物件

物件(Object)類型是電腦程式的一種資料類型，用抽象化概念來比喻為人類現實世界中的物體。

在JavaScript中，除了原始的資料類型例如數字、字串、布林等等之外，所有的資料類型都是物件。不過，JavaScript的物件與其他目前流行的物件導向程式語言的設計有明顯的不同，它一開始是使用原型基礎(prototype-based)的設計，而其他的物件導向程式語言，大部份都是使用類別基礎(class-based)的設計。

在ES6之後加入了類別為基礎的語法(是原型基礎的語法糖)，JavaScript仍然是原型基礎，但可以用類別語法建立物件與繼承之用，雖然目前來說，仍然是很基本的類別語法，但讓開發者多了另一種選擇。

物件在JavaScript語言中可分為兩種應用層面來看:

- 主要用於資料的描述，它扮演類似關連陣列的資料結構，儲存"鍵-值"的成對資料。很常見到用陣列中包含物件資料來代表複數的資料集合。
- 主要用於物件導向的程式設計，可以設計出各種的物件，其中包含各種方法，就像已經介紹過的各種包裝物件，例如字串、陣列等等的包裝物件。

物件類型使用以屬性與方法為主要組成部份，這兩種合稱為物件成員(member):

- 屬性: 物件的基本可被描述的量化資料。例如水果這個物件，有顏色、產地、大小、重量、甜度等等屬性。
- 方法: 物件的可被反應的動作或行為。例如車子這個物件，它的行為有加速、煞車、轉彎、打方向燈等等的行為或可作的動作。

物件定義方式

物件字面(Object Literals)

用於資料描述的物件定義，使用花括號(curly braces)({})作為區塊宣告，其中加入無關順序的"鍵-值"成對值，屬性的值可以是任何合法的值，可以包含陣列、函式或其他物件。

而在物件定義中的"鍵-值"，如果是一般的值的情況，稱為"屬性(property, prop)"，如果是一個函式，稱之為"方法(method)"。屬性與方法我們通常合稱為物件中的成員(member)。

> 註：屬性名稱(鍵)中也不要使用保留字，請使用合法的變數名稱

```
const emptyObject = {}

const player = {
    fullName: 'Inori',
    age: 16,
    gender: 'girl',
    hairColor: 'pink'
}
```

以如果你已經對陣列有一些理解的基礎下，物件的情況相當類似，首先在定義與獲取值上:

```
const aArray = []
const aObject = {}

const bArray = ['foo', 'bar']
const bObject = {
    firstKey: 'foo',
    secondKey: 'bar'
}

bArray[2] = 'yes'
bObject.thirdKey = 'yes'

console.log(bArray[2]) //yes
console.log(bObject.thirdKey) //yes
```

不過，對於陣列的有順序索引值，而且只有索引值的情況，我們會更加關心物件中"鍵"的存在，物件中的成員(屬性與方法)，都是使用物件加上點(.)符號來存取。上面的程式碼雖然在 `thirdKey` 不存在時，會自動進行擴充，但通常物件的定義是在使用前就會定義好的，總是要處於可預測情況下是比較好的作法。物件的擴充是經常使用在對現有的JavaScript語言內建物件，或是函式庫的擴充之用。

> 心得口訣: 對於初學者要記憶是用花括號({})來定義物件，而方括號([])來定義陣列，可以用口訣來快速記憶: 物(霧)裡看花。方陣快炮。

> 註: 存取物件中的成員(屬性或方法)，使用的是句點(.)符號，這已經在書中的很多內建方法的使用時都有用到，相信你應該不陌生。

> 註: 相較於陣列中不建議使用的 `new Array()` 語法，也有 `new Object()` 的語法，也是不需要使用它。

> 註: 物件內的成員(方法與屬性)的存取，的確也可以使用像 `obj[prop]` 的語法，有一些情況下會這樣使用，例如在成員(方法與屬性)還未確定的函式裡面使用，一般情況下為避免與陣列的成員存取語法混淆，所以很少用。以下範例來自這裡:

```
const luke = {
  jedi: true,
  age: 28,
}

function getProp(prop) {
  return luke[prop]
}

const isJedi = getProp('jedi');
```

上面這種定義物件的字面文字方式，這是一種單例(singleton)的物件，也就是在程式碼中只能有唯一一個物件實體，就是你定義的這個物件。當你需要產生同樣內容的多個物件時，那又該怎麼作？那就是要用另一種定義方式了。

物件字面定義方式，通常單純只用於物件類型的資料描述，也就是只用於定義單純的"鍵-值"對應的資料，在裡面不會定義函式(方法)。而基於物件字面定義，發展出JSON(JavaScript Object Notation)的資料定義格式，這是現今在網路上作為資料交換使用的一種常見的格式，在特性篇會再對JSON格式作更多的說明。

類別(Class)

類別(Class)是先裡面定義好物件的整體結構藍圖(blue print)，然後再用這個類別定義，來產生相同結構的多個的物件實例，類別在定義時並不會直接產生出物件，要經過實體化的過程(`new` 運算符)，才會產生真正的物件實體。另外，目前因為類別定義方式還是個很新的語法，在實作時除了比較新的函式庫或框架，才會開始用它來撰寫。以下的為一個簡單範例:

> 註: 在ES6標準時，現在的JavaScript中的物件導向特性，並不是真的是以類別為基礎(class-based)的，這是骨子裡還是以原型為基礎(prototype-based)的物件導向特性語法糖。

```
class Player {
    constructor(fullName, age, gender, hairColor) {
        this.fullName = fullName
        this.age = age
        this.gender = gender
        this.hairColor = hairColor
    }

    toString() {
        return 'Name: '+this.fullName+', Age:'+this.age
    }
}

const inori = new Player('Inori', 16, 'girl', 'pink')
console.log(inori.toString())
console.log(inori.fullName)

const tsugumi = new Player('Tsugumi', 14, 'girl', 'purple')
console.log(tsugumi.toString())
```

> 註: 注意類別名稱命名時要使用大駝峰(ClassName)的寫法

下面分別說明一些這個例子中用到的語法與關鍵字的重要概念，以及類別延伸的一些語法。

this

在這個物件的類別定義中，我們第一次真正見到 this 關鍵字的用法， this 簡單的說來，是物件實體專屬的指向變數， this 指向的就是"這個物件實體"，以上面的例子來說，也就是當物件真正實體化時， this 變數會指向這個物件實體。 this 是怎麼知道要指到哪一個物件實體？是因為 new 運算符造成的結果。

this 變數是JavaScript的一個特性，它是隱藏的內部變數之一，當函式呼叫或物件實體化時，都會以這個 this 變數的指向對象，作為執行期間的依據。

還記得我們在函式的章節中，使用作用範圍(Scope)來說明以函式為基礎的檢視角度，在函式區塊中可見的變數與函式的領域的概念。而JavaScript中，另外也有一種上下文環境(Context)的概念，就是對於 this 的在執行期間所依據的影響，即是以物件為基礎的的檢視角度。

this 也就是執行上下文可以簡單用三個情況來區分:

1. 函式呼叫: 在一般情況下的函式呼叫， this 通常都指向global物件。這也是預設情況。
2. 建構式(constructor)呼叫: 透過 new 運算符建立物件實體，等於呼叫類型的建構式， this 會指向新建立的物件實例
3. 物件中的方法呼叫: this 指向呼叫這個方法的物件實體

所以當建構式呼叫時，也就是使用 new 運算符建立物件時， this 會指向新建立的物件，也就是下面這段程式碼:

```
const inori = new Player('Inori', 16, 'girl', 'pink')
```

因此在建構式中的指定值的語句，裡面的 this 值就會指向是這個新建立的物件，也就是 inori :

```
constructor(fullName, age, gender, hairColor) {
        this.fullName = fullName
        this.age = age
        this.gender = gender
        this.hairColor = hairColor
    }
```

也就是說在建立物件後，經建構式的執行語句，這個 inori 物件中的屬性值就會被指定完成，所以可以用像下面的語法來存取屬性:

```
inori.fullName
inori.age
inori.gender
inori.hairColor
```

第3種情況是呼叫物件中的方法，也就是像下面的程式碼中， this 會指向這個呼叫toString方法的物件，也就是 inori :

```
inori.toString()
```

對於 this 的說明大致上就是這樣而已，這裡都是很直覺的說明。 this 還有一部份的細節與應用情況，在特性篇中有獨立的一個章節來說明 this 的一些特性與應用情況， this 的概念在JavaScript中十分重要，初學者真的需要多花點時間才能真正搞懂。

建構式(constructor)

建構式是特別的物件方法，它必會在物件建立時被呼叫一次，通常用於建構新物件中的屬性，以及呼叫上層父母類別(如果有繼承的話)之用。用類別(class)的定義時，物件的屬性都只能在建構式中定義，這與用物件字面的定義方式不同，這一點是要特別注意的。如果物件在初始化時不需要任何語句，那麼就不要寫出這個建構式，實際上類別有預設的建構式，它會自動作建構的工作。

關於建構式或物件方法的多形(polymorphism)或覆蓋(Overriding)，在JavaScript中**沒有**這種特性。建構式是會被限制只能有一個，而在物件中的方法(函式)也沒這個特性，定義同名稱的方法(函式)只會有一個定義被使用。所以如果你需要定義不同的建構式在物件中，因應不同的物件實體的情況，只能用函式的不定傳入參數方式，或是加上傳入參數的預設值來想辦法改寫，請參考函式內容中的說明。以下為一個範例:

```
class Option {
    constructor(key, value, autoLoad = false) {
        if (typeof key != 'undefined') {
            this[key] = value
        }
        this.autoLoad = autoLoad
    }
}
```

```
const op1 = new Option('color', 'red')
const op2 = new Option('color', 'blue', true)
```

私有成員

JavaScript截至ES6標準為止，在類別中並沒有像其他程式語言中的私有的(private)、保護的(protected)、公開的(public)這種成員存取控制的修飾關鍵字詞，基本上所有的類別中的成員都是公開的。雖然也有其他"模擬"出私有成員的方式，不過它們都是複雜的語法，這裡就不說明了。

目前比較簡單常見的區分方式，就是在私有成員(或方法)的名稱前面，加上下底線符號(_)前綴字，用於區分這是私有的(private)成員，這只是由程式開發者撰寫上的區分差別，與語言本身特性無關，對JavaScript語言來說，成員名稱前有沒有有下底線符號(_)的，都是視為一樣的變數。以下為簡單範例:

```
class Student {
    constructor(id, firstName, lastName) {
        this._id = id
        this._firstName = firstName
        this._lastName = lastName
    }

    toString() {
        return 'id is '+this._id+' his/her name is '+this.firstName+' '+this.lastName
    }
}
```

> 註: 如果是私有成員，就不能直接在外部存取，要用getter與setter來實作取得與修改值的方法。私有方法也不能在外部呼叫，只能在類別內部使用。

getter與setter

在類別定義中可以使用 `get` 與 `set` 關鍵字，作為類別方法的修飾字，可以代表getter(取得方法)與setter(設定方法)。一般的公開的原始資料類型的屬性值(字串、數字等等)，不需要這兩種方法，原本就可以直接取得或設定。只有私有屬性或特殊值，才需要用這兩種方法來作取得或設定。getter(取得方法)與setter(設定方法)的呼叫語法，長得像一般的存取物件成員的語法，都是用句號(.)呼叫，而且setter(設定方法)是用指定值的語法，不是傳入參數的那種語法。以下為範例:

```
class Option {
    constructor(key, value, autoLoad = false) {
        if (typeof key != 'undefined') {
            this['_' + key] = value;
        }
        this.autoLoad = autoLoad;
    }

    get color() {
      if (this._color !== undefined) {
        return this._color
      } else {
        return 'no color prop'
      }
    }

    set color(value) {
      this._color = value
    }
}

const op1 = new Option('color', 'red')
op1.color = 'yellow'

const op2 = new Option('action', 'run')
op2.color = 'yellow' //no color prop
```

> 註: 所以getter不會有傳入參數，setter只會有一個傳入參數。

靜態成員

靜態(Static)成員指的是屬於類別的屬性或方法，也就是不論是哪一個被實體化的物件，都共享這個方法或屬性。而且，實際上靜態(Static)成員根本不需要實體化的物件來呼叫或存取，直接用類別就可以呼叫或存取。JavaScript中只有靜態方法，沒有靜態屬性，使用的是 `static` 作為方法的修飾字詞。以下為一個範例:

```
class Student {
    constructor(id, firstName, lastName) {
        this.id = id
        this.firstName = firstName
        this.lastName = lastName

        //這裡呼叫靜態方法，每次建構出一個學生實體就執行一次
        Student._countStudent()
    }

    //靜態方法的定義
    static _countStudent(){
      if(this._numOfStudents === undefined) {
          this._numOfStudents = 1
      } else {
          this._numOfStudents++
      }
    }

    //用getter與靜態方法取出目前的學生數量
    static get numOfStudents(){
      return this._numOfStudents
    }

}

const aStudent = new Student(11, 'Eddy', 'Chang')
console.log(Student.numOfStudents)

const bStudent = new Student(22, 'Ed', 'Lu')
console.log(Student.numOfStudents)

const cStudent = new Student(33, 'Horward', 'Liu')
console.log(Student.numOfStudents)
```

靜態屬性目前來說有兩種解決方案，一種是使用ES7的Class Properties標準，可以使用 `static` 關鍵字來定義靜態屬性，另一種是定義到類別原本的定義外面:

```
// ES7語法方式
class Video extends React.Component {
  static defaultProps = {
    autoPlay: false,
    maxLoops: 10,
  }
  render() { ... }
}



// ES6語法方式
class Video extends React.Component {
  constructor(props) { ... }
  render() { ... }
}

Video.defaultProps = { ... }
```

> 註: ES7的靜態(或類別)屬性的轉換，要使用bebal的stage-0 perset。

繼承

用extends關鍵字可以作類別的繼承，而在建構式中會多呼叫一個 `super()` 方法，用於執行上層父母類別的建構式之用。 `super` 也可以用於指向上層父母類別，呼叫其中的方法或存取屬性。

繼承時還有有幾個注意的事項:

- 繼承的子類別中的建構式， `super()` 需要放在第一行，這是標準的呼叫方式。
- 繼承的子類別中的屬性與方法，都會覆蓋掉原有的在父母類別中的同名稱屬性或方法，要區為不同的屬性或方法要用 `super` 關鍵字來存取父母類別中的屬性或方法

```
class Point {
    constructor(x, y) {
        this.x = x
        this.y = y
    }
    toString() {
        return '(' + this.x + ', ' + this.y + ')';
    }
}

class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y)
        this.color = color
    }
    toString() {
        return super.toString() + ' in ' + this.color
    }
}
```

物件相關方法

instanceof運算符

`instanceof` 運算符用於測試某個物件是否由給定的建構式所建立，聽起來可能會覺得有點怪異，這個字詞從字義上看起來應該是"測試某個物件是否由給定的**類別**所建立"，但要記得JavaScript中本身就沒有類別這東西，物件的實體化是由建構函式，組成原型鏈而形成的。以下為一個簡單的範例:

```
const eddy = new Student(11, 'Eddy', 'Chang')
console.log(eddy instanceof Student) //true
```

> 註: instanceof運算符並不是100%精確的，它有一個例外情況是在處理來自HTML的frame或iframe資料時會失效。

> 註: 關於原型鏈的說明請見特性篇的"原型物件導向"章節

物件的拷貝

在陣列的章節中，有談到淺拷貝(shallow copy)與深拷貝(deep copy)的概念，同樣在物件資料結構中，在拷貝時也同樣會有這個問題。陣列基本上也是一種特殊的物件資料結構，其實這個概念應該是由物件為主的發展出來的。詳細的內容就不多說，以下只針對淺拷貝的部份說明:

Object.assign()

> 推薦方式

`Object.assign()` 是ES6中的新方法，在ES6前都是用迴圈語句，或其他的方式來進行物件拷貝的工作。 `Object.assign()` 的用法很直覺，它除了拷貝之外，也可以作物件的合併，合併時成員有重覆名稱以愈後面的物件中的成員為主進行覆蓋:

```
//物件拷貝
const aObj = { a: 1, b: 'Test' }
const copy = Object.assign({}, aObj)
console.log(copy) // {a: 1, b: "Test"}

//物件合併
const bObj = { a: 2, c: 'boo' }
const newObj = Object.assign(aObj, bObj)
console.log(newObj) //{a: 2, b: "Test", c: "boo"}
```

> 註: `null` 或 `undefined` 在拷貝過程中會被無視。

> 註: 如果需要額外的擴充(Polyfill)可以參考Object.assign(MDN)，或是ES2015 Object.assign() ponyfill

JSON.parse加上JSON.stringify

> 不建議的方式

JSON是使用物件字面文字的定義方式，延伸用來專門定義資料格式的一種語法。它經常用來搭配AJAX技術，作為資料交換使用，也有很多NoSQL的資料庫更進一步用它改良後，當作資料庫裡的資料定義格式。

這個方式是把物件定義的字面文字字串化，然後又分析回去的一種語法，它對於物件中的方法(函式)直接無視，所以只能用於只有數字、字串、陣列與一般物件的物件定義字面:

```
const aObj = { a: 1, b: 'b', c: { p : 1 }, d: function() {console.log('d')} }

const aCopyObj = JSON.parse(JSON.stringify(aObj))
console.log(aCopyObj)

const bCopyObj = Object.assign({}, aObj)
console.log(bCopyObj)
```

這方式其實不推薦使用，為什麼會寫出來的原因，是你可能會看到有人在使用這個語法，會使用這個語法的主要原因以前沒有像 `Object.assign` 這麼簡單的語法。除此之外，你可能還可以找到各種物件拷貝的各種函式或教學文件。

物件的拷貝的使用原則與陣列拷貝的說明類似，要不就使用 `Object.assign` ，要不然就使用外部函式庫例如jQuery、underscore或lodash中拷貝的API。

物件屬性的"鍵"或"值"判斷

undefined判斷方式

直接存取物件中不存在的屬性，會直接回傳 `undefined` 時，這是最直接的判斷物件屬性是否存在的方式，也是最快的方式。不過它有一個缺點，就是當這個屬性本身就是 `undefined` 時，這個判斷方法就失效了，如果你本來要的值本來就絕對不是 `undefined` ，所以可以這樣用。

```
//判斷鍵是否存在
typeof obj.key !== 'undefined'

//判斷值是否存在
obj.key !== undefined
obj['key'] !== undefined
```

> 註: 這個語法也可以判斷某個方法是否存在於物件中。

in運算符 與 hasOwnProperty方法

> 推薦使用 hasOwnProperty方法

這兩個語法在正常情況下，都是可以正確回傳物件屬性的"鍵"是否存在的判斷:

```
obj.hasOwnProperty('key')
'key' in obj
```

它們還是有明顯的差異， `hasOwnProperty` 方法不會檢查物件的原型鏈(prototype chain，或稱之為原型繼承)，也就是說 `hasOwnProperty` 方法只會檢查這個物件中有的屬性鍵，用類別定義時的方法是沒辦法檢測到，由原型繼承的方法也沒辦法檢測到，以下為範例:

```
const obj ={}
obj.prop = 'exists'

console.log(obj.hasOwnProperty('prop') )
console.log(obj.hasOwnProperty('toString'))       // false
console.log(obj.hasOwnProperty('hasOwnProperty'))   // false
```

```
console.log('prop' in obj)
console.log('toString' in obj)
console.log('hasOwnProperty' in obj)
```

搭配物件類別定義使用時， `hasOwnProperty` 的行為是無法檢測出在類別中定義的方法，只能檢測該物件擁有的屬性，以及在建構式(constructor)中定義的物件擁有方法(算是一種具有函式值的屬性)。

```
class Base {
  constructor(a){
    this.a = a
    this.fnBase = function(){
      console.log('fnBase')
    }
  }

  baseMethod(){
    console.log('base')
  }

}

class Child extends Base{
  constructor(a, b){
    super(a)
    this.b = b
    this.fnChild = function(){
      console.log('fnChild')
    }
  }

  childMethod(){
    console.log('child')
  }

}

const aObj = new Child(1, 2)

console.log(aObj.hasOwnProperty('a'))
console.log(aObj.hasOwnProperty('b'))
console.log(aObj.hasOwnProperty('fnBase'))
console.log(aObj.hasOwnProperty('fnChild'))
console.log(aObj.hasOwnProperty('baseMethod')) //false
console.log(aObj.hasOwnProperty('childMethod')) //false

console.log('a' in aObj)
console.log('b' in aObj)
console.log('fnBase' in aObj)
console.log('fnChild' in aObj)
console.log('baseMethod' in aObj)
console.log('childMethod' in aObj)
```

`hasOwnProperty` 由於只有判斷物件本身屬性的限制，它會比較常被使用， `in` 運算符反而很少被用到。但這兩種判斷的效率都比直接用 `undefined` 判斷屬性值慢得多，所以要不就用 `undefined` 判斷就好，雖然這並不完全精準，要不然就用 `hasOwnProperty` 。

物件的遍歷(traverse)

在JavaScript中的定義，一般物件不是內建為可迭代的(Iterable)，只有像陣列、字串與TypedArray、Map、Set這三種特殊物件，才是可迭代的。所以這種一般稱為對物件屬性遍歷(traverse，整個走過一遍)或列舉(enumerate)的語句，而且一般物件的遍歷的效率與陣列的迭代相比非常的差。

> 註: `for..of` 只能用在可迭代的(Iterable)的物件上。

for...in語句

`for...in` 語句是用來在物件中以鍵(key)值進行迭代，因為是無序的，所以有可能每次運算的結果會不同。它通常會用來配合 `hasOwnProperty` 作判斷，主要原因是 `in` 運算符和前面在判斷時一樣，它會對所有原型鏈(prototype chain)都整個掃過一遍， `hasOwnProperty` 可以限定在物件本身的屬性。

```
for(let key in obj){
    if (obj.hasOwnProperty(key)) {
        console.log(obj[key]);
    }
}
```

> 註: `for...in` 語句不要用在陣列上，它不適合用於陣列迭代。

Object.keys轉為陣列，然後加上使用forEach方法

`Object.keys` 方法會把給定物件中可列舉(enumerable)的鍵，組合成一個陣列回傳，它的結果情況和 `for...in` 語句類似，差異就是在對原型鏈並不會整個掃過，只會對物件擁有的屬性的鍵。

```
Object.keys(obj).forEach(function(key){
    console.log(obj[key])
});
```

風格指引

- (Airbnb 22.3) 在命名類別或建構式時，使用大駝峰(PascalCase)命名方式。
- (Airbnb 9.4) 撰寫自訂的toString()方法是很好的，但要確定它是可以運作，而且不會有副作用的。
- (Airbnb 23.3) 如果 屬性/方法 是布林值，使用像isVal()或hasVal()的命名。

常見問題

物件導向程式設計在JavaScript語言中很重要嗎？

由物件導向程式設計以及週邊發展出的相關設計模式、API，到更進一步用於整體結構的程式框架，在現在流行的其他程式語言中，都是非常重要的程式語言特性。不過，在JavaScript中就很少有這類的發展情況，大致的原因有幾個:

- 以原型為基礎的與類別為基礎的物件導向的設計相當不同，許多程式開發的樣式是由函式發展而來，而非物件導向。
- JavaScript長久以上的執行環境是瀏覽器，首要任務是處理HTML的DOM結構，對於程式的執行效率與相容性為首要重點，JavaScript對於函式的設計反而較具彈性與效率，而物件導向的程式是高消費的應用程式，使得程式設計師較專注於函式的設計部份，也就是功能性(函式)導向(functional oriented)程式設計，而非物件導向的程式設計。
- 長久以來，許多設計模式都是由函式發展出來的而非物件導向，目前較為流行的許多工具函式庫，也都是以功能性(函式)導向程式設計，例如jQuery。另一方面，提倡以物件導向設計為主的函式庫或框架，除了效率一定不會太好之外，學習門檻反而很高，造成會使用的程式設計師很少。

以下是幾個物件導向使用的現況:

- 物件字面定義的資料描述，用於單純的物件資料類型，常稱之為只有資料的物件(data-only objects)
- 物件字面定義延伸出的JSON資料格式，成為JavaScript用來作資料交換的格式
- 物件導向的相關特性與語法只會拿來應用現成的DOM或內建物件，或是用來擴充原本內建的物件之用
- 對於程式碼的組織方式與命名空間的解決方案，主要會使用模組樣式(即IIFE)或模組系統為主要方式，而非物件導向的相關語法或模式
- 以合成(或擴充)代替繼承(composition over inheritance)

JavaScript中的物件可以多重繼承嗎？或是有像介面的設計？

沒有多重繼承，可以用合併多個物件產生新物件(合成 composition)、Mixins(混合)樣式，或是用ES6標準的Proxy物件，來達到類似多重繼承的需求。

介面或抽象類別也沒有，因為這些就是類別為基礎的物件導向才會有的概念。不過，有模擬類似需求的語法或樣式。