
電腦8_CPU 型號 = M9
```

# 在這裡實作輸出的細節

...

# 為什麼我們需要 OOP (5/8)

- 可以發現到，這邊有許多共同的屬性

```
電腦1_作業系統 = Windows
電腦1_螢幕大小 (吋) = 14
電腦1_有沒有 OLED = 有
電腦1_有沒有數字鍵 = 沒有
電腦1_電腦名稱 = 這是一台酷電腦
電腦1_CPU 型號 = Letni i9-48763 4.87GHz
```

```
電腦2_作業系統 = MacOS
電腦2_螢幕大小 (吋) = 14
電腦2_有沒有 OLED = 有
電腦2_有沒有數字鍵 = 沒有
電腦2_電腦名稱 = 這是另一台酷電腦
電腦2_CPU 型號 = M9
```

# 在這裡實作輸出的細節

```
電腦1_作業系統 = Windows
電腦1_螢幕大小 (吋) = 14
電腦1_有沒有 OLED = 有
電腦1_有沒有數字鍵 = 沒有
電腦1_電腦名稱 = 這是一台酷電腦
電腦1_CPU 型號 = Letni i9-48763 4.87GHz
電腦2_作業系統 = MacOS
電腦2_螢幕大小 (吋) = 14
電腦2_有沒有 OLED = 有
電腦2_有沒有數字鍵 = 沒有
電腦2_電腦名稱 = 這是另一台酷電腦
電腦3_作業系統 = Windows
電腦3_螢幕大小 (吋) = 14
電腦3_有沒有 OLED = 有
電腦3_有沒有數字鍵 = 沒有
電腦3_電腦名稱 = 這是另一台酷電腦
電腦3_CPU 型號 = M9
電腦4_作業系統 = Windows
電腦4_螢幕大小 (吋) = 14
電腦4_有沒有 OLED = 有
電腦4_有沒有數字鍵 = 沒有
電腦4_電腦名稱 = 這是另一台酷電腦
電腦4_CPU 型號 = M9
```

...

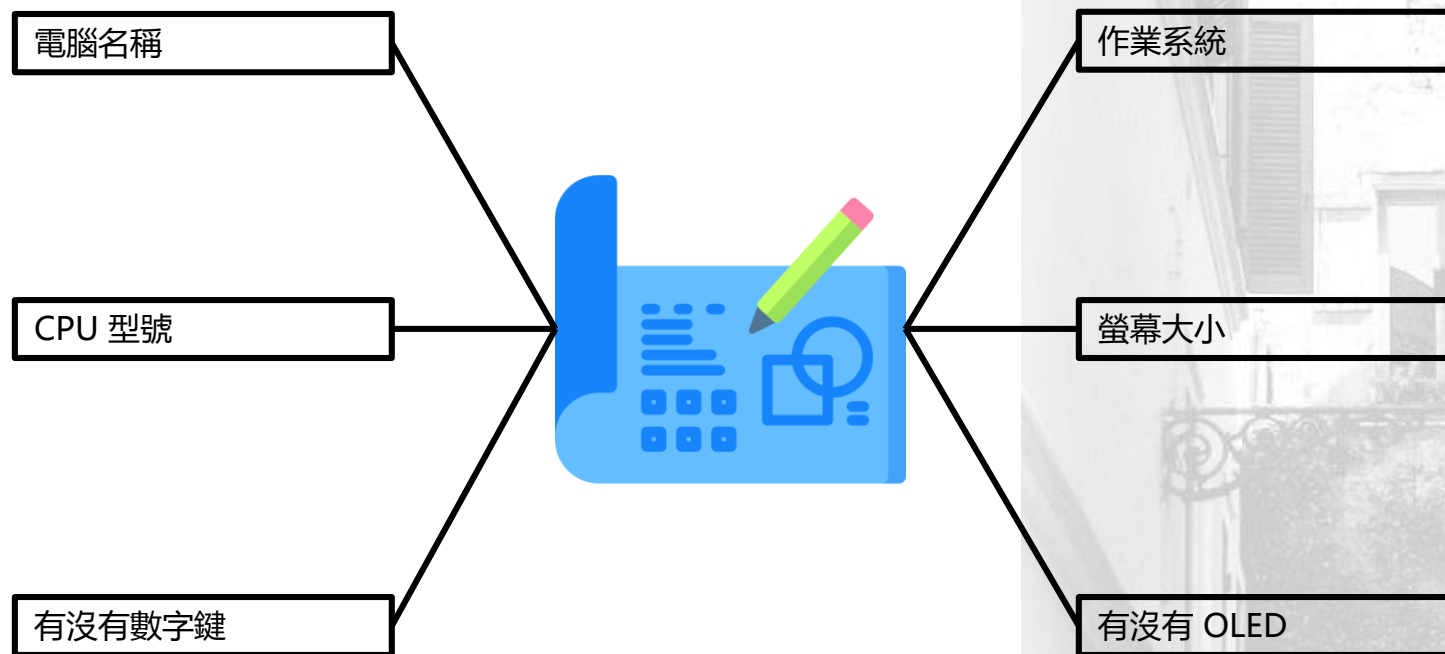
```
...
電腦5_作業系統 = Windows
電腦5_螢幕大小 (吋) = 14
電腦5_有沒有 OLED = 有
電腦5_有沒有數字鍵 = 沒有
電腦5_電腦名稱 = 這是一台酷電腦
電腦5_CPU 型號 = Letni i9-48763 4.87GHz
電腦6_作業系統 = Windows
電腦6_螢幕大小 (吋) = 14
電腦6_有沒有 OLED = 有
電腦6_有沒有數字鍵 = 沒有
電腦6_電腦名稱 = 這是另一台酷電腦
電腦7_作業系統 = Windows
電腦7_螢幕大小 (吋) = 14
電腦7_有沒有 OLED = 有
電腦7_有沒有數字鍵 = 沒有
電腦7_電腦名稱 = 這是另一台酷電腦
電腦7_CPU 型號 = M9
電腦8_作業系統 = Windows
電腦8_螢幕大小 (吋) = 14
電腦8_有沒有 OLED = 有
電腦8_有沒有數字鍵 = 沒有
電腦8_電腦名稱 = 這是另一台酷電腦
電腦8_CPU 型號 = M9
```

# 在這裡實作輸出的細節

...

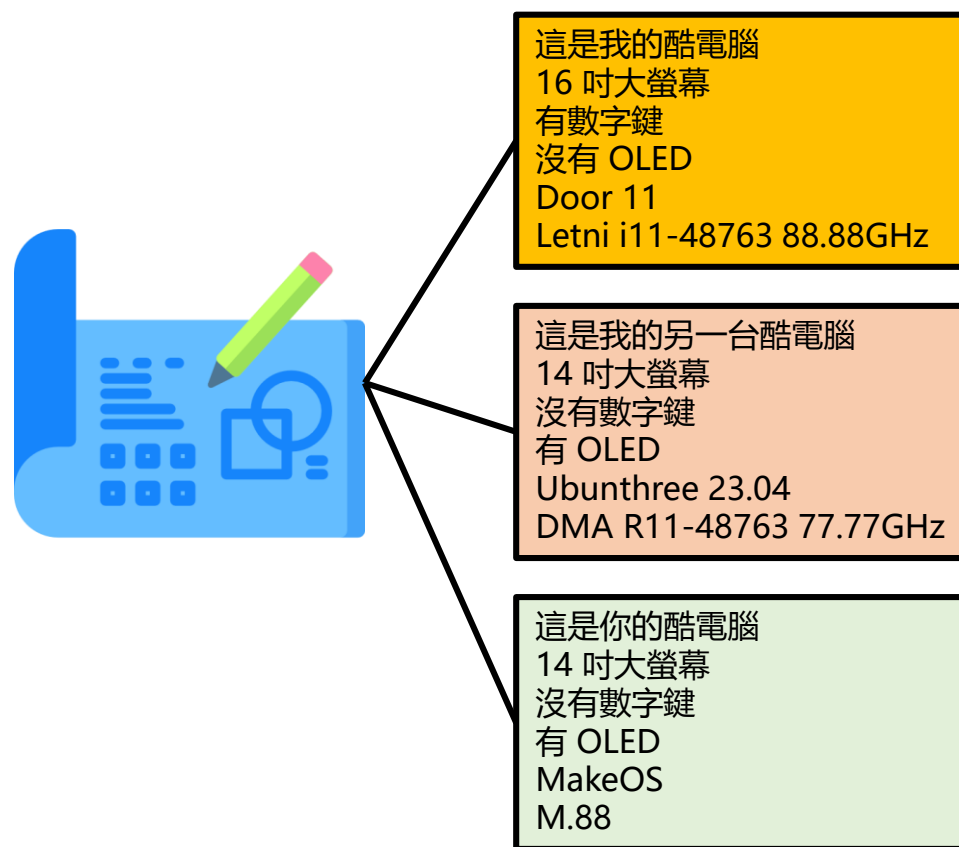
# 為什麼我們需要 OOP (6/8)

- 也許可以把這些共有屬性抽出來? (變成類別)



# 為什麼我們需要 OOP (7/8)


- 然後根據不同的特性，把電腦做出來（從類別實作成物件）





# 為什麼我們需要 OOP (8/8)

- 然後就能在程式上，三行解決在程式上需要處理三台不同電腦的問題！

 ("這是我的酷電腦", "16吋", true, false, "Doors 11", "Letni i11-48763")

=

這是我的酷電腦  
16 吋大螢幕  
有數字鍵  
沒有 OLED  
Doors 11  
Letni i11-48763




 ("這是另一台酷電腦", "14吋", false, true, "Ubunthree 23.04", "DMA R11-48763")

=

這是另一台酷電腦  
14 吋大螢幕  
沒有數字鍵  
有 OLED  
Ubunthree 23.04  
DMA R11-48763



 ("這是你的酷電腦", "14吋", false, true, "MakeOS", "M.88")

=

這是你的酷電腦  
14 吋大螢幕  
沒有數字鍵  
有 OLED  
MakeOS  
M.88



# 為什麼我們需要 OOP (補充)

- 然後你要描述更多台電腦...

這樣的寫法沒有不行，但：

1. 難以更新
2. 假設今天程式寫到一半我們發現遺漏了一個屬性  
Ex. 這台電腦有沒有支援Type C
3. 你可能需要一台一台電腦新增屬性  
Ex. 電腦1\_有沒有支援Type C = 有  
電腦2\_有沒有支援Type C = 沒
4. 結果漏加了一項...

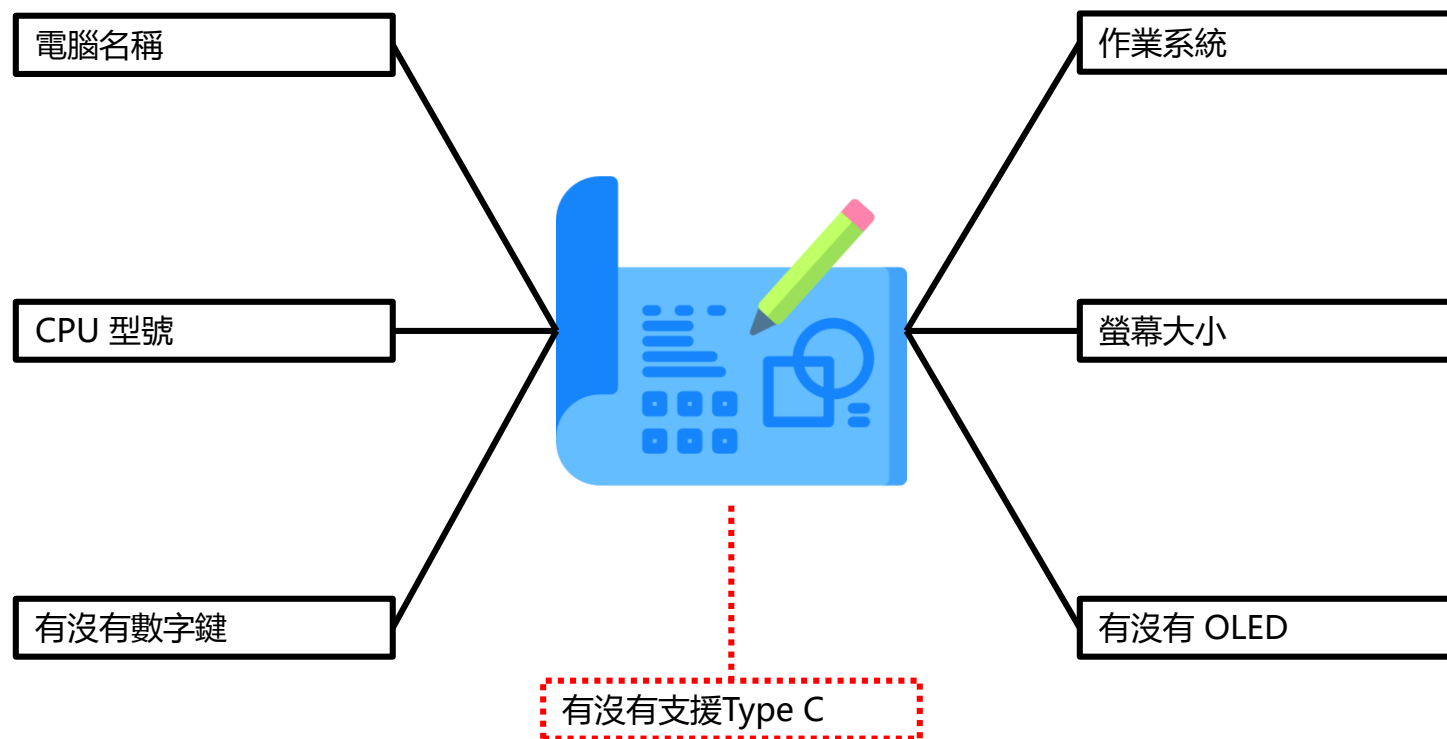
```
電腦1_作業系統 = Windows
電腦1_螢幕大小 (吋) = 14
電腦1_有沒有 OLED = 有
電腦1_有沒有數字鍵 = 沒有
電腦1_電腦名稱 = 這是一台酷電腦
電腦1_CPU 型號 = Letni i9-48763 4.87GHz
電腦2_作業系統 = MacOS
電腦2_螢幕大小 (吋) = 14
電腦2_有沒有 OLED = 有
電腦2_有沒有數字鍵 = 沒有
電腦2_電腦名稱 = 這是另一台酷電腦
電腦3_作業系統 = Windows
電腦3_螢幕大小 (吋) = 14
電腦3_有沒有 OLED = 有
電腦3_有沒有數字鍵 = 沒有
電腦3_電腦名稱 = 這是另一台酷電腦
電腦3_CPU 型號 = M9
電腦4_作業系統 = Windows
電腦4_螢幕大小 (吋) = 14
電腦4_有沒有 OLED = 有
電腦4_有沒有數字鍵 = 沒有
電腦4_電腦名稱 = 這是另一台酷電腦
電腦4_CPU 型號 = M9
...
```


```
...
電腦5_作業系統 = Windows
電腦5_螢幕大小 (吋) = 14
電腦5_有沒有 OLED = 有
電腦5_有沒有數字鍵 = 沒有
電腦5_電腦名稱 = 這是一台酷電腦
電腦5_CPU 型號 = Letni i9-48763 4.87GHz
電腦6_作業系統 = Windows
電腦6_螢幕大小 (吋) = 14
電腦6_有沒有 OLED = 有
電腦6_有沒有數字鍵 = 沒有
電腦6_電腦名稱 = 這是另一台酷電腦
電腦7_作業系統 = Windows
電腦7_螢幕大小 (吋) = 14
電腦7_有沒有 OLED = 有
電腦7_有沒有數字鍵 = 沒有
電腦7_電腦名稱 = 這是另一台酷電腦
電腦7_CPU 型號 = M9
電腦8_作業系統 = Windows
電腦8_螢幕大小 (吋) = 14
電腦8_有沒有 OLED = 有
電腦8_有沒有數字鍵 = 沒有
電腦8_電腦名稱 = 這是另一台酷電腦
電腦8_CPU 型號 = M9
```

# 在這裡實作輸出的細節

...

# 為什麼我們需要 OOP (補充)



 ("這是你的酷電腦", "14吋", false, true, "MakeOS", "M.88", **True**) =

這是你的酷電腦  
14 吋大螢幕  
沒有數字鍵  
有 OLED  
MakeOS  
M.88  
支援Type C

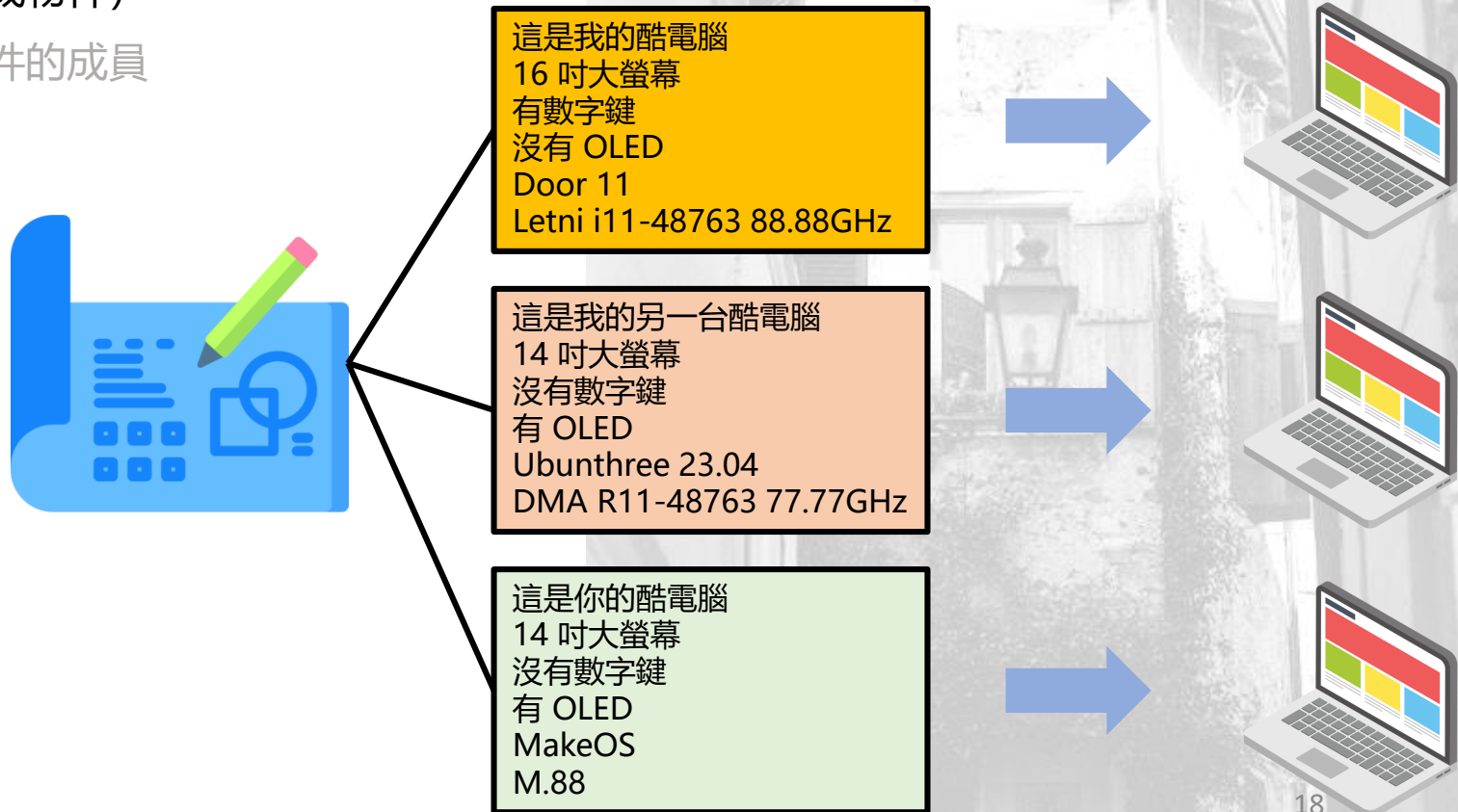


# 什麼是 OOP



# 什麼是 OOP

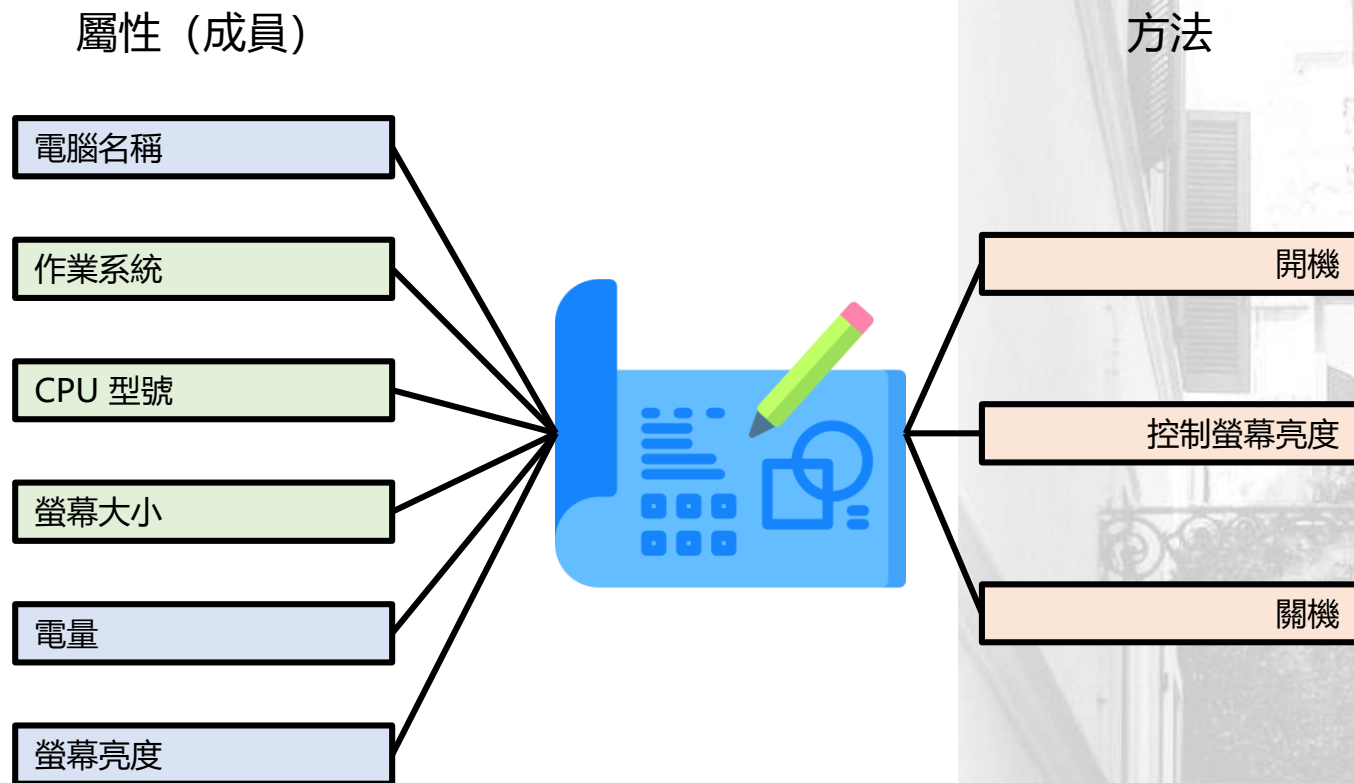
- 具有物件概念的程式設計典範<sup>[1]</sup>
  - 換句話說，把你想到的東西變成藍圖（規劃類別）
  - 然後根據藍圖來實作你想要的東西（變成物件）
  - 你可以去使用物件內的方法，來控制物件的成員





# 什麼是 OOP

- 規劃類別？

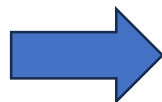


# 什麼是 OOP

- 化成物件?



這是我的酷電腦  
16 吋大螢幕  
有數字鍵  
沒有 OLED  
Door 11  
Letni i11-48763 88.88GHz

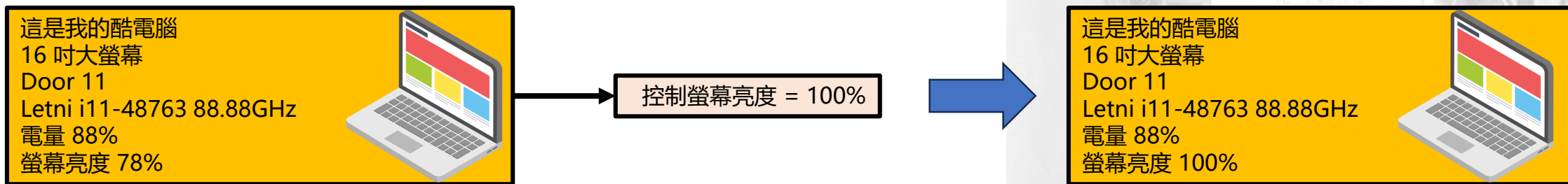


這是我的酷電腦  
16 吋大螢幕  
Door 11  
Letni i11-48763 88.88GHz  
電量 88%  
螢幕亮度 78%



# 什麼是 OOP

- 控制物件？



# 什麼是 OOP

- 實際上，我們在程式上可能會怎麼做呢？



=



(“這是我的酷電腦”, “16 吋大螢幕”, “Door 11”,  
“Letni i11-48763 88.88GHz”, 88%, 78%)

電腦名稱

=



取得電腦名稱

回傳：這是我的酷電腦



設定電腦螢幕亮度 (66%)

電腦螢幕變成了從 78% 變成了 66%

電腦亮度

=



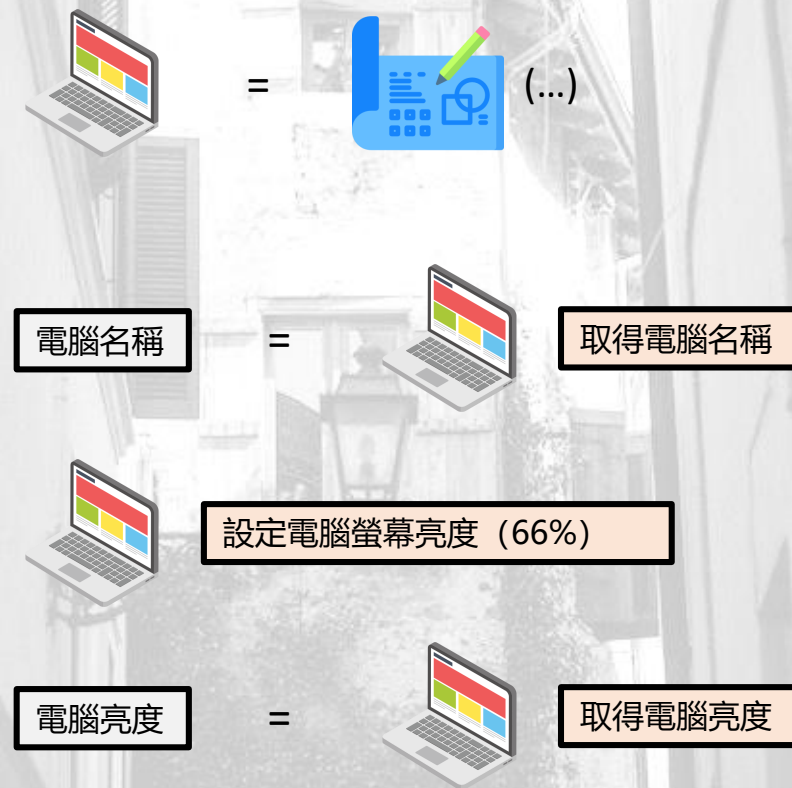
取得電腦亮度

回傳：66%

# 什麼是 OOP

- 更具體化

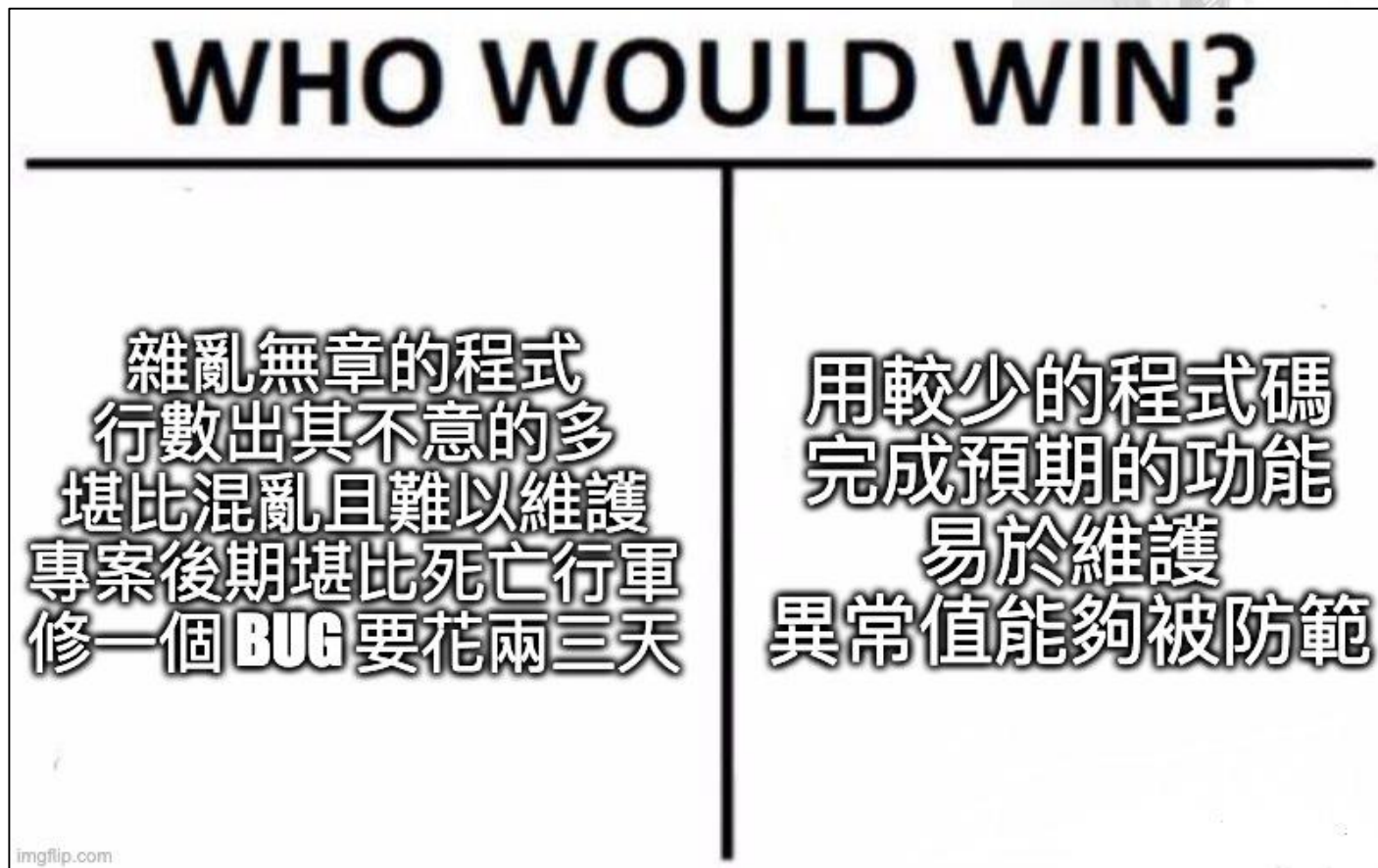
```
Computer computer1 = Computer("這是我的酷電腦", "16 吋大螢幕",  
"Doors 11", "Letni i11-48763 88.88GHz", 88%, 78%);  
  
string name = computer1.getName(); // Return: "這是我的酷電腦"  
  
computer1.setScreenBrightness(0.66); // 設定螢幕亮度變成 66%  
  
double screenBrightness = computer1.getScreenBrightness(); // Return: 0.66
```





# 什麼是 OOP

- 透過 OOP 可以幫助我們更直觀的處理物件的操作，讓程式變得更加簡潔與乾淨。



# 幾個 OOP 的主題



# 幾個 OOP 的主題

- 類別 (Class) 與物件 (Object)
- 封裝 (Encapsulation)
- 繼承 (Inheritance)
- 多型 (Polymorphism)
- 組合/複合 (Composition)
- 介面 (Interface)
- 工廠模式 (Factory)
- 依賴注入 (Dependency Injection)





# 封裝 (Encapsulation)



# 封裝

- 為什麼我們需要學習封裝？



=



("這是我的酷電腦", "16 吋大螢幕", "Door 11",  
"Letni i11-48763 88.88GHz", 88%, 78%)

電腦名稱

=



取得電腦名稱

回傳：這是我的酷電腦



設定電腦 CPU 型號為 M11

設定電腦 CPU 型號為 M11

這不合理



# 封裝

- 也就是說，我們需要防範掉一些東西能否被修改，以及怎麼樣是正確的修改。



=



("這是我的酷電腦", "16 吋大螢幕", "Door 11",  
"Letni i11-48763 88.88GHz", 88%, 78%)



設定電腦 CPU 型號為 M11

設定電腦 CPU 型號為 M11



設定電腦名稱為「這是被更改過名的電腦」

設定電腦名稱為「這是被更改過名的電腦」



設定螢幕亮度為 400%

設定螢幕亮度為 400%

不可以  
CPU 不能被修改

可以

不可以  
螢幕亮度介於 0% 到  
100%

# 封裝

- 如何實踐？



方法名稱：設定螢幕亮度為 E

如果 E 小於 0 或大於 100 就拋出例外

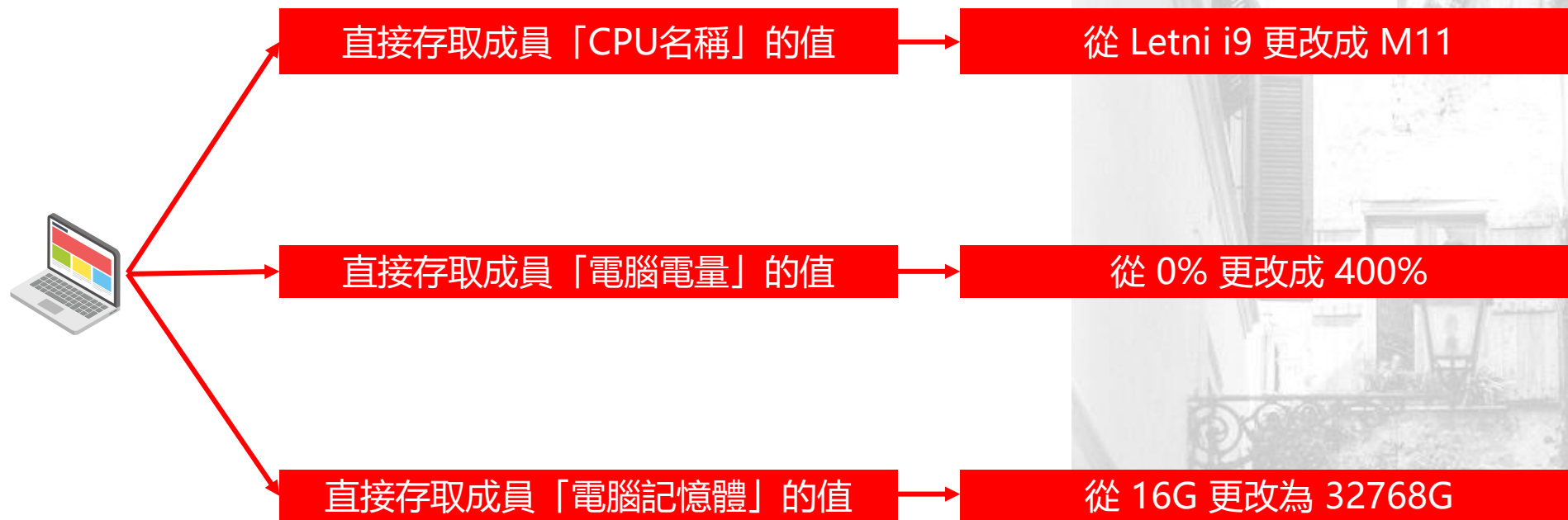
設定螢幕亮度為 E

方法名稱：取得電腦的螢幕亮度

回傳螢幕亮度

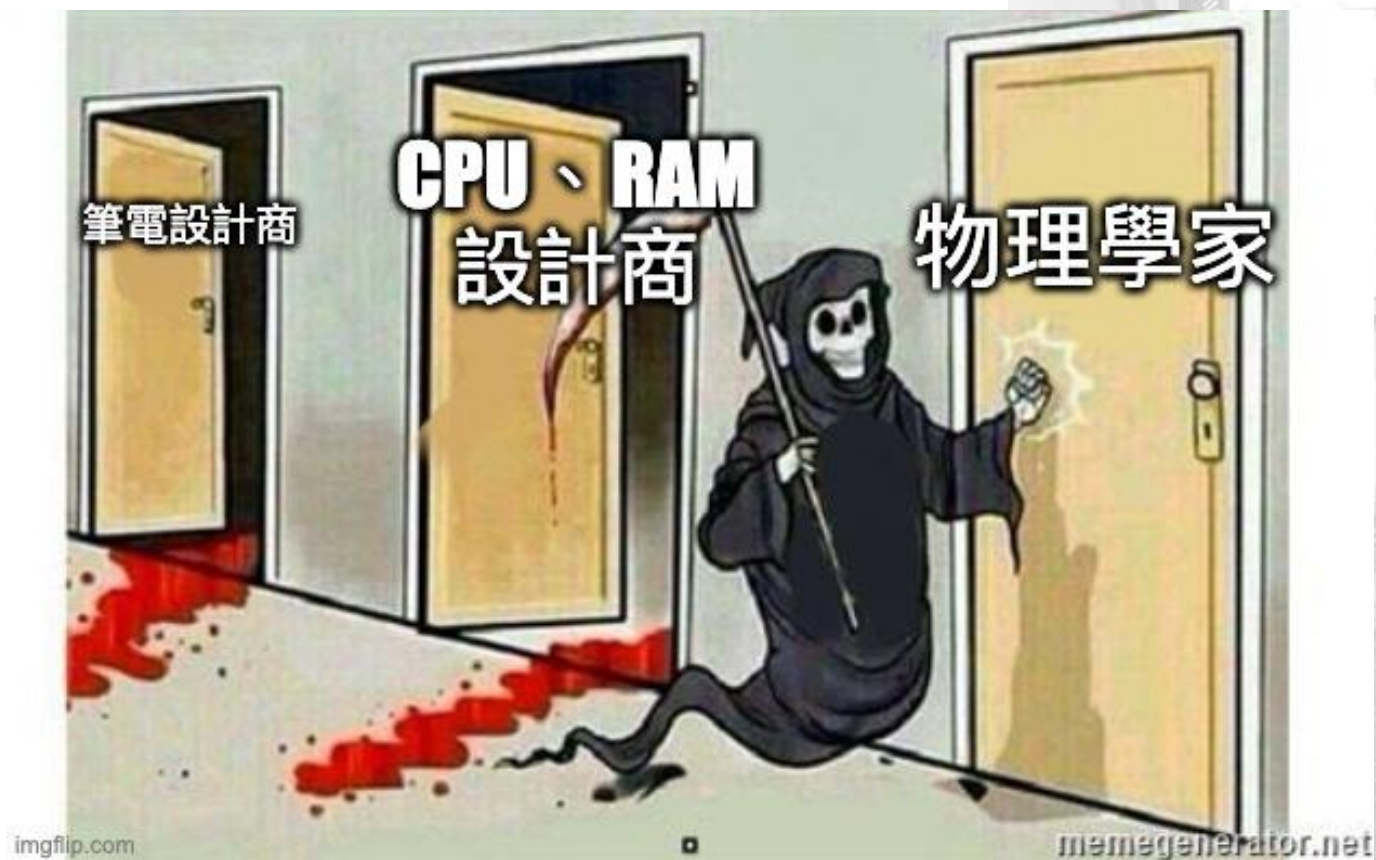
# 封裝

- 如果沒有實踐？



# 封裝

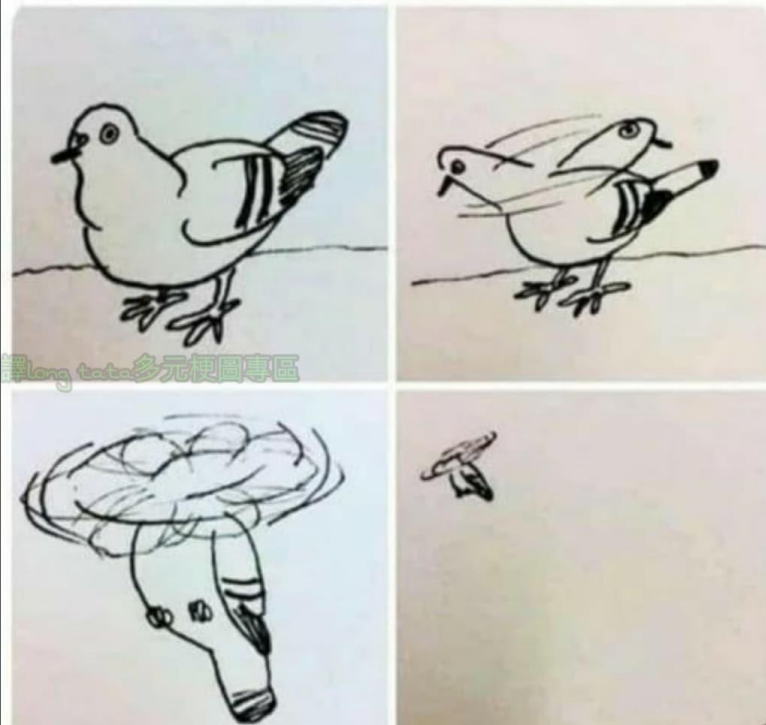
- 然後就... 全球大失業



# 封裝

- OK 那至少我在寫程式的時候盡量避免異常值出現的可能就好
- OK 但 Bug 可能會層出不窮。
- 記得這一門課主要是要讓程式寫得更好，不是更需要細心。

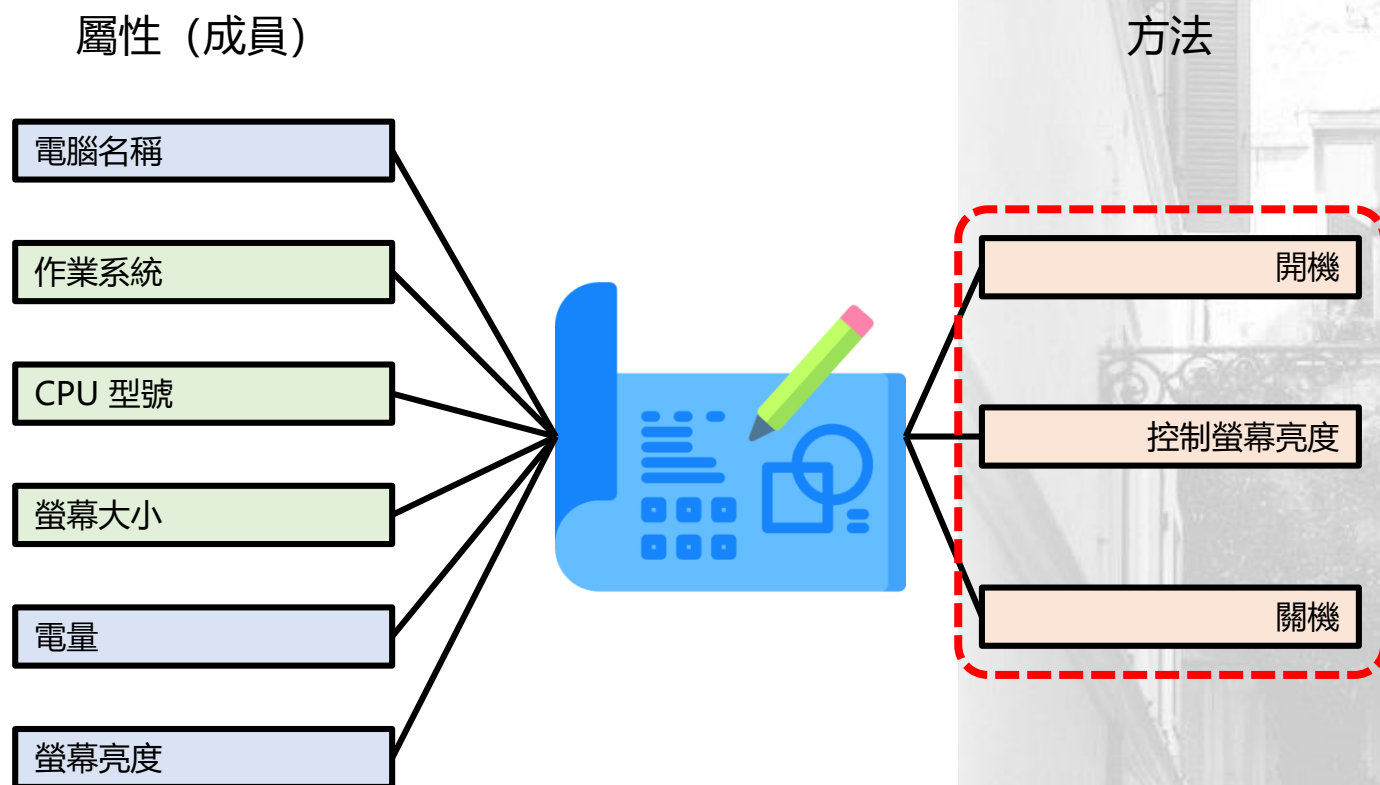
當你寫出來的程式一團糟  
但他還可以運作





# 封裝

- 另外一個優點：方便
  - 將方法的實作方式隱藏起來
  - 只要會用就好，不用理會實作細節



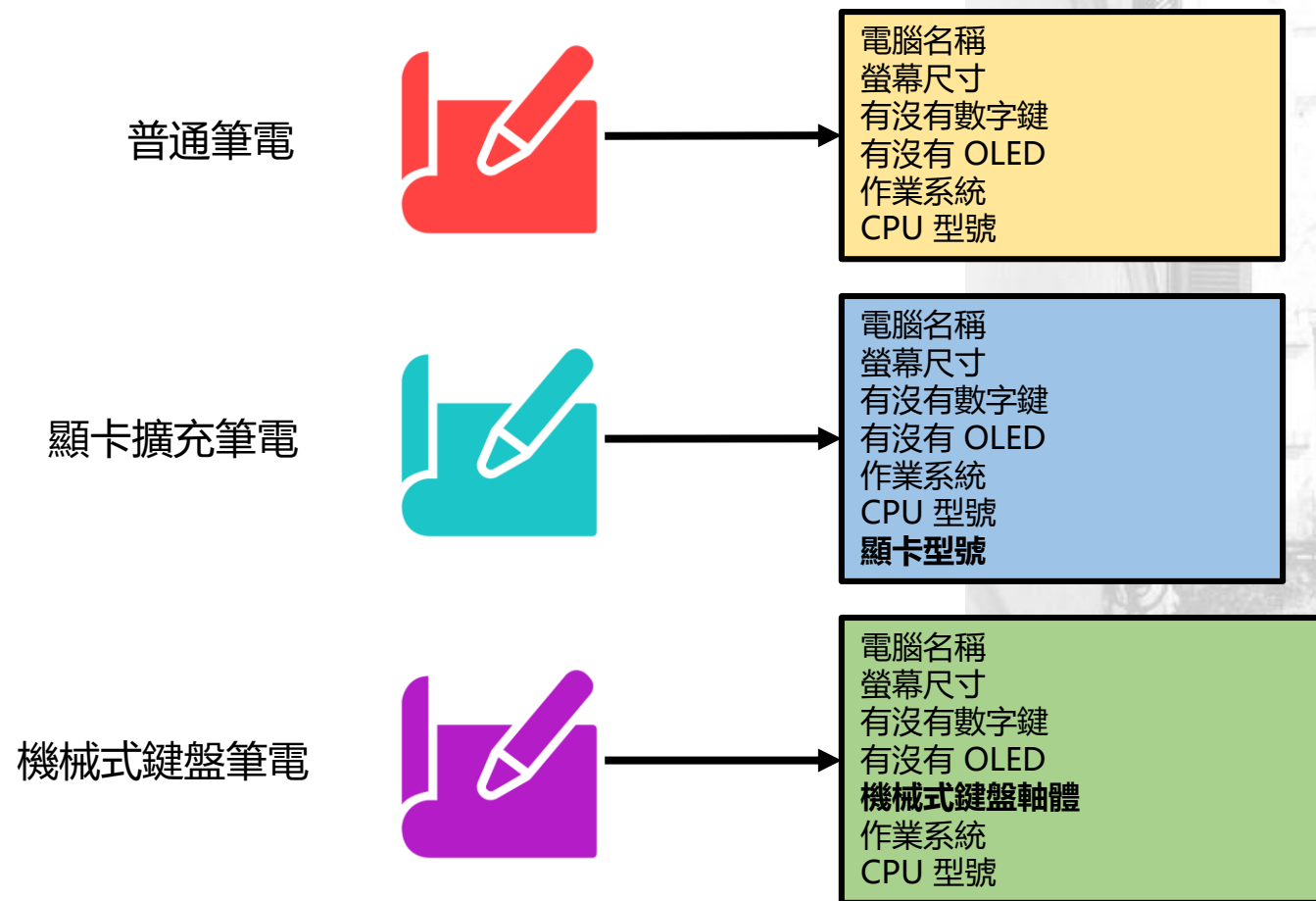


# 繼承 (Inheritance)



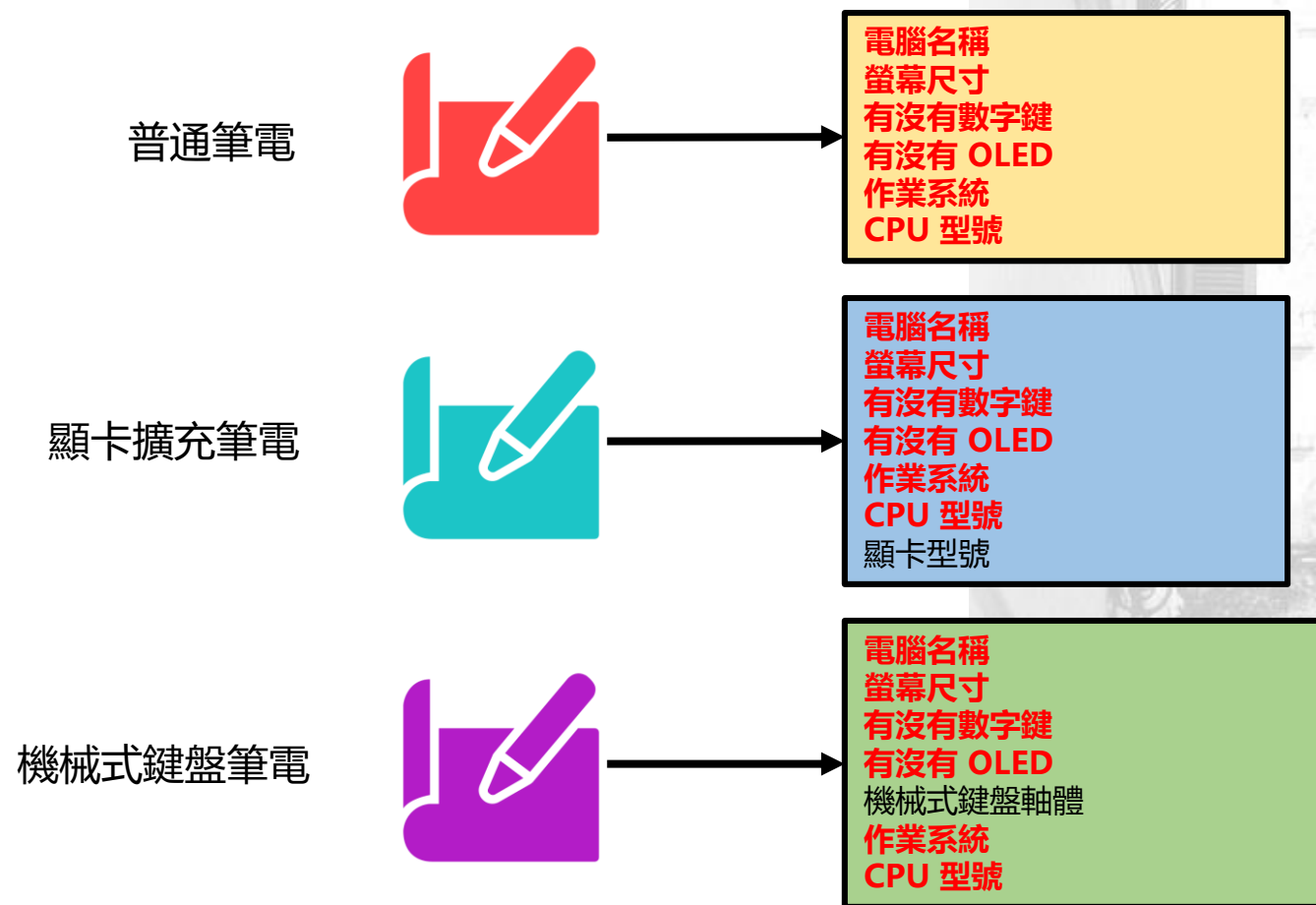
# 繼承 (1/10)

- 為什麼我們需要學習繼承？



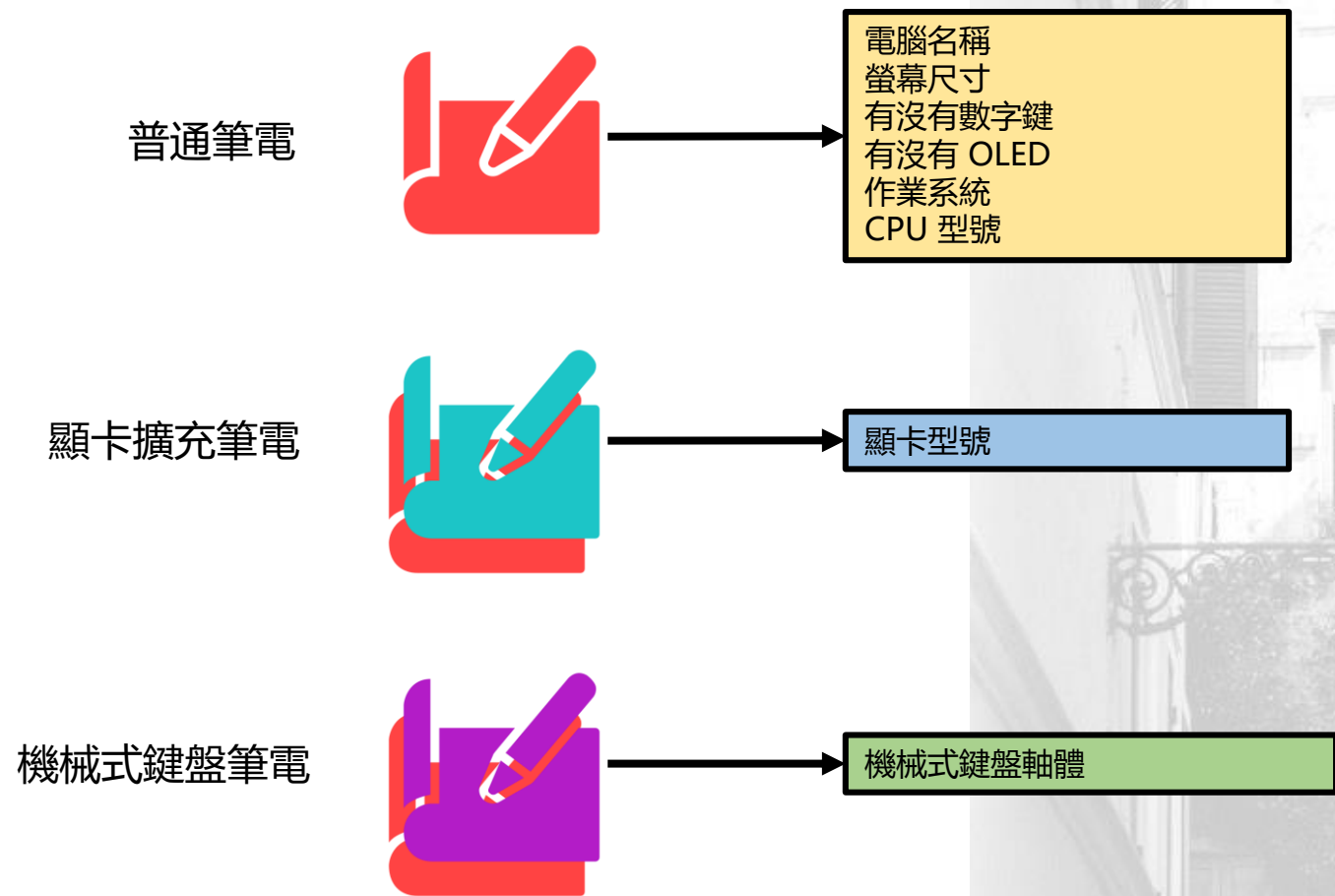
# 繼承 (2/10)

- 可以發現這邊有很多共同的屬性



# 繼承 (3/10)

- 如果我們拿紅色圖紙的成員來擴充，似乎可以減少很多重複的東西？



# 繼承 (4/10)

- 繼承可以用來繼承父類所有的成員與方法，並且可以自己擴充該子類的成員與方法
- 拿現有的東西進行擴充！

普通筆電



電腦名稱  
螢幕尺寸  
有沒有數字鍵  
有沒有 OLED  
作業系統  
CPU 型號

顯卡擴充筆電



顯卡型號

機械式鍵盤筆電

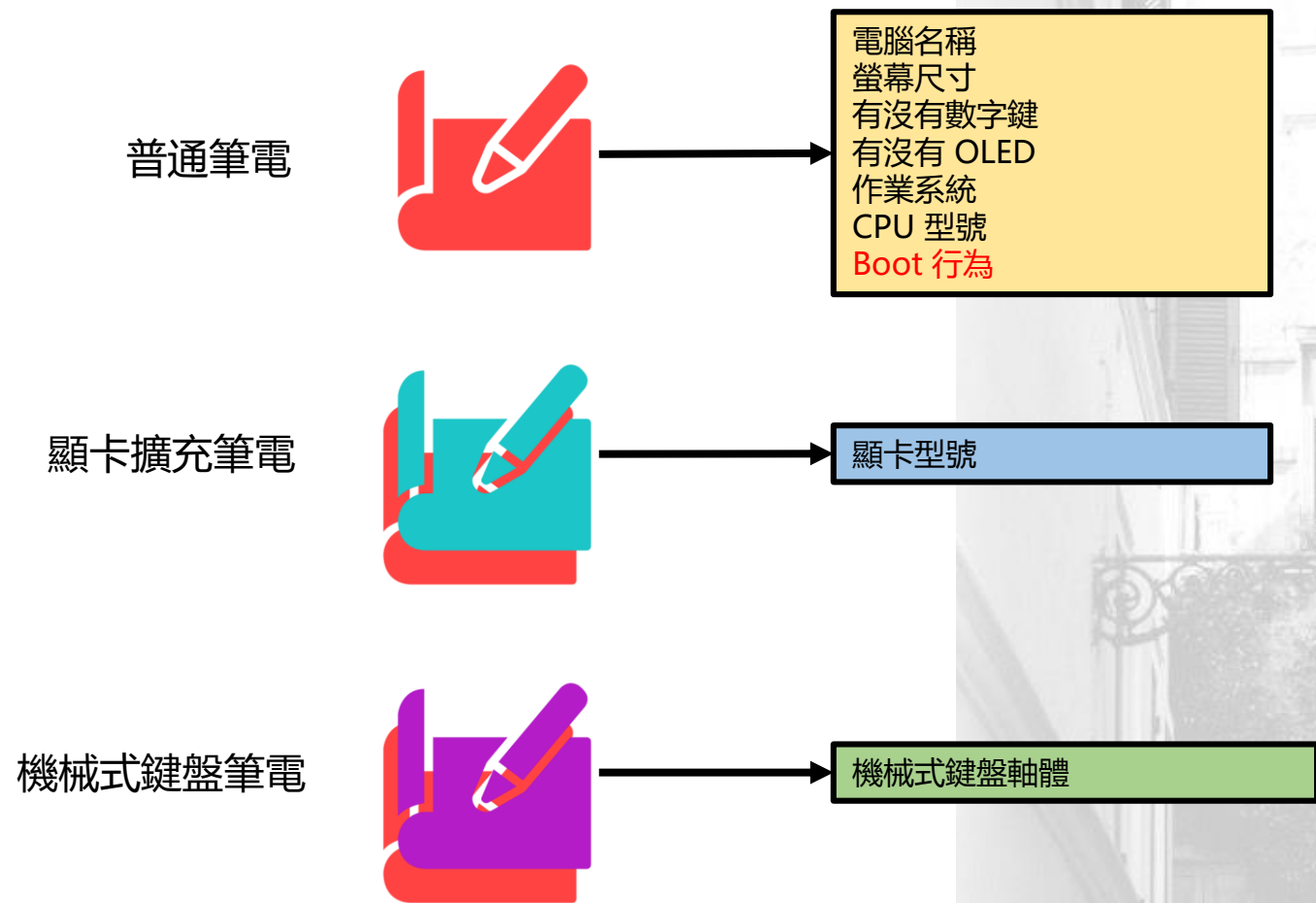


機械式鍵盤軸體



# 繼承 (5/10)

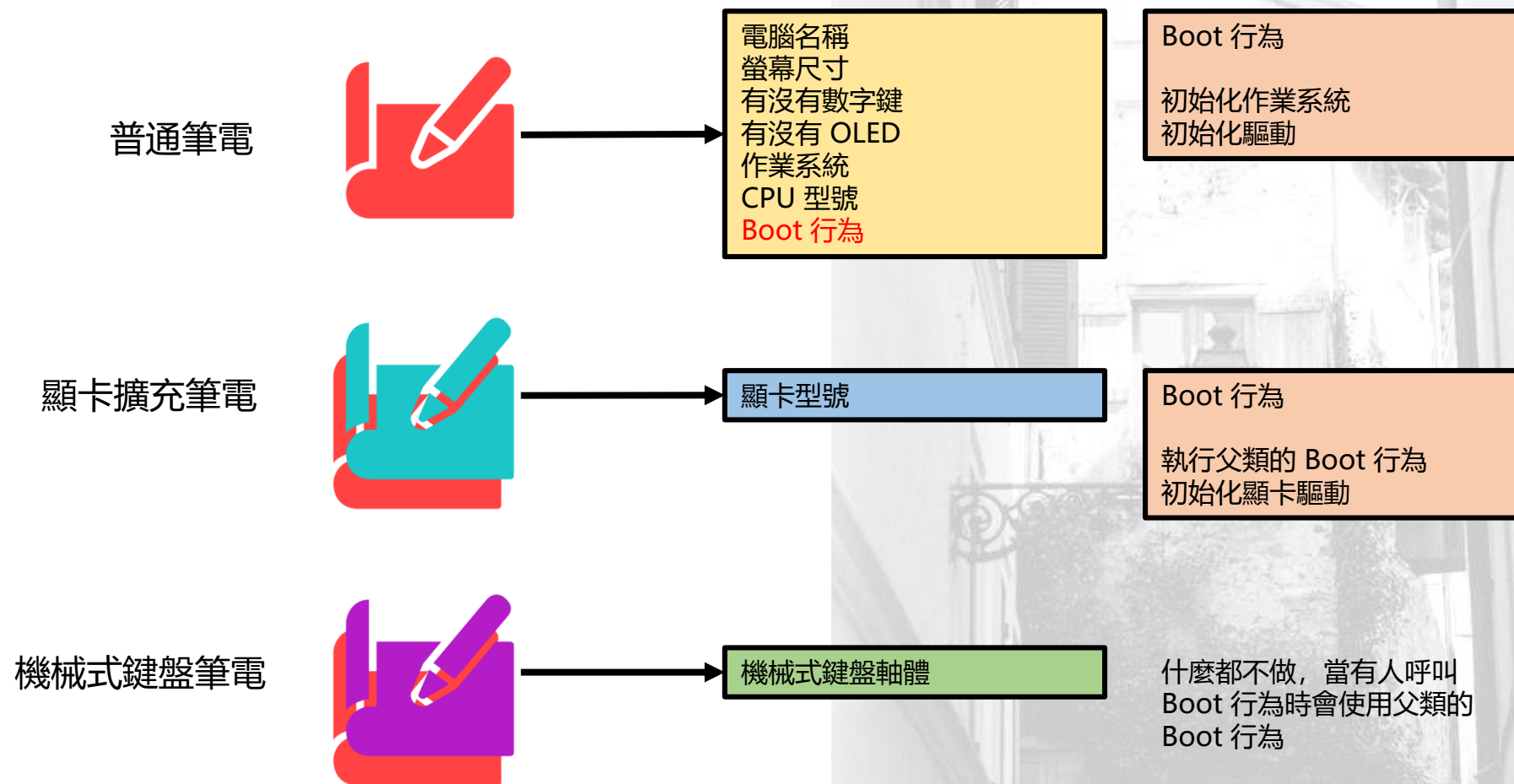
- 另外繼承也同時繼承了方法





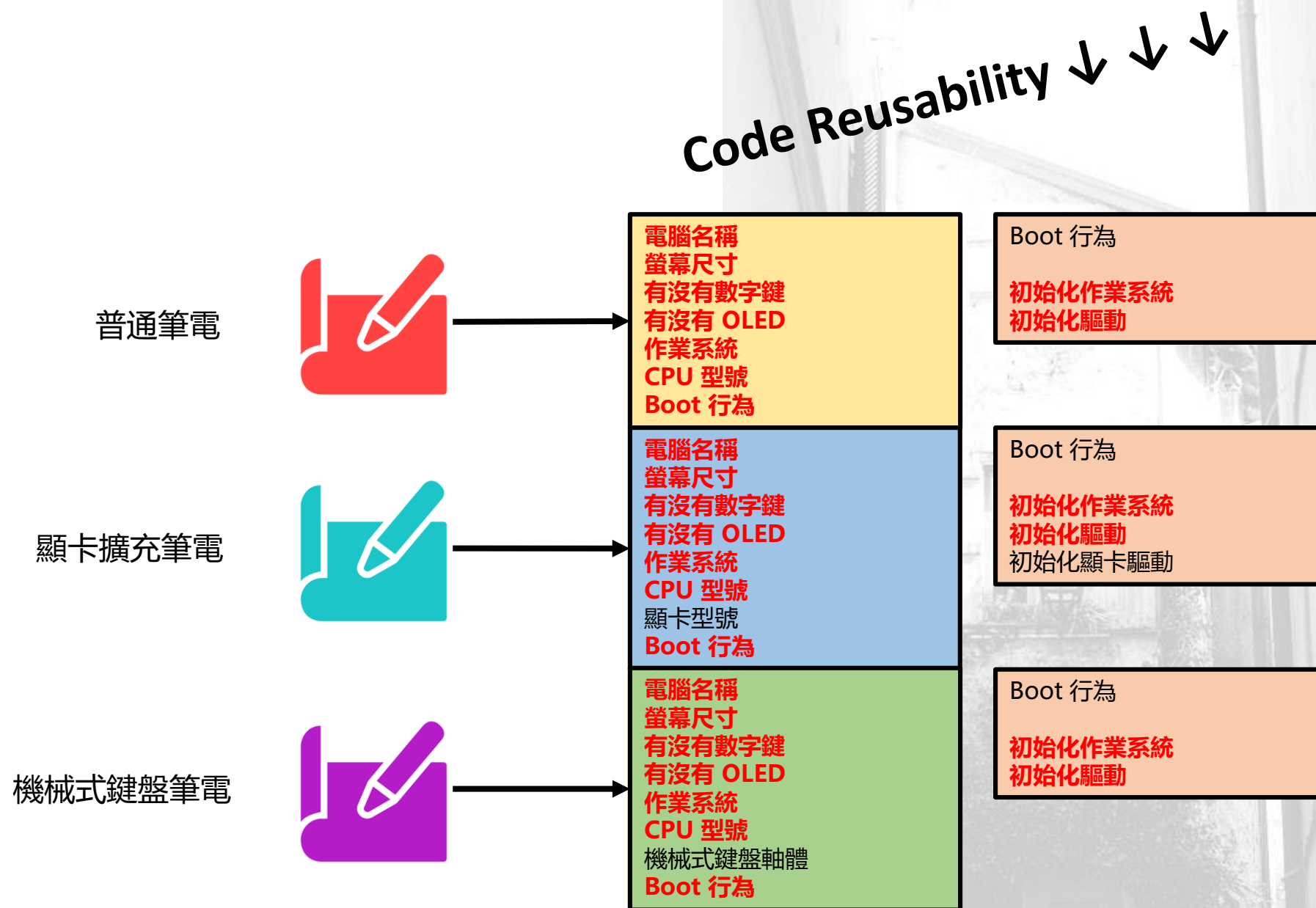
# 繼承 (6/10)

- 同時它們也能選擇要用自己的 Boot 行為，還是要用父類的 Boot 行為



# 繼承 (7/10)

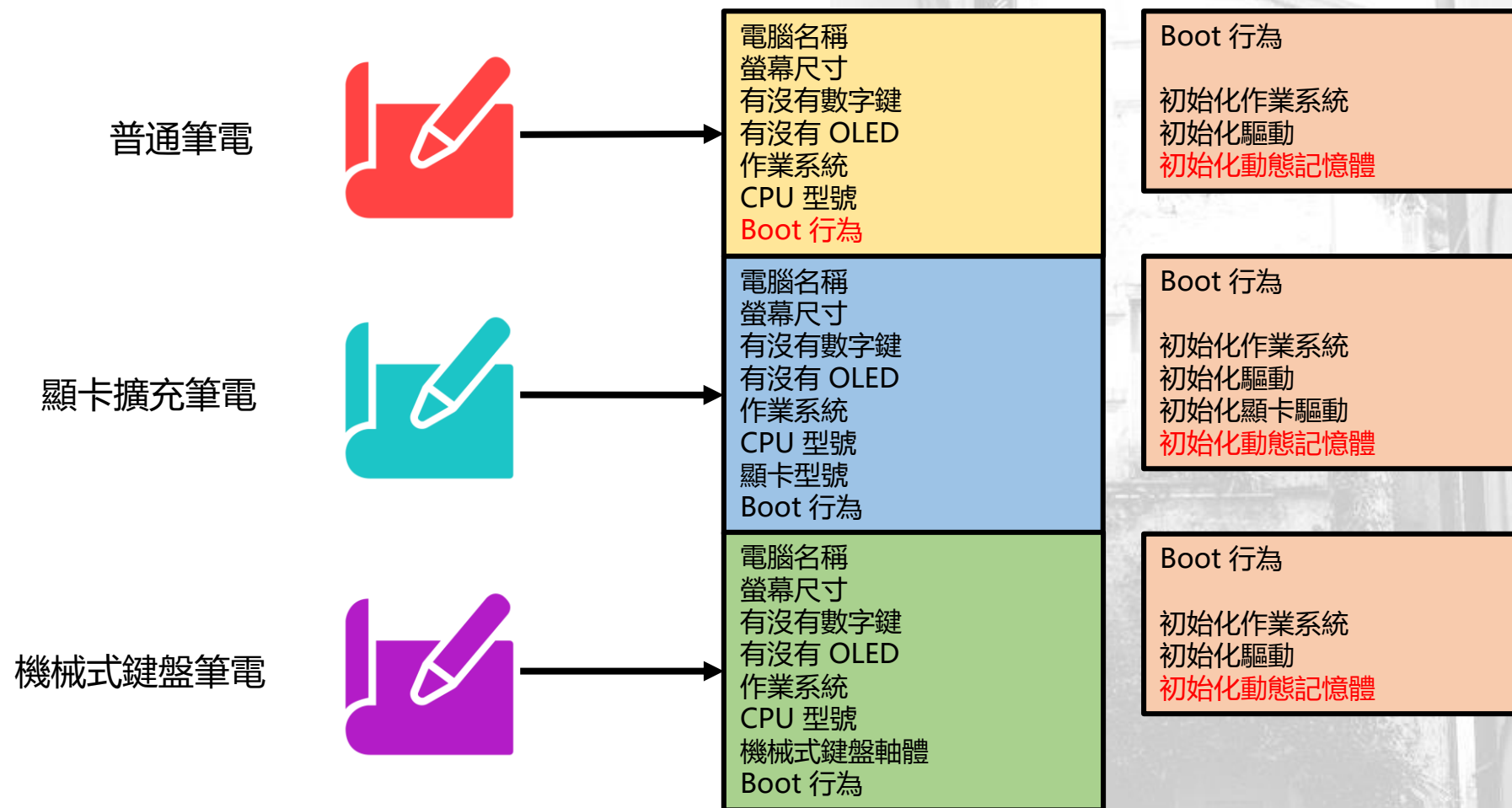
- 如果沒有繼承



# 繼承 (8/10)

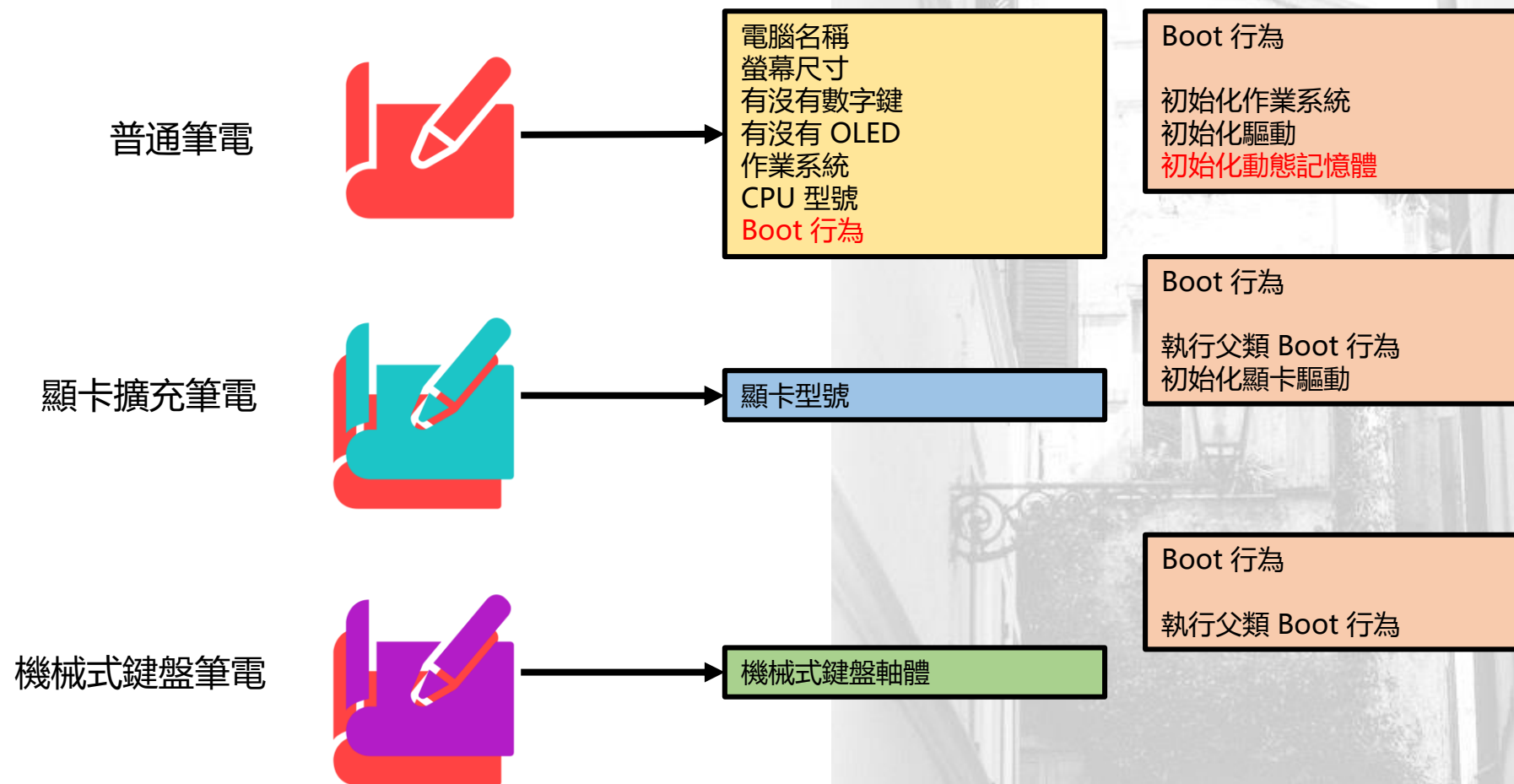
- 你需要同時更動三個類別預期都要執行的程式碼。

Code Extensibility ↓ ↓ ↓



# 繼承 (9/10)

- 但如果有繼承之後，你只需要更動一個。



# 繼承 (10/10)

- 透過繼承的方式，我們可以
  - 利用已經有的類別 (Reusability)
  - 也能夠擴充自己想要的其他成員 (Extensibility)



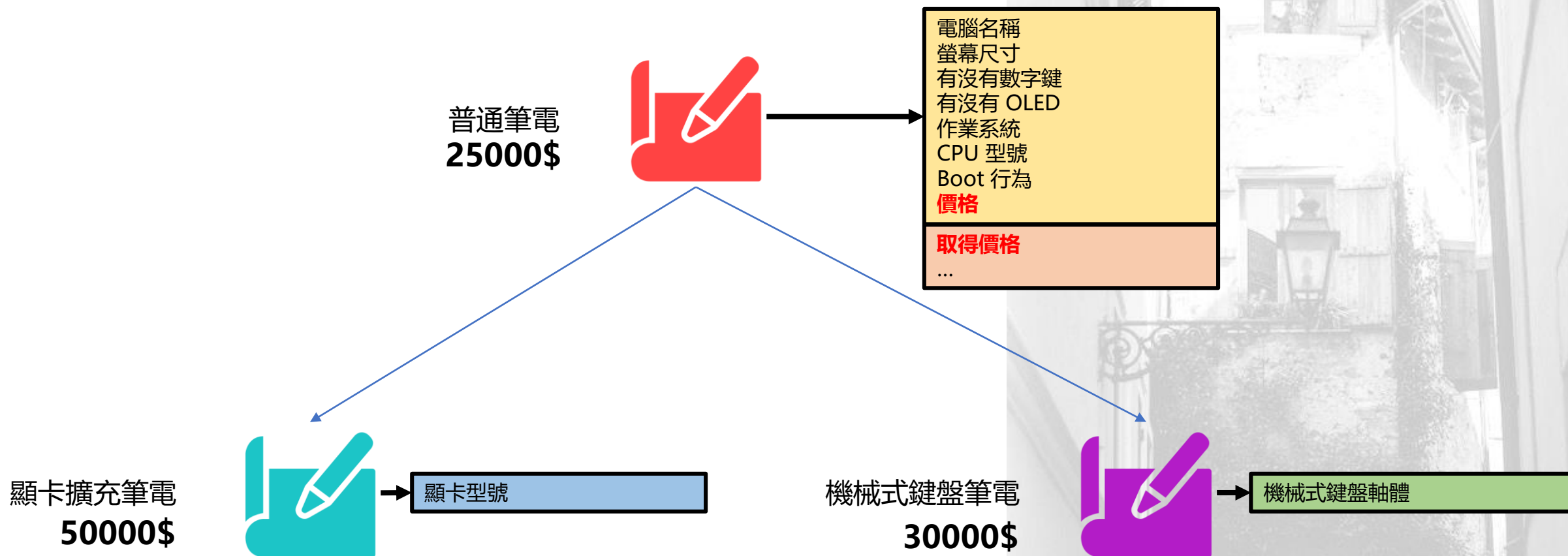
# 多型 (Polymorphism)





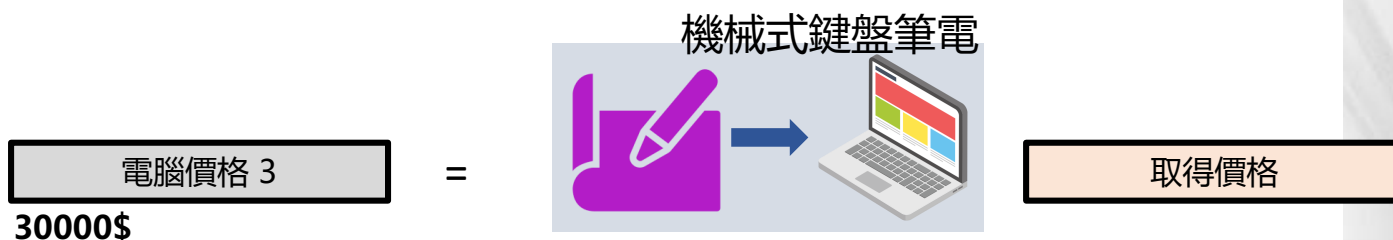
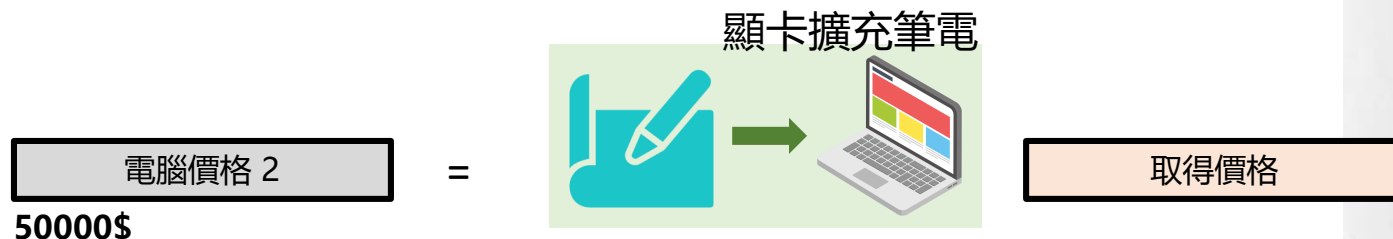
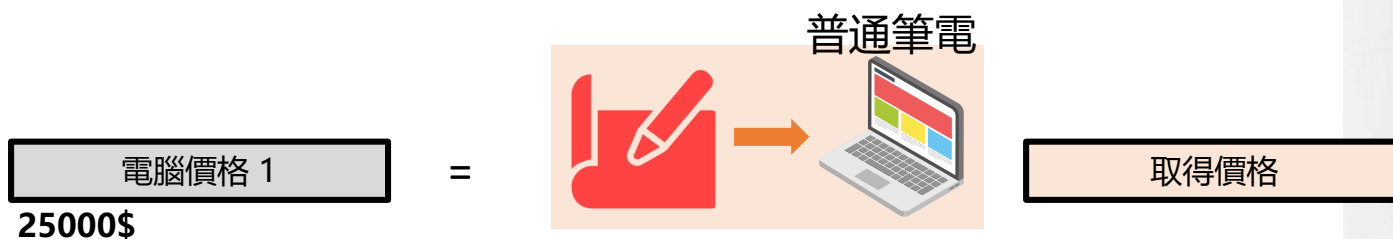
# 多型 (1/6)

- 先前提到物件的繼承，對於多型而言，他們有共同的類別函數。
- 舉個例子：我們有三臺不同的筆電，這些筆電有各自的價格，且有取得價格的函數。



## 多型 (2/6)

- 通常來說，如果我們需要去統計這些電腦的價格，可能要耗費不少的行數去實踐取得價格。



計算成本 = 顯卡成本 +  
(記憶體價格\*數量) +  
power成本 +  
...

計算成本 = 鍵盤成本 +  
(記憶體價格\*數量) +  
其他硬體設備成本 +  
...

# 多型 (3/6)

- 如果沒有使用多型，會遇到什麼問題？
  - 兩個禮拜後，你要**加需求**（新增 SSD擴充筆電）
    - 整份程式中到處修改（維護性↓， Maintainability）
    - 真實中，不會只有取得價格功能而已（可讀性↓， Readability）
    - 需要修改現有code，才能新增新的功能（擴充性↓， Expandability）

```
if 顯卡筆電{
    成本 = xxx + xxx + xxx;
    Return 成本;
}

elseif 機械鍵盤{
    成本 = yyy + yyy + yyy;
    Return 成本;
}

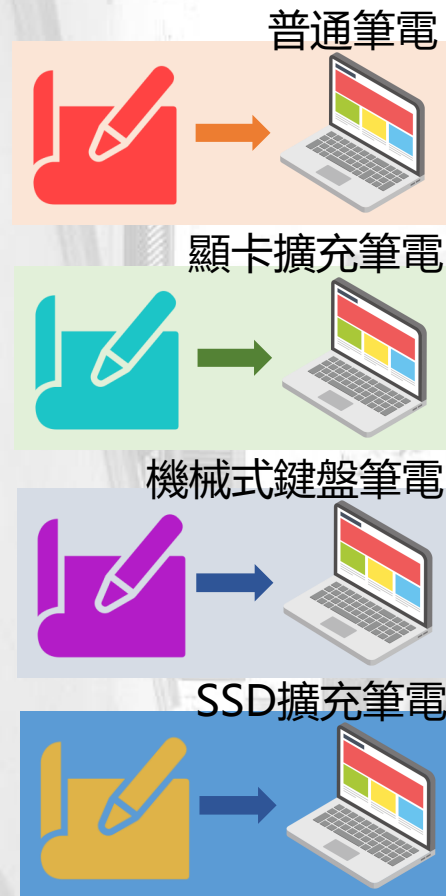
else{
    //普通筆電
    成本 = zzz + zzz + zzz;
}
```

```
if 顯卡筆電{
    成本 = xxx + xxx + xxx;
    Return 成本;
}

elseif 機械鍵盤{
    成本 = yyy + yyy + yyy;
    Return 成本;
}

elseif SSD筆電{
    ...
}

else{
    //普通筆電
    成本 = zzz + zzz + zzz;
}
```



取得價格

取得價格

取得價格

# 多型 (4/6)

- 如果沒有使用多型，會遇到什麼問題？
  - 兩個禮拜後，你要**加需求**（SSD擴充筆電）
    - 整份程式中到處修改（維護性↓， Maintainability）
    - 真實中，不會只有取得價格功能而已（可讀性↓， Readability）
    - 一直需要修改現有code，才能新增新的功能（擴充性↓， Expandability）
      - 源源不絕的Bug...

```
//取得價格
if 顯卡筆電{
    成本 = xxx + xxx + xxx;
    Return 成本;
}

elseif 機械鍵盤{
    成本 = yyy + yyy + yyy;
    Return 成本;
}

elseif SSD筆電{
    ...
}

else{
    //普通筆電
    成本 = zzz + zzz + zzz;
}

//其他功能1
if 顯卡筆電{
    ...
}

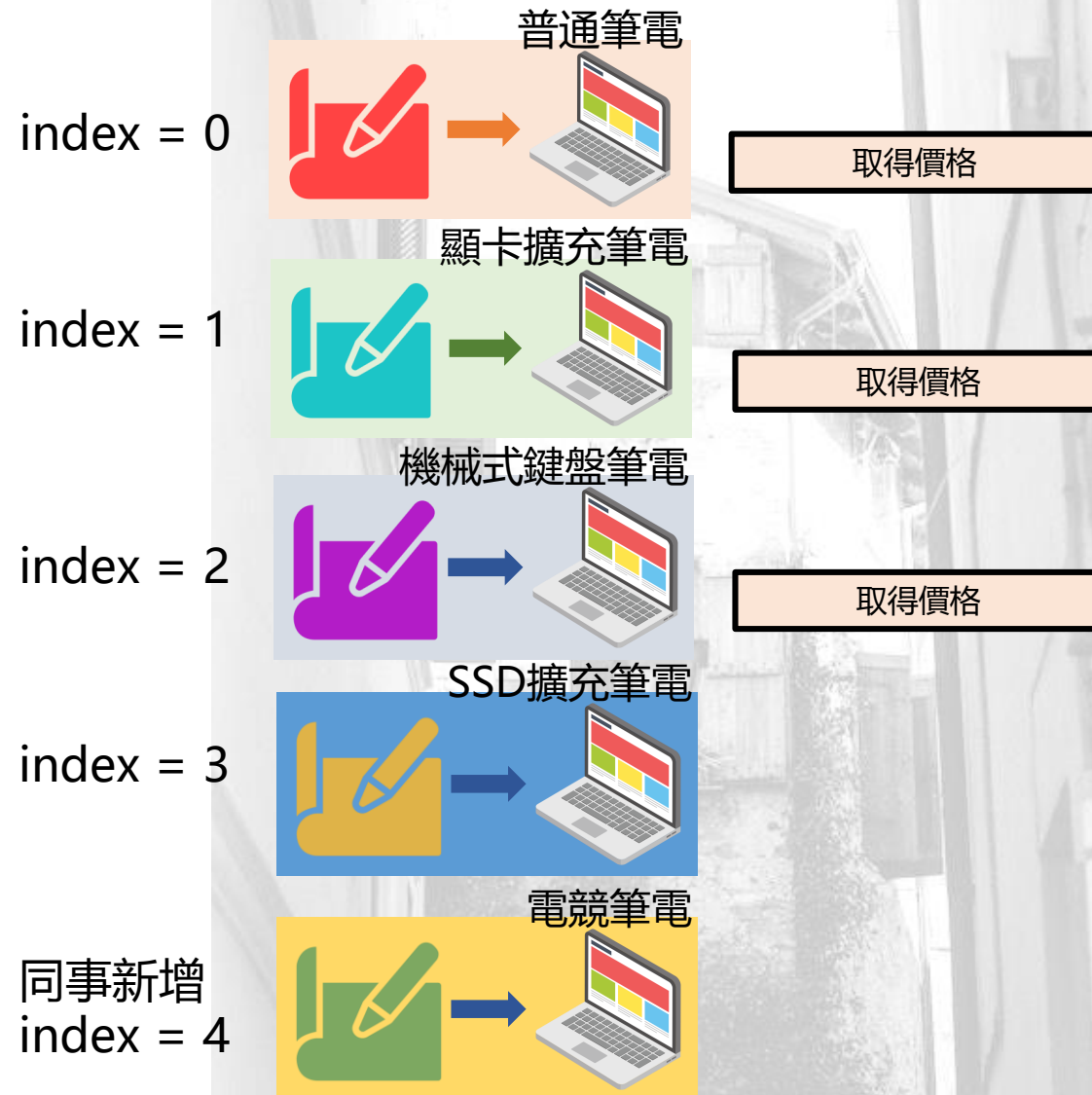
elseif 機械鍵盤{
    ...
}

elseif SSD筆電{
    ...
}

else{
    ...
}
```

# 多型 (5/6)

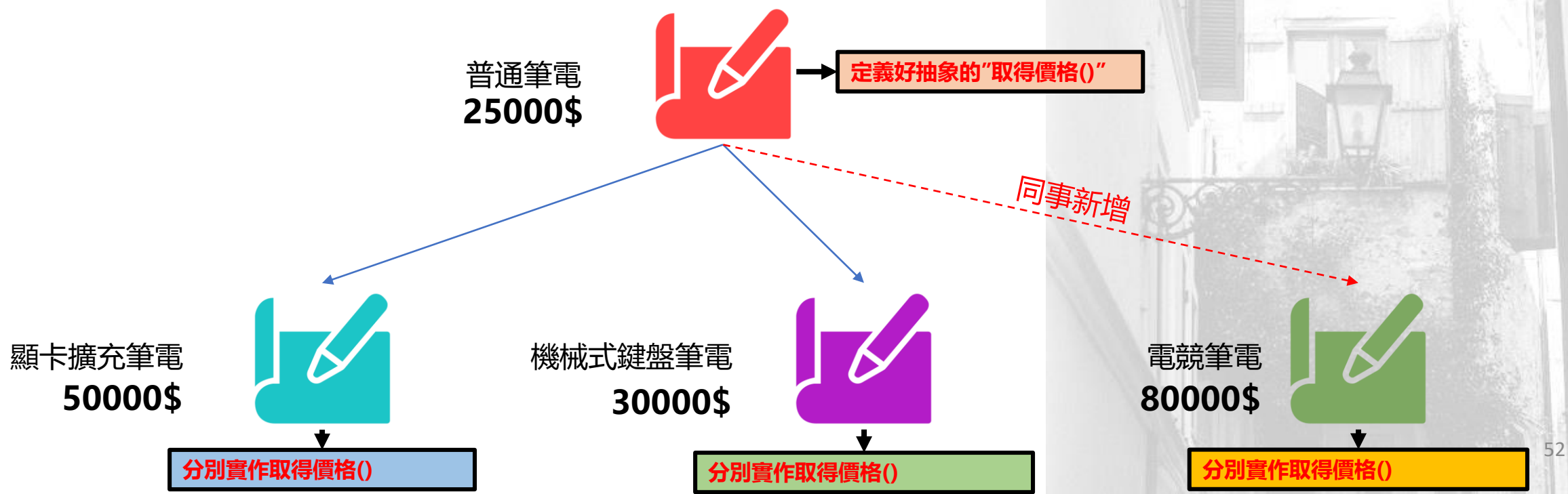
- 如果沒有使用多型，會遇到什麼問題？
  - 半年後，你同事要加電競筆電取價功能
    - 有可能會遇到前述問題
    - 假設你又將很多東西封裝起來了...





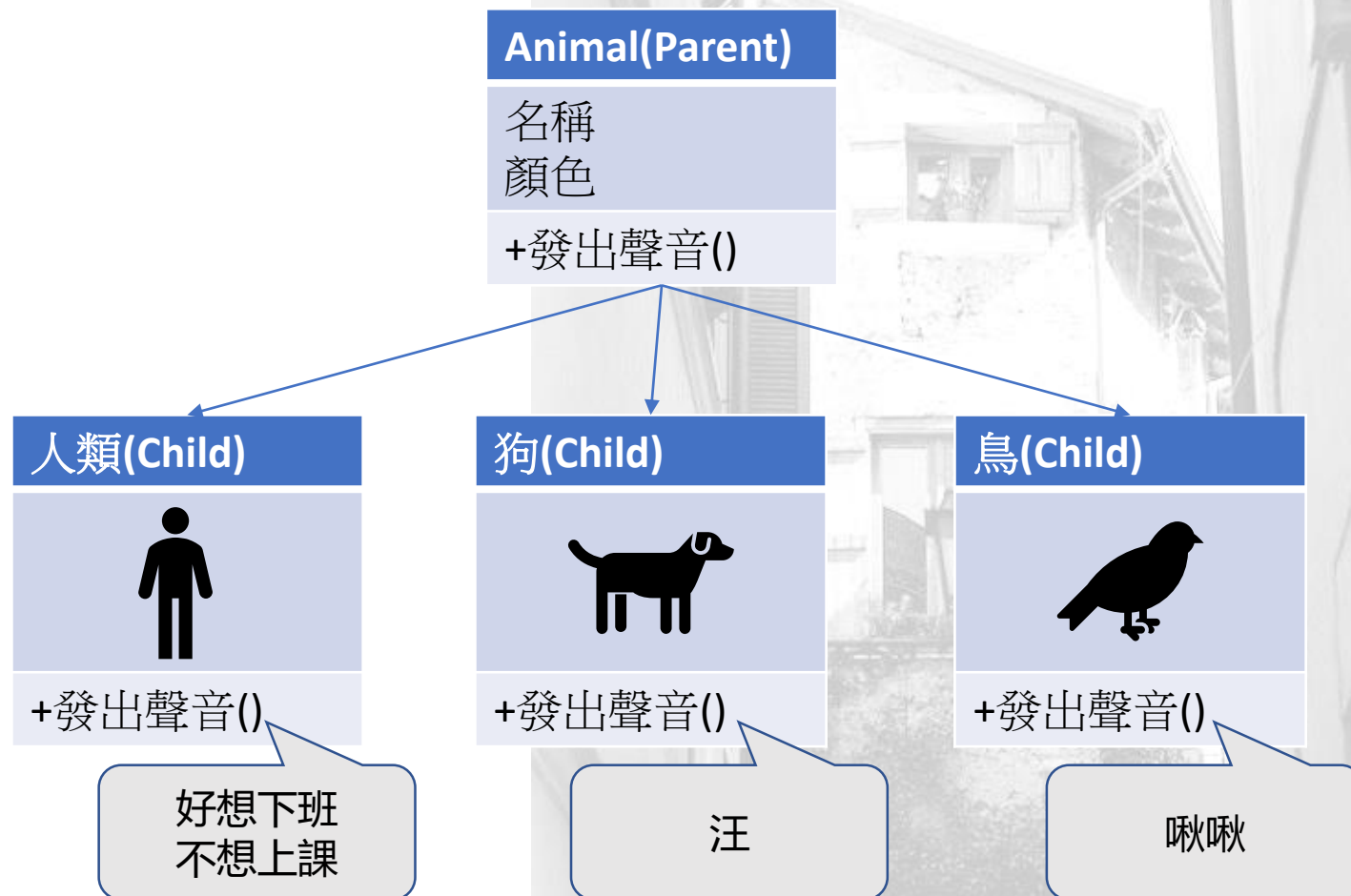
## 多型 (6/6)

- 使用多型之後，我們可以統一規範紅色圖紙（普通電腦）具有價格的方法，並讓其他兩台電腦繼承這個價格方法
- 我們能夠統一化價格方法的名稱，並且由於每台電腦都是紅色圖紙為底產出的電腦，可以使用共同取得價格的方法
- 雖然是不同圖紙生成出來的物件，我們允許能用相同名稱來定義方法，但在執行時會有不同的結果



# 多型 (補充)

- 允許能用相同名稱來定義/使用方法
- 當使用發出聲音()
  - 不用管發出聲音的實現方式
  - 根據不同Class來執行不同的發出聲音



# 組合 ( Composition )



# 組合 (1/6)

- 假設我們有兩台不同的電腦，他們各自有螢幕的瓦數、螢幕的尺寸、電池的毫安培數、與電池的型號



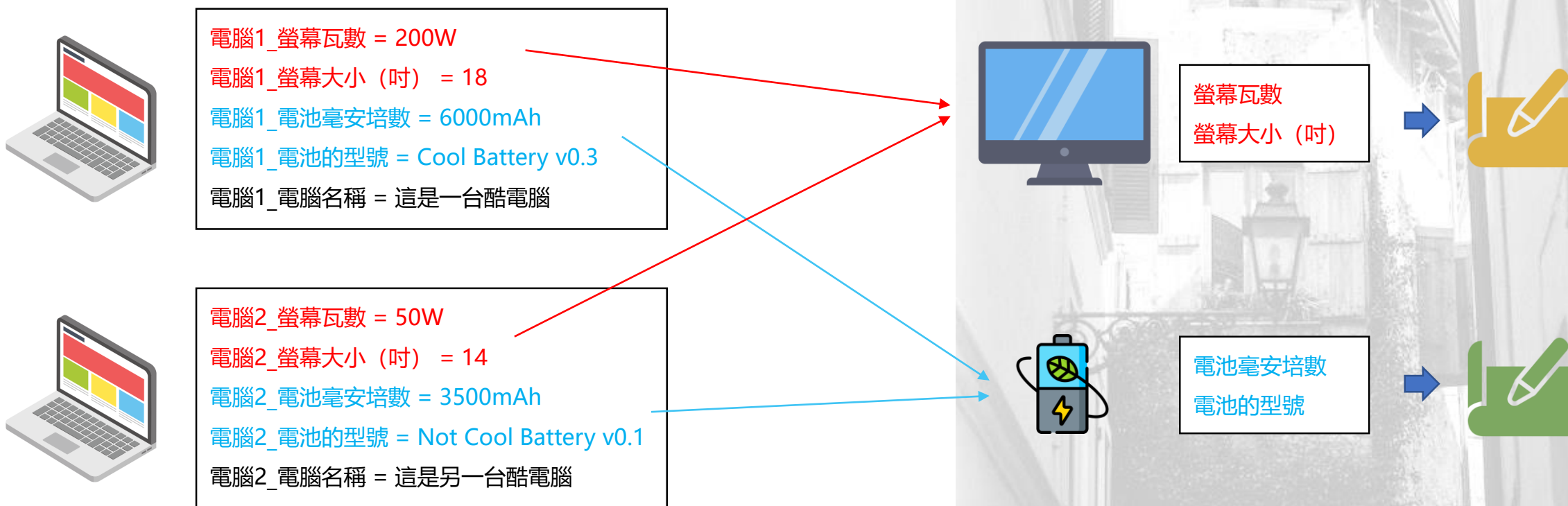
電腦1\_螢幕瓦數 = 200W  
電腦1\_螢幕大小 (吋) = 18  
電腦1\_電池毫安培數 = 6000mAh  
電腦1\_電池的型號 = Cool Battery v0.3  
電腦1\_電腦名稱 = 這是一台酷電腦



電腦2\_螢幕瓦數 = 50W  
電腦2\_螢幕大小 (吋) = 14  
電腦2\_電池毫安培數 = 3500mAh  
電腦2\_電池的型號 = Not Cool Battery v0.1  
電腦2\_電腦名稱 = 這是另一台酷電腦

## 組合 (2/6)

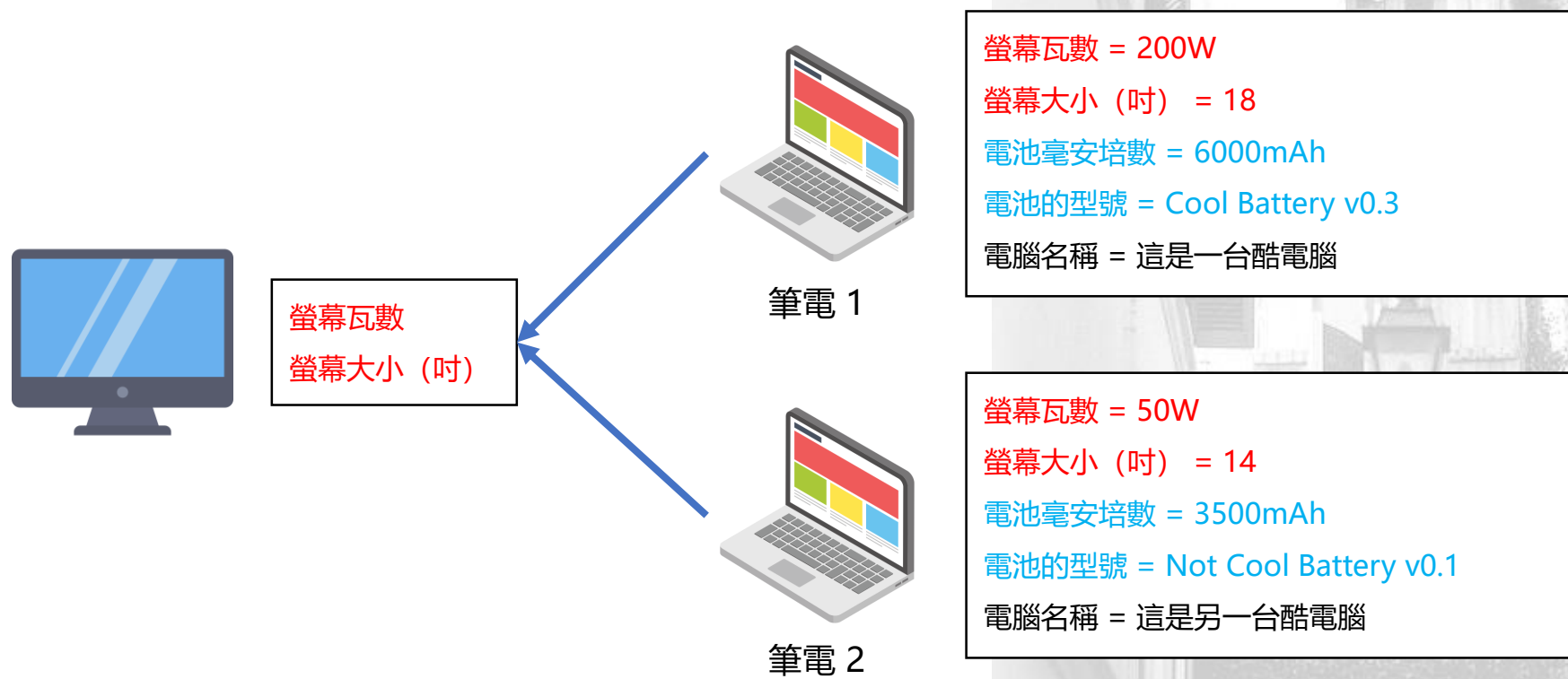
- 我們發現到了他們有共同的成員，所以考慮將他們抽出來當作繼承





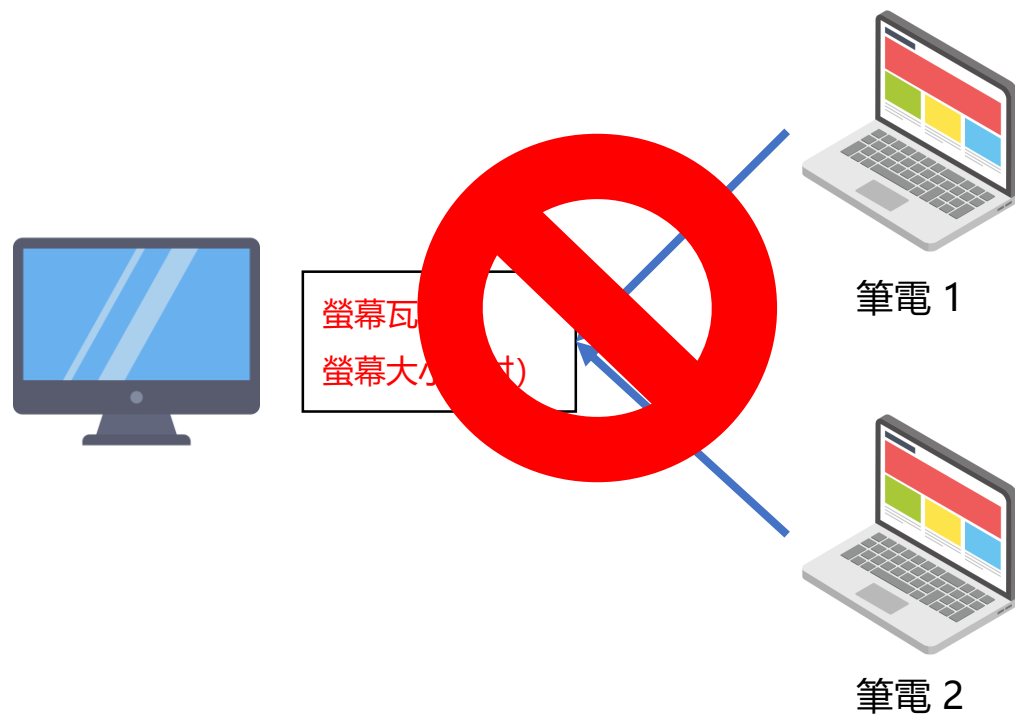
## 組合 (3/6)

- 於是你將筆電繼承了螢幕



## 組合 (4/6)

- 筆電不是螢幕，所以筆電繼承螢幕是件很奇怪的事情，不應該被發生



螢幕瓦數 = 200W

螢幕大小 (吋) = 18

電池毫安培數 = 6000mAh

電池的型號 = Cool Battery v0.3

電腦名稱 = 這是一台酷電腦

螢幕瓦數 = 50W

螢幕大小 (吋) = 14

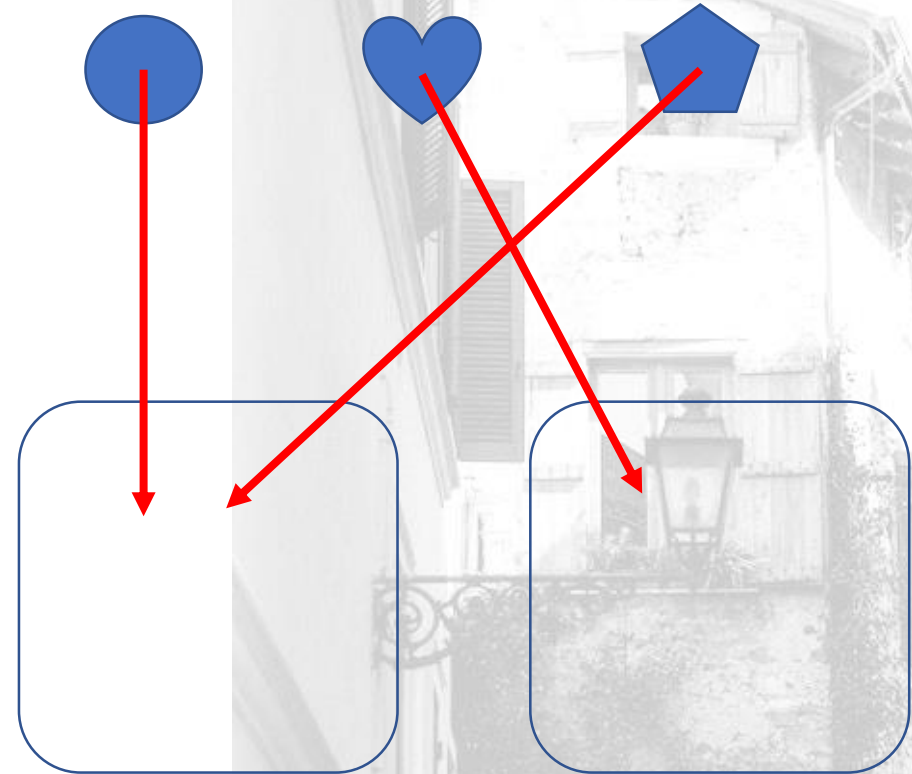
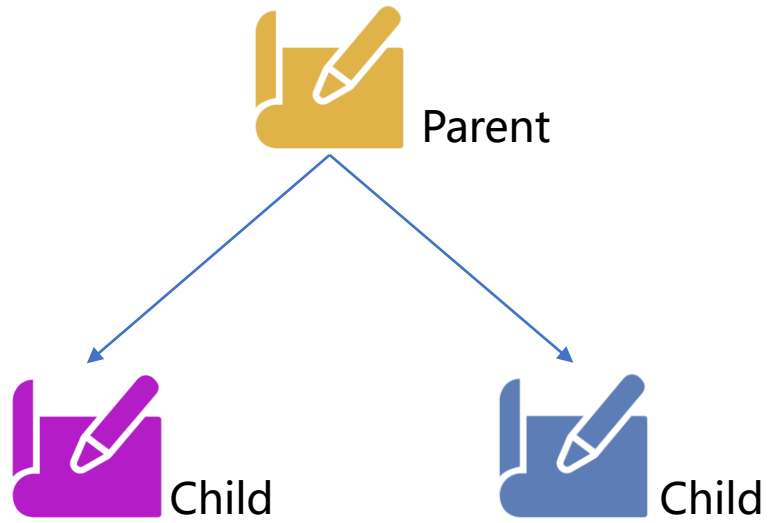
電池毫安培數 = 3500mAh

電池的型號 = Not Cool Battery v0.1

電腦名稱 = 這是另一台酷電腦

# 組合 (5/6)

- 繼承 V.S. 組合



# 組合 (6/6)

- 所以我們可以把螢幕與電池當成筆電的成員，並在筆電上初始化螢幕與電池



電腦名稱 = 這是一台酷電腦



電池毫安培數 = 6000mAh  
電池的型號 = Cool Battery v0.3



螢幕瓦數 = 200W  
螢幕大小 (吋) = 18



電腦名稱 = 這是另一台酷電腦



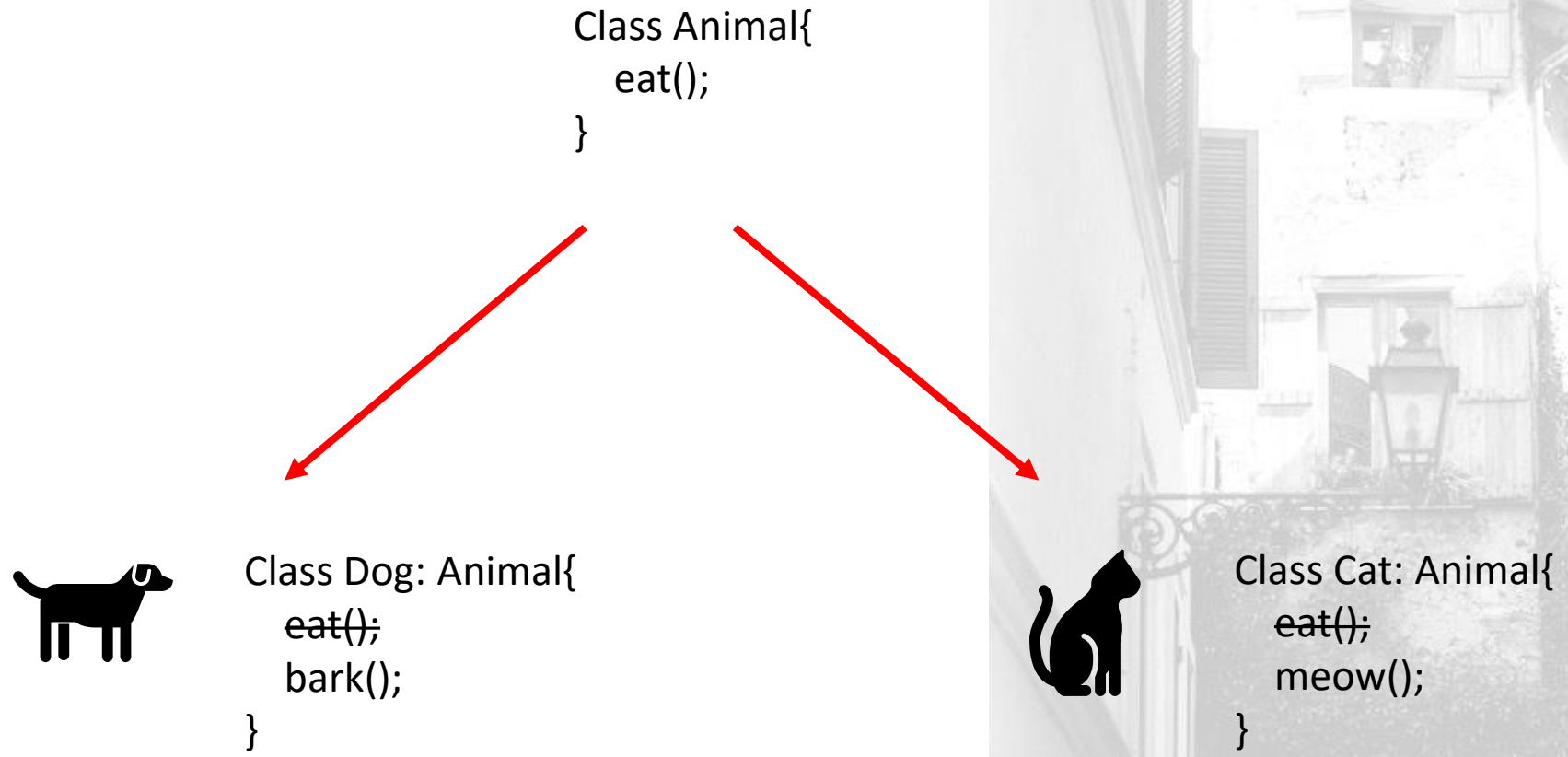
電池毫安培數 = 3600mAh  
電池的型號 = Not Cool Battery v0.1



螢幕瓦數 = 60W  
螢幕大小 (吋) = 14

# 組合 (補充)

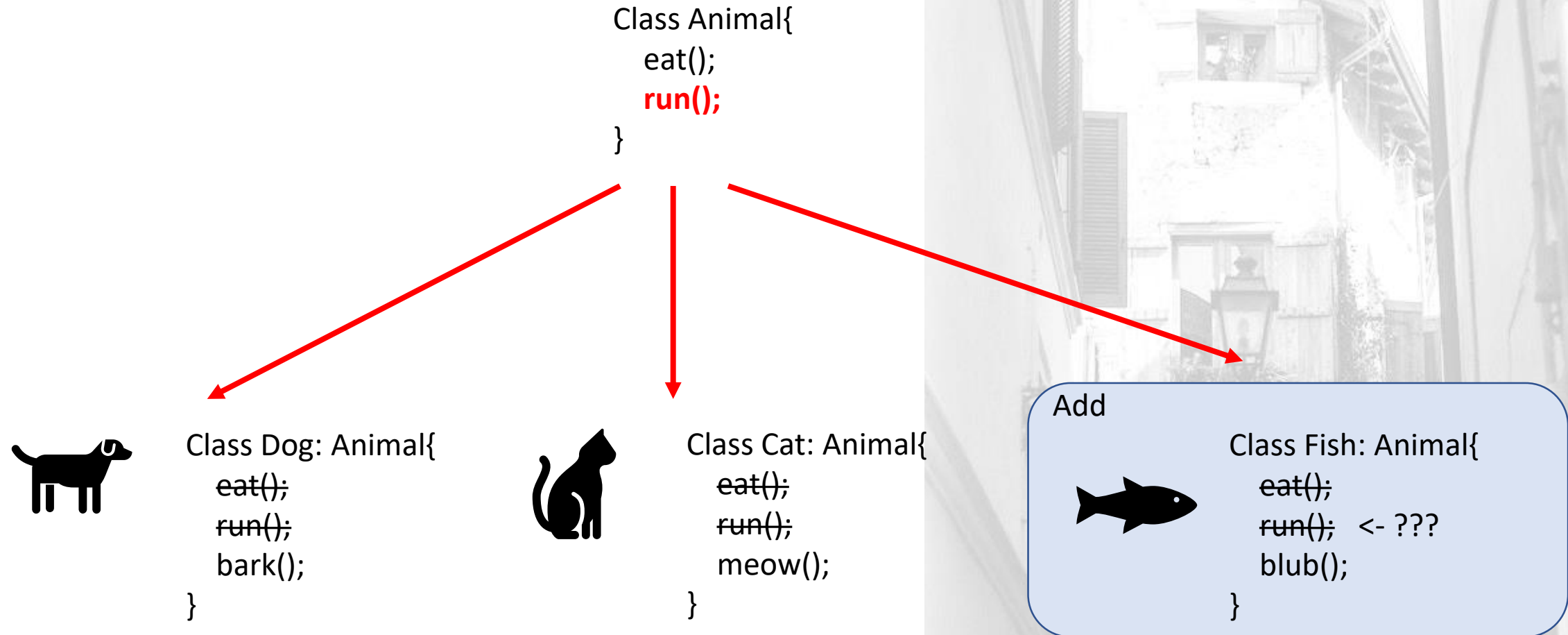
- 繼承





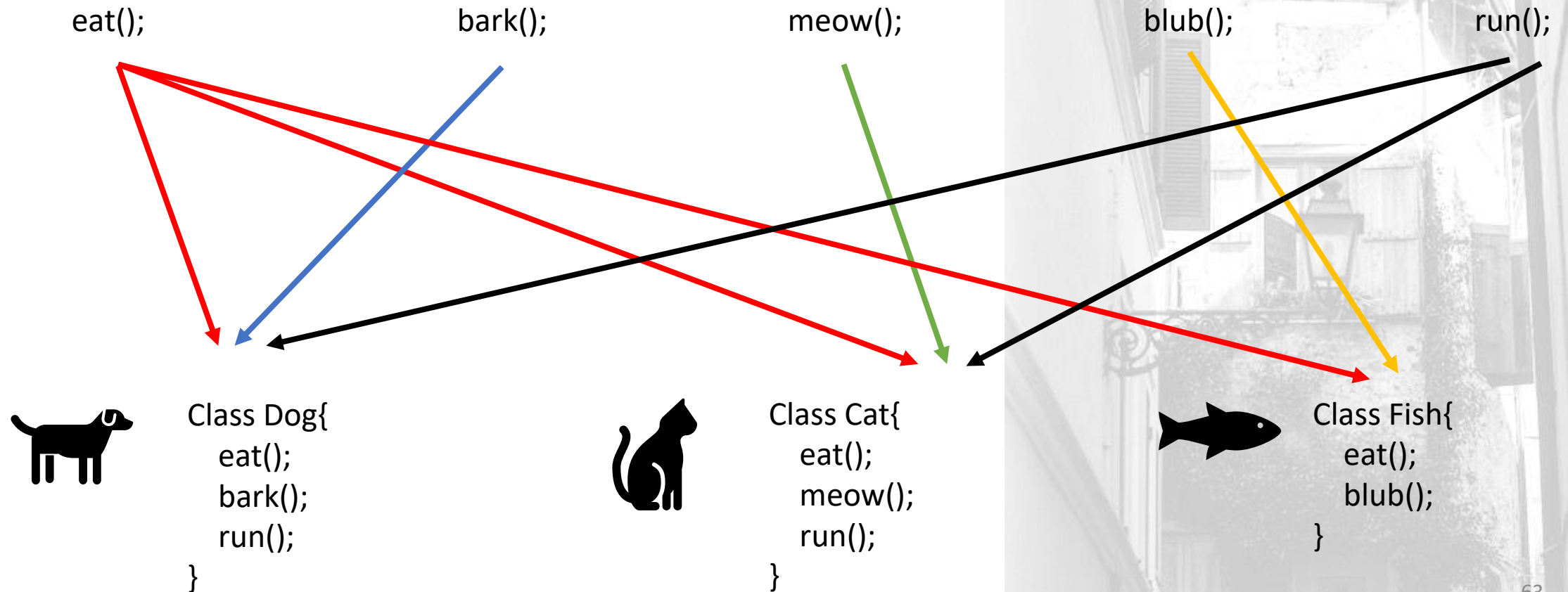
# 組合 (補充)

- 繼承的問題



# 組合 (補充)

- 組合



# 依賴注入 (Dependency Injection)

📄 ("這是我的酷電腦", "16吋", true, false, "Doors 11", "Letni i11-48763")

=

這是我的酷電腦  
16 吋大螢幕  
有數字鍵  
沒有 OLED  
Doors 11  
Letni i11-48763



# 依賴注入 (1/3)

- 回到前一個例子，我們要怎麼樣說明電池規格與螢幕規格呢？



電腦名稱 = 這是一台酷電腦



電池毫安培數 = ? mAh

電池的型號 = ?



螢幕瓦數 = ?

螢幕大小 (吋) = ?



電腦名稱 = 這是另一台酷電腦



電池毫安培數 = ? mAh

電池的型號 = ?



螢幕瓦數 = ?

螢幕大小 (吋) = ?

# 依賴注入 (2/3)

- 平常方法的話，我們應該會把所有的數值包成一包丟進去

("這是一台酷電腦", 6000, "Cool Battery v0.3", 200, 18)



電腦名稱 = 這是一台酷電腦



電池毫安培數 = ? mAh  
電池的型號 = ?



螢幕瓦數 = ?  
螢幕大小 (吋) = ?

("這是另一台酷電腦", 3600, "Not Cool Battery v0.1", 60, 14)



電腦名稱 = 這是另一台酷電腦



電池毫安培數 = ?  
電池的型號 = ?



螢幕瓦數 = ?  
螢幕大小 (吋) = ?



# 依賴注入 (3/3)

- 如果我們使用依賴注入的話，我們丟進去的東西可以不用是純數值或字串，而是物件。

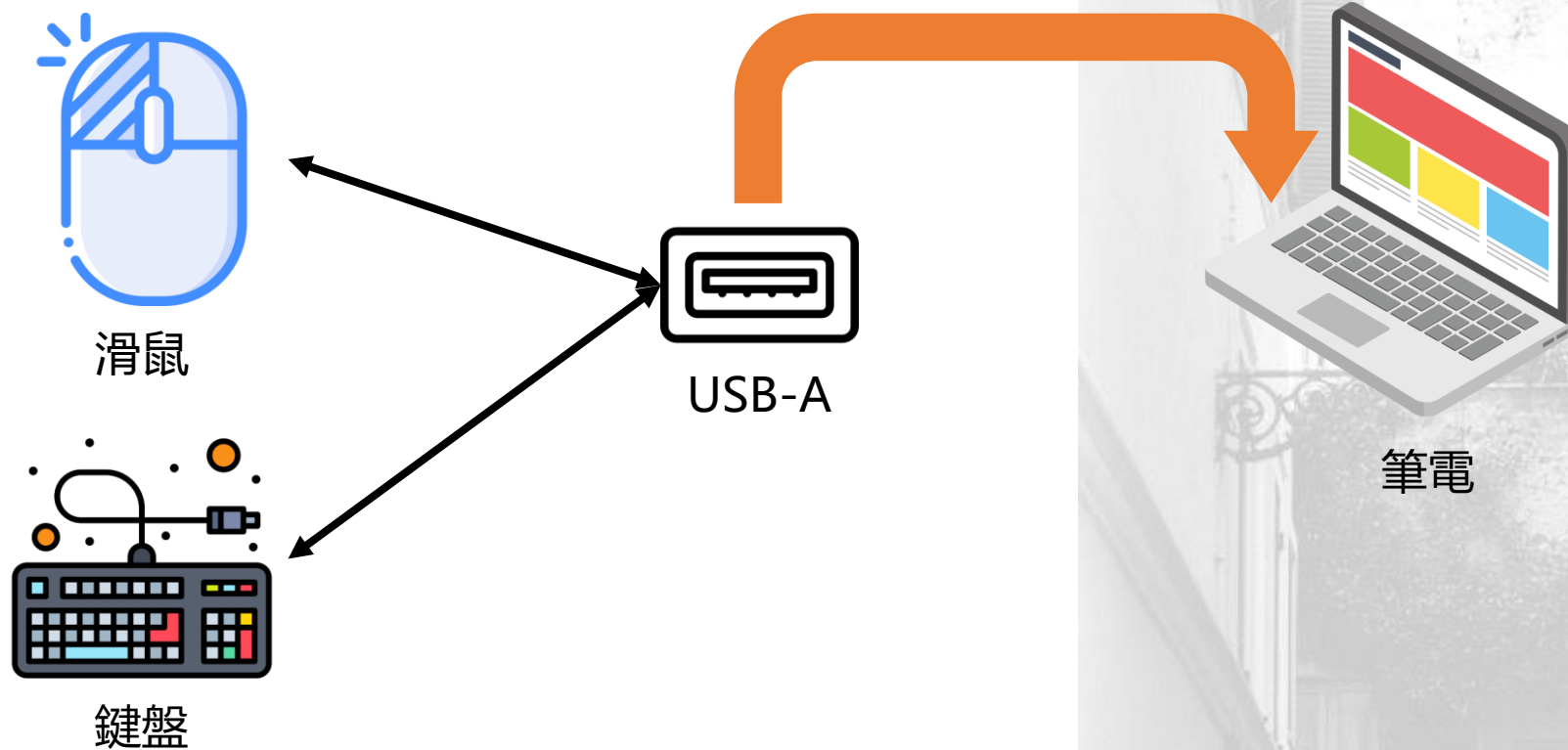


# 介面 ( Interface )



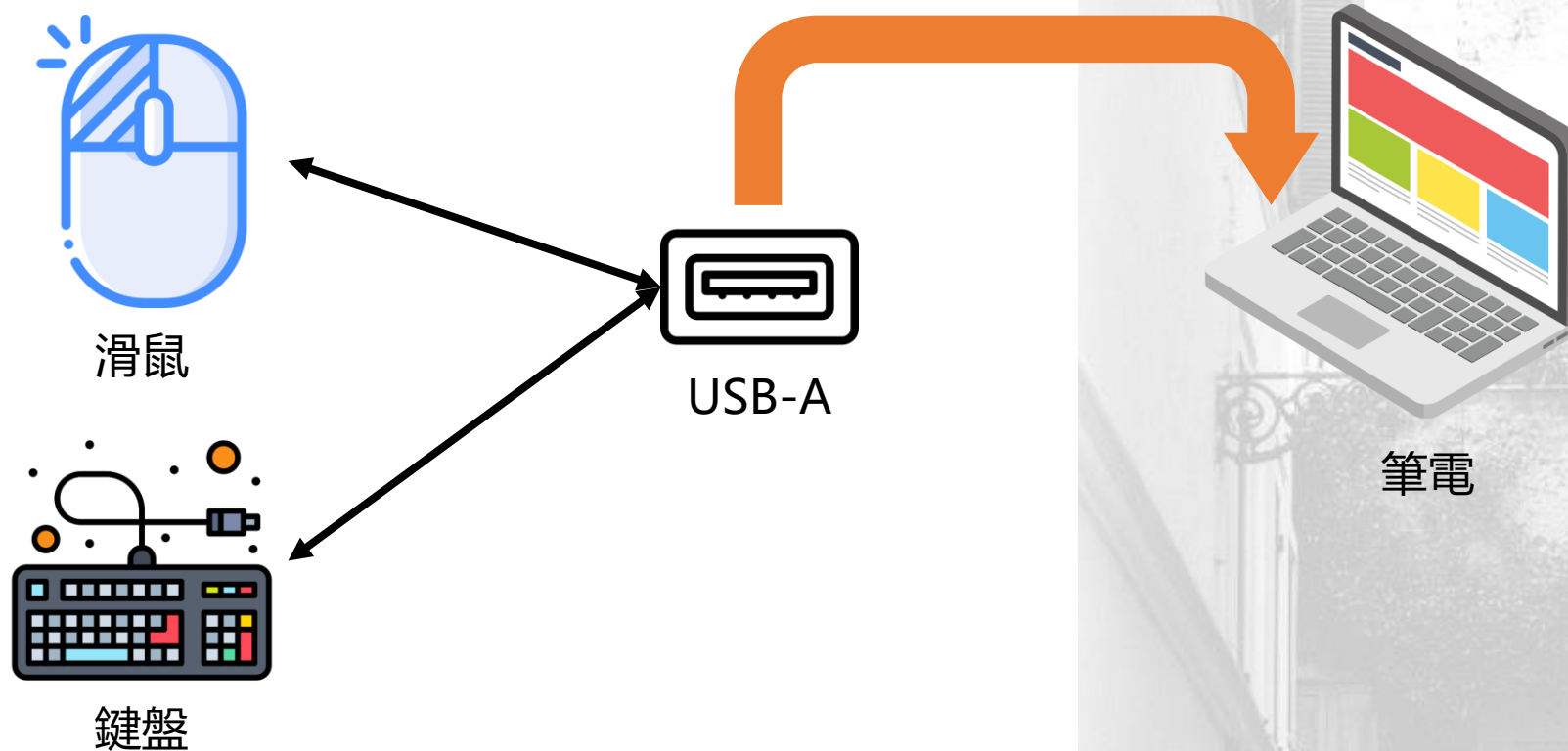
# 介面 (1/6)

- 試想一下，你電腦的 USB 埠。



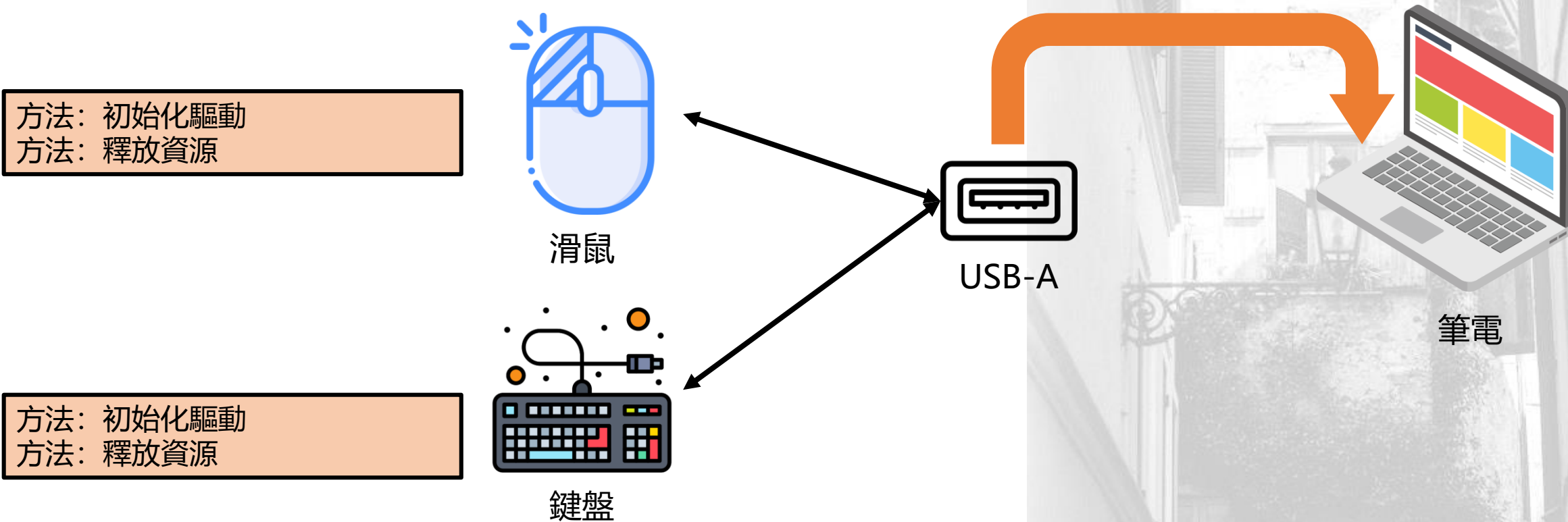
## 介面 (2/6)

- 當滑鼠與鍵盤插入 USB 埠時，設備會被初始化驅動並能夠被使用，並且在拔出時釋放資源
- 我們可以假設 USB-A 有「初始化驅動」與「拔出時釋放資源」的動作（方法）



## 介面 (3/6)

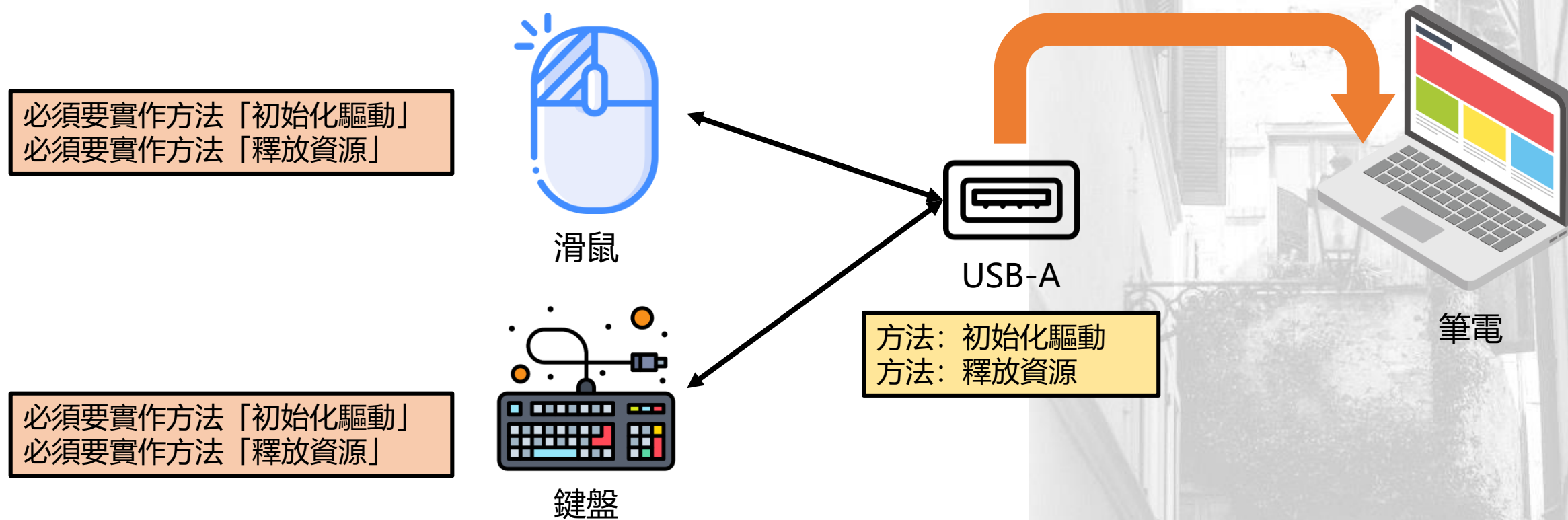
- 所以，滑鼠與鍵盤就會**被規定**需要有「初始化驅動」與「釋放資源」的方法





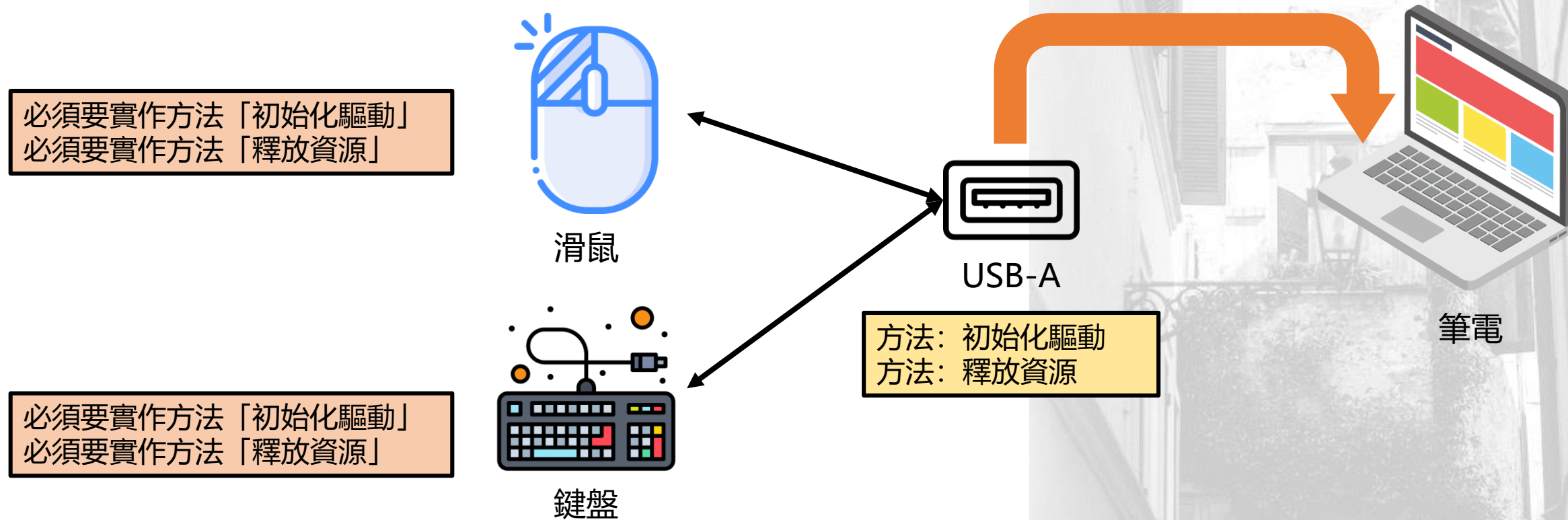
## 介面 (4/6)

- 因此，我們可以用介面來規範每個接入 USB-A 的設備必須要「初始化驅動」與「釋放資源」。



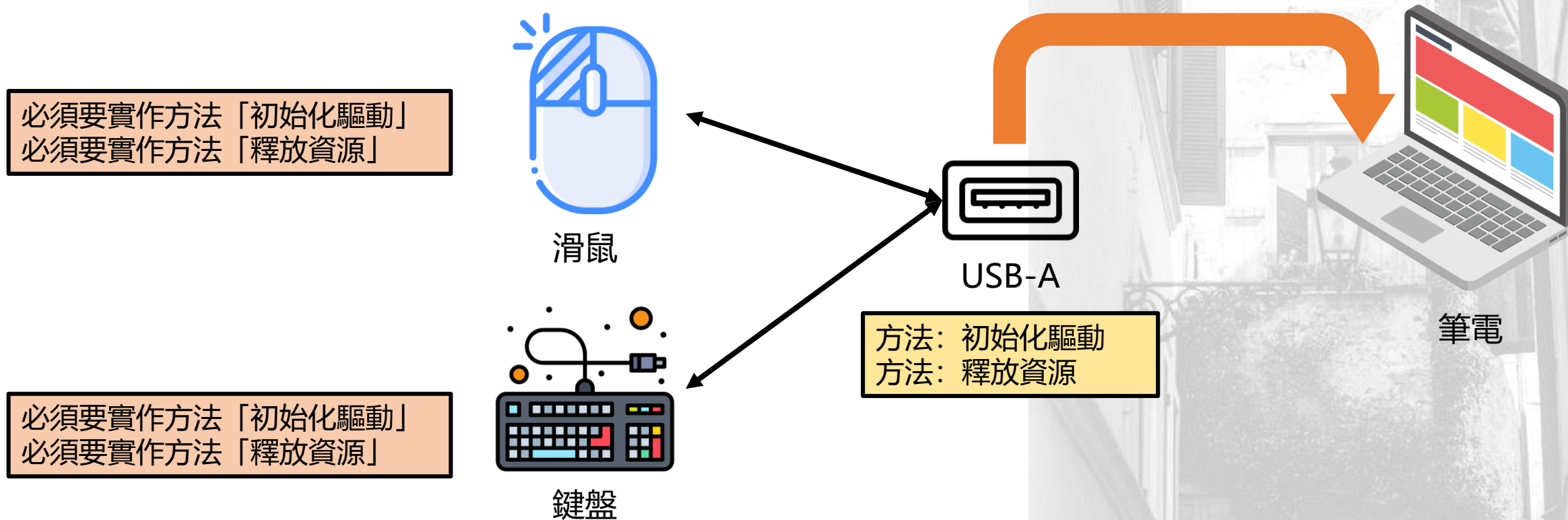
## 介面 (5/6)

- 當我們只需要規範方法而不在乎方法內的實作與成員時，我們應該使用介面來規範方法。



## 介面 (6/6)

- 為什麼不用繼承?
- 因為滑鼠與鍵盤所需要實作這兩個方法的成員可能是完全不一樣的，因此 USB-A 有成員這件事情是不合理的。



**Thanks!**