陣列

陣列是一種有順序的複合式的資料結構，用於定義、存放複數的資料類型，在JavaScript的陣列中，並沒有規定它能放什麼資料進去，可以是原始的資料類型、其他陣列、函式等等。

陣列是用於存放大量資料的結構，要如何有效地處理資料，需要更加注意。它的搭配方法與語法很多，也有很多作相同事情的不同方式，並不是每樣都要學，有些只需要用到再查詢相關用法即可。

> 註: 雖然陣列資料類型是屬於物件，但Array這個包裝物件的 `typeof Array` 也是回傳'function'

陣列定義

陣列定義有兩種方式，一種是使用陣列字面文字，以下說明定義的方式。

陣列的索引值(index)是從0開始的順序整數值，陣列可以用方括號([])來取得成員的指定值，用這個方式也可以改變成員包含的值:

```
const aArray = []
const bArray = [1, 2, 3]

console.log(aArray.length) //0

aArray[0] = 1
aArray[1] = 2
aArray[2] = 3
aArray[2] = 5

console.log(typeof aArray) // object
console.log(aArray) // [1,2,5]
console.log(aArray.length) //3
console.log(aArray[3]) //undefined
```

> 註: 陣列為參照的(reference)資料類型，其中包含的值是可以再更改的，這與const的常數宣告無關。

另一種是使用Array包裝物件的預先分配空間的方式，但這種方式並不建議使用，這種定義語法除了容易搞混之外，經測試過它對效能並沒有太大幫助。而且這語法在分配後，一定會把長度值(成員個數)固定住，除非你百分之百確定陣列裡面的成員個數，不然千萬不要用。以下為範例:

```
//警告: 不要使用這種定義方式

//預先分配的定義
const aArray = new Array(10)
//混淆語法，這是定義三個陣列值
const bArray = new Array(1, 2, 3)
console.log(aArray.length) //10
```

> 註: JavaScript的內建物件都不建議 `new` 作初始定義的。不過有一些特例是一定要的，例如Date、Error等等。

陣列定義注意事項

一開始就搞混的語法

注意初學者很容易犯的一個錯誤，就是用下面這種陣列定義語法，這語法並不會導致執行錯誤，這是很怪異的合法定義語法，特別在這裡指出來就是希望你不要搞混了:

```
//這是錯誤示範
const aArray = [10]
```

實際上這相當於:

```
const aArray = []
aArray[0] = 10
```

多維陣列

對JavaScript中的陣列結構來說，多維陣列指的是"陣列中的陣列"結構，只是在陣列中保存其他的陣列值，例如像下面的範例:

```
const magicMatrix = [
      [2, 9, 4],
      [7, 5, 3],
      [6, 1, 8]
]

magicMatrix[2][1]
```

維數過多時在處理上會愈複雜，一般常見的只有二維。通常需要另外撰寫專屬的處理函式，或是搭配額外的函式庫在容易上會較為方便。

關聯陣列

JavaScript中並沒有關聯陣列(Associative Array)這種資料結構，關聯陣列指的是"鍵-值"的資料陣列，也常被稱為字典(dictionary)的資料陣列，有很多程式語言有這樣的資料結構。

基本上在JavaScript中有複合類型只有物件和陣列，物件屬性的確是"鍵-值"對應的，但它並不是陣列，也沒有像陣列有這麼多方法可使用。雖然在ES6標準加入幾個特殊的物件結構，例如Set與Map，但目前的使用還不廣泛，其中支援的方法也少。在處理大量複合資料時，陣列的處理效率明顯高出物件許多，陣列的用途相當廣泛。

儲存多種資料類型

雖然並沒有規定說，你只能在同一個陣列中使用單一種資料類型。但是，在陣列中儲存多種不同的資料類型，絕對是個壞主意。在包含有大量資料的陣列中會嚴重影響處理效能，例如像下面這樣的例子。如果是多種資料類型，還不如先直接都用字串類型，需要取值時再作轉換。

```
var arr = [1, '1', undefined, true, 'true']
```

另外你需要注意的是，雖然數字類型在JavaScript中並沒有分浮點數或整數，但實際上在瀏覽器的JavaScript引擎(例如Google Chrome的V8引擎)中，整數的陣列的處理效率高於浮點數的陣列，可見其實引擎可以分辨各種不同的資料類型，然後會作最有效的儲存與運算，比你想像中聰明得很。

在ES6後加入了一種新式的進階資料結構，稱為型別陣列(Typed Arrays)，它是類似陣列的物件，但並非一般的陣列，也沒有大部份的陣列方法。這種資料結構是儲存特定的資料時使用的，主要是為了更有效率的處理二進位的資料(raw binary data)，例如檔案、圖片、聲音與影像等等。(註: Typed Arrays標準)

不過，就像上面一段說明的，聰明的JavaScript引擎在執行時會認得在一般陣列中儲存的資料類型，然後作最有效率的運算處理，在某些情況型別陣列(Typed Arrays)在運算上仍然不見得會比一般陣列還有效率。

多洞的陣列(Holey Arrays)或稀疏陣列(Sparse Arrays)

多洞的陣列代表你在定義陣列時，它的陣列值和索引(index)並沒有塞滿，或是從一個完整的陣列刪除其中一個(留了個空位)。像下面這樣的陣列定義:

```
const aArray = []
aArray[0] = 1
aArray[6] = 3

const bArray = [1, , 3]
```

另一種情況是，陣列大部份的值都是根本不使用的，例如用 `new Array(100)` 先定義出一個很大的陣列，但實際上裡面的值很少，這叫作稀疏陣列(Sparse Arrays)，其實多洞的陣列(Holey Arrays)或稀疏陣列(Sparse Arrays)都差不多指的是這一類的陣列，稀疏陣列可以再重新設計讓它在程式上的效率更高。這種陣列在處理效能上都會有很大的影響，在大的陣列中要避免使用到這樣的情況。

拷貝(copy)陣列

把原有的陣列指定給另一個變數(或常數)並不會讓它成為一個全新的陣列，這是因為當指定值是陣列時，是指定到同一個參照，也就是同一個陣列，看下面的範例:

```
const aArray = [1, 2, 3]
const bArray = aArray

aArray[0] = 100

console.log(bArray) //[100, 2, 3]
```

由範例可以看到， `bArray` 與 `aArray` 是共享同一陣列中的值，就像是不同名字的連體嬰，不論你在 `aArray` 或 `bArray` 中修改其中的值，增加或減少其中的值，都是對同一值作這件事。這種不叫拷貝陣列，只是指向同一個陣列而已。

拷貝陣列並不是像這樣作的，而且它可能是件複雜的事情，複雜的原因是陣列中包含的值，可以是各種不同的值，包含數字、字串、布林這些基本原始資料，也可以是其他陣列、物件、函式、其他特殊物件。而且物件類型的資料，都是使用參照(reference)指向的，裡面的資料也是複合式的，所以有可能也是複雜的。題外話是當你在進行拷貝物件資料時，也是會遇到同樣的複雜的情況。

拷貝陣列的情況，大致可以區分為淺拷貝(shallow copy)與深拷貝(deep copy)兩種。淺拷貝只能完全複製原陣列中，包含像數字、字串之類的基本原始資料值，而且當然是只有一維的平坦陣列，如果其中包含巢狀(多維)陣列、物件、函式、其他物件，只會複製到參照，意思是還是只能指向原來同一個值。

深拷貝(deep copy)反而容易理解，它是真正複製出另一個完全獨立的陣列。不過，深拷貝是一種高花費的執行程序，所以對於效率與精確，甚至能包含的特殊物件範圍都需要考慮。如果要進行深拷貝，一般就不直接用JavaScript內建的語法來達成，而是要用外部的函式庫例如jQuery、understore或lodash來作，而這些函式庫中的深拷貝不是只有支援陣列結構而已，同樣也支援一般物件或特殊物件的資料結構。不過，如果你的陣列結構很簡單，也可以用for或while迴圈自己作深拷貝這件事。

以下的方式主要是針對"淺拷貝"部份，一樣也有很多種方式可以作同樣這件事，以下列出四種:

展開(spread)運算符

> 推薦使用

ES6後的新運算符，長得像之前在函式章節講到的其餘參數，使用的也是三個點的省略符號(ellipsis)(...)，語法相當簡單，現在很常被使用:

```
const aArray = [1, 2, 3]
const copyArray = [...aArray]
```

它也可以用來組合陣列

```
const aArray = [1, 2, 3]
const bArray = [5, 6, ...aArray, 8, 9]
```

> 註: 展開(spread)運算符目前用babel轉換為ES5相容語法時，是使用 `concat` 方法

slice

slice(分割)原本是用在分割陣列為子陣列用的，當用0當參數或不加參數，相當於淺拷貝，這個方式是目前是效率較好的方式，語法也很簡單:

```
const newArray = oldArray.slice(0)
const newArray = oldArray.slice()
```

concat

concat(串聯)是用於合併多個陣列用的，把一個空的陣列和原先陣列合併，相當於拷貝的概念。在這裡寫出來是為了比較一下展開運算符:

```
const newArray = [].concat(oldArray)
```

for/while迴圈語句

迴圈語句也可以作為淺拷貝，語句寫起來不難也很直覺，只是相較於其他方式要多打很多字，通常不會單純只用來作淺拷貝。以下為範例程式:

```
const newArray = []

for (let i = 0, len = oldArray.length ; i < len ; i++){
    newArray[i] = oldArray[i]
}
```

```
const newArray = []
let i = oldArray.length

while (i--){
    newArray[i] = oldArray[i]
}
```

判別是否為陣列

最常見的情況是，如果有個函式要求它的傳入參數之一為陣列，而且確定不能是物件或其他類型。你要如何判斷傳進來的值是真的一個陣列？

直接使用 `typeof` 來作判斷是沒辦法作這件事的，它對陣列資料類型只會直接回傳 `'object'` 。

在JavaScript中，有很多種方式可以作同一件事，這件事也不意外，不過每種方法都有一些些不同的細節或問題。以下的 `variable` 代表要被判斷的變數(或常數)值。

isArray

> 推薦使用

最簡單的判斷語法應該是這個，用的是內建Array物件中的 `isArray` ，它是個ES5標準方法:

```
Array.isArray(variable)
```

constructor

下面這個是在Chrome瀏覽器中效能最佳的判斷方法，它是直接用物件的建構式來判斷:

```
variable.constructor === Array
```

如果你是要判斷物件中的其中屬性是否為陣列，你可以先判斷這個屬性是否存在，像下面這樣(prop指的是物件屬性):

```
variable.prop && variable.prop.constructor === Array
```

> 失效情況: 當使用在一個繼承自陣列的陣列會失效

instanceof

這也是用物件的相關判別方法來判斷， `instanceof` 是用於判斷是否為某個物件的實例，優點為語法簡潔清楚:

```
variable instanceof Array
```

> 失效情況: 處理不同window或iframe時的變數會失效

toString.call

> 推薦使用

這也是用物件中的 `toString` 方法來判斷，這是所有情況都可以正確判斷的一種。它也是萬用方式，可以判斷陣列以外的其他特別物件，缺點是效率最差:

```
Object.prototype.toString.call(variable) === '[object Array]'
```

> 註: jQuery、underscore函式庫中的判斷陣列的API是用這種判斷方法

> 註: 在JavaScript: The Definitive Guide, 6th Edition書中有提到， `Array.isArray` 其實就是用這個方式的實作方法。

方式結論

這幾個方式的選擇，我的建議是只要學最後一種就行了(不考慮舊瀏覽器就用第一種)，它可以正確判斷並應用在各種情況，有時候正確比再快的效能更重要，更何況它其實是萬用的，除了陣列之外也可以用於其它的判斷情況。雖然它的語法對初學者來說，可能無法在此時完全理解，不過就先知道要這樣用就行了。

參考資料: How do you check if a variable is an array in JavaScript?

陣列屬性與方法

陣列屬性與方法的細節多如牛毛，以下只列出常用到的方法與屬性。

屬性

length長度(成員個數)

`length` 用來回傳陣列的長度(成員個數)，這個屬性有時候是不可信的，就如同上面用 `new Array(10)` 定義時，會被固定住為10，不論現在的裡面的值有多少個。多洞的陣列中也是與目前有值成員的個數不同:

```
const aArray = [1, , undefined, '123'] //有洞的陣列
console.log(aArray.length) //4
```

多維陣列的情況上面有說明過了，只是陣列中的陣列而已，它的 `length` 只會回傳最上層陣列的個數:

```
const magicMatrix = [
      [2, 9, 4],
      [7, 5, 3],
      [6, 1, 8]
]

console.log(magicMatrix.length) //3
```

`length` 的整數值**竟然**是可以更動的，它並不是只能讀不能寫的屬性:

```
const bArray = [1, 2, 3]
console.log(bArray.length)

bArray.length = 4
console.log(bArray.length)
console.log(bArray)

bArray.length = 2
console.log(bArray.length)
console.log(bArray)
```

更動 `length` 經測試過，事實上是從陣列最後面"**截短(truncate)**"的語法，它的效率是所有類似功能語法中最好的:

```
const sArray = ['apple', 'banana', 'orange', 'mongo']

sArray.length = 2
console.log(sArray)
```

另外， `length` 指定為0也可以用於清空陣列，清空陣列一樣也是有好幾種方式，以下為各種清空陣列的範例程式碼，注意第一種的原陣列不能使用 `const` 宣告，就意義上它不是真的把原來的陣列清空。一般情況下第一種效率最好，第四種最差:

```
let aArray = ['apple', 'banana', 'orange', 'mongo']

//第一種
aArray = []

//第二種
aArray.length = 0

//第三種
while(aArray.length > 0) {
    aArray.pop();
}

//第四種
aArray.splice(0, aArray.length)
```

方法

indexOf

`indexOf` 是簡便的搜尋索引值用的方法，它可以給定一個要在陣列中搜尋的成員(值)，如果找到的話就會回傳成員的索引值，沒找到就會回傳"-1"數字。多個成員符合的話，它只會回傳最先找到的那個(一律是從左至右)，它的比對是使用完全符合的比較運算符(===)，可加入第二個參數，這是可選擇的，它是"開始搜尋的索引值"，如果是負整數則從最後面倒過來計算位置的(最後一個索引值相當於 `-1`)。

```
const aArray = ['a', 'b', 'c', 'a', 'c', 'c']

console.log(aArray.indexOf('a')) //0
console.log(aArray.indexOf('c')) //2
console.log(aArray.indexOf('c', 3)) //4
console.log(aArray.indexOf('a', -3)) //3，-3代表要從索引值3開始搜尋
```

pop與push、shift與unshift

> 副作用方法

陣列的傳統處理方法，pop是"砰出"最後面一個值，然後把這個值從陣列移除掉。push是"塞入"一個值到陣列最後面。shift與pop類似，不過它是砰出最前面的值。unshift則與push類似，它是塞到陣列列前面。

`pop` 的例子如下，它會回傳被砰出的值:

```
const aArray = [1, 2, 3]
const popValue = aArray.pop()

console.log(aArray) //[1,2]
console.log(popValue) //3
```

`push` 的例子如下，它則是回傳新的長度值:

```
const aArray = [1, 2, 3]
aArray.push(4)

console.log(aArray) //[1,2,3,4]

const pushValue = aArray.push(5)

console.log(aArray) //[1,2,3,4,5]
console.log(pushValue) //5
```

> 口訣記法: 有"p"的pop與push是針對陣列的"屁股"(最後面)。 `pop-corn` 是爆米花，所以pop用來爆出值的。有u的push與unshift是同一掛的。

concat

`concat` (串聯)是用於合併其他陣列或值，到一個陣列上的方法。它最後會回傳一個新的陣列。

> 語法: array.concat(value1[, value2[, ...[, valueN]]])

前面已經有看到它可以作陣列的淺拷貝，它與展開運算符可以互為替代，以下為一個簡單的範例:

```
const strArray = ['a', 'b', 'c']
const numArray = [1, 2, 3]
const aString = 'Hello'

const newArray = strArray.concat(numArray)
console.log(newArray)

//連鎖(chain)運算
const newArray1 = strArray.concat(numArray).concat(aString)
console.log(newArray1)

//展開運算符 相等的作法
const newArray2 = [...strArray, ...numArray]
console.log(newArray2)
```

> 注意: 陣列沒有運算符這種東西。但是字串類型可以用合併運算符(+)，或是用同名方法concat作合併。

slice

slice(分割)是用於分割出子陣列的方法，它會用淺拷貝(shallow copy)的方式，回傳一個新的陣列。這個方法與字串中的分割子字串所使用的的同名稱方法 `slice` ，類似的作法。

> 語法: array.slice(start[, end])

`slice` 使用陣列的索引值作為前後參數，大部份的重點都是這兩個索引值的正負值情況。以下是對於特殊情況的大致規則:

- 當開頭索引值為undefined時(空白沒寫)，它會以0計算，也就是陣列最開頭。
- 當索引值有負值的情況下，它是從最後面倒過來計算位置的(最後一個索引值相當於 `-1`)
- 只要遵守"開頭索引值"比"結束索引值"的位置更靠左，就有回傳值。
- `slice(0)` 相當於陣列淺拷貝

```
const aArray = [1, 2, 3, 4, 5, 6]

const bArray = aArray.slice(1, 3)
const cArray = aArray.slice(0)
const dArray = aArray.slice(-1, -3)
const eArray = aArray.slice(-3, -1)
const fArray = aArray.slice(1, -3)
```

splice

> 副作用方法

splice(粘接)這個字詞與上面的slice(分割)長得很像，但用途不相同，這個方法是用於刪除或增加陣列中的成員，以此來改變原先的陣列的。

為何會有這種用途？主要是要在陣列的中間"插入"幾個新的成員(值)，或是"刪掉"其中的幾個成員(值)用的。

> 語法: array.splice(start, deleteCount[, item1[, item2[, ...]]])

這個方法的參數值會比較多用起來會複雜些，先說明它的參數如下:

- start 是開始要作這件事的索引值，左邊從0開始，如果是負數則從右邊(最後一個)開始計算
- deleteCount 是要刪除幾個成員(值)，最少為0代表不刪除成員(值)
- item1... 這是要加進來的成員(值)，不給這些成員(值)的話，就只會作刪除的工作，不會新增成員(值)。注意如果是陣列值，會變成巢狀(子)陣列成員(值)。

實際上有幾個基本的用法範例，splice通常會用在一些特定的情況，也可能會搭配搜尋語法或迴圈語句。以下為常用的幾個範例:

插入一個新成員(值)在某個值之後

插入某個值之後需要先找出這個某個值的索引值，所以會用 `indexOf` 來找，不過如果是多個值的情況，就要用迴圈語句了，這是只有單個"某個值"的情況。下面的兩個範例的結果是相同的，所以你要在"某個值"的後面插入新成員(值)，"後面"代表新成員(值)的索引值還需要加1才行。

```
const dictionary = ['a', 'b', 'd', 'e', 'f']

//先找到b的位置，等會要在b後面插入c
const bIndex = dictionary.indexOf('b')

//bIndex大於-1代表有存在，插入c，不刪除
if (bIndex > -1) {
    dictionary.splice(bIndex+1, 0, 'c')
}
```

用新成員(值)取代某個值

單個值的情況:

```
const dictionary = ['a', 'b', 'x', 'd', 'e']

//先找到x的位置，等會要用c來取代x
const xIndex = dictionary.indexOf('x')

//bIndex大於-1代表有存在，插入c，刪除x
if (bIndex > -1) {
    dictionary.splice(xIndex, 1, 'c')
}
```

多個值的情況，用迴圈語句:

```
const dictionary = ['x', 'b', 'x', 'x', 'b']

for (let i = 0, len = dictionary.length; i < len; i++){

    if (dictionary[i] === 'x'){
        dictionary.splice(i, 1, 'c')
    }
}
```

用於刪除成員(值)

> 註: 其實完全與取代範例幾乎一樣

單個值的情況:

```
const dictionary = ['a', 'b', 'x', 'd', 'e']

//先找到x的位置，等會要刪除
const xIndex = dictionary.indexOf('x')

//xIndex大於-1代表有存在，刪除x
if (xIndex > -1) {
    dictionary.splice(xIndex, 1)
}
```

多個值的情況，用迴圈語句:

```
const dictionary = ['x', 'b', 'x', 'x', 'b']

for (let i = 0, len = dictionary.length; i < len; i++){

    if (dictionary[i] === 'x'){
        dictionary.splice(i, 1)
```

```
    }
}
```

陣列與字串 join與split

join(結合)與split(分離)是相對的兩個方法，join(結合)是陣列的方法，用途是把陣列中的成員(值)組合為一個字串，組合的時候可以加入組合時用的分隔符號(或空白字元)。

```
const aArray = ['Hello', 'Hi', 'Hey']
const aString = aArray.join()   //Hello,Hi,Hey
const bString = aArray.join(', ') //Hello, Hi, Hey
const cString = aArray.join('') //HelloHiHey
```

split(分離)是倒過來，它是字串中的方法，把字串拆解成陣列的成員，拆解時需要給定一個拆解基準的符號(或空白)，通常是用逗號(,)分隔，以下為範例:

```
const monthString = 'Jan,Feb,Mar,Apr,May'
const monthArray1 = monthString.split(',') //["Jan", "Feb", "Mar", "Apr", "May"]

//以下為錯誤示範
const monthArray2 = monthString.split()  //["Jan,Feb,Mar,Apr,May"]
const monthArray3 = monthString.split('') //["J", "a", "n", ",", "F", "e", "b", ",", "M", "a", "r", ",", "A", "p", "r", ",", "M",
```

迭代

> forEach為副作用方法

陣列的迭代可以單純地使用迴圈語句都可以作得到，但在ES6後加入幾個新的方法，例如 `forEach` 、 `map` 與 `reduce` 提供更多彈性的運用。

forEach

`forEach` 類似於for迴圈，但它執行語句的是放在回調函式(callback)的語句中，下面這兩個範例是相等功能的:

```
const aArray = [1, 2, 3, 4, 5]
for (let i = 0, len = aArray.length; i < len; i++) {
    // 對陣列元素作某些事
    console.log(i, aArray[i])
}
```

```
const aArray = [1, 2, 3, 4, 5]
aArray.forEach(function(value, index, array){
// 對陣列元素作某些事
console.log(index, value)
})
```

forEach的回調函式(callback)共可使用三個參數:

- value 目前成員的值
- index 目前成員的索引
- array 要進行尋遍的陣列

forEach雖然可以與for迴圈語句相互替代，但他們兩個設計原理不同，也有一些明顯的差異，在選擇時你需要考慮這些。不同之處在於以下幾點:

- forEach無法提早結束或中斷
- forEach被呼叫時， `this` 會傳遞到回調函式(callback)裡
- forEach方法具有副作用

map(映射)

map(映射)反而是比較常被使用的迭代方法，由於它並不會改變輸入陣列(呼叫map的陣列)的成員值，所以並不會產生副作用，現在有很多程式設計師改用它來作為陣列迭代的首選使用方法。

map(映射)一樣會使用回調函式(callback)與三個參數，而且行為與forEach幾乎一模一樣，不同的地方是它會回傳一個新的陣列，也因為它可以回傳新陣列，所以map(映射)可以用於串接(chain)語法結構。

```
var aArray = [1, 2, 3, 4];
var newArray = aArray.map(function (value, index, array) {
    return value + 100
})

//原陣列不會被修改
console.log(aArray) // [1, 2, 3, 4]
console.log(newArray) // [101, 102, 103, 104]
```

reduce(歸納)

reduce(歸納)這個方法是一種應用於特殊情況的迭代方法，它可以藉由一個回調(callback)函式，來作前後值兩相運算，然後不斷縮減陣列中的成員數量，最終回傳一個值。reduce(歸納)並不會更動作為傳入的陣列(呼叫reduce的陣列)，所以它也沒有副作用。

reduce(歸納)比map又更複雜了一些，它多了一個參數值，代表前一個值，主要就是它至少要兩個值才能進行前後值兩相運算。你也可以給定它一個初始值，這時候reduce(歸納)會從索引值的0接著取第二個值進行迭代，如果你沒給初始值，reduce(歸納)會從索引值1開始開始迭代。可以用下面這個範例來觀察它是如何運作的:

```
const aArray = [0, 1, 2, 3, 4]

const total = aArray.reduce(function(pValue, value, index, array){
    console.log('pValue = ', pValue, ' value = ', value, ' index = ', index)
    return pValue + value
})

console.log(aArray) // [0, 1, 2, 3, 4]
console.log(total) // 10

//下面給定初始值10，注意它放的地方是在回調函式之後的參數中
const total2 = aArray.reduce(function(pValue, value, index, array){
    console.log('pValue = ', pValue, ' value = ', value, ' index = ', index)
    return pValue + value
}, 10)

console.log(total2) // 20
```

按照這個邏輯，reduce(歸納)具有分散運算的特點，可以用於下面幾個應用之中:

- 兩相比較最後取出特定的值(最大或最小值)
- 計算所有成員(值)，總合或相乘
- 其它需要兩兩處理的情況(組合巢狀陣列等等)

排序與反轉 sort與reverse

sort

> 副作用方法

sort是一個簡單的排序方法，以Unicode字串碼順序來排序，對於英文與數字有用，但中文可能就不是你要的。以下為簡單的範例:

```
const fruitArray = ['apple', 'mongo', 'cherry', 'banana' ]
fruitArray.sort()
console.log(fruitArray) //["apple", "banana", "cherry", "mongo"]


const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉' ]
fruitArray.sort()
console.log(fruitArray) //["櫻桃", "芒果", "蘋果", "香蕉"]
```

中文的排序一般來說只會有兩種情況，一種是要依big5編碼來排序，另一種是要依筆劃來排序，這時候需要在sort方法傳入參數中，另外加入比較的回調(callback)函式，這個回調函式中將使用localeCompare這個可以比較本地字串的方法，以下為範例程式，其中本地(locale)的參數請參考locales argument:

```
const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉', '大香蕉', '小香蕉' ]

//使用原本的排序
fruitArray.sort()

console.log(fruitArray)
//["大香蕉", "小香蕉", "櫻桃", "芒果", "蘋果", "香蕉"]


fruitArray.sort(function(a, b){
  //zh-Hans-TW、zh-Hans-TW-u-co-big5han、pinyin等等參數同樣結果
  return a.localeCompare(b, 'zh-Hans-TW')
})

console.log(fruitArray)
//["大香蕉", "芒果", "蘋果", "香蕉", "小香蕉", "櫻桃"]

fruitArray.sort(function(a, b){
  //按筆劃從小到大排序
  return a.localeCompare(b, 'zh-Hans-TW-u-co-stroke')
})

console.log(fruitArray)
//["大香蕉", "小香蕉", "芒果", "香蕉", "蘋果", "櫻桃"]
```

> 註: 這個字串比較方法應該可以再最佳化其效率，有需要可以進一步參考其文件的選項設定

reverse

> 副作用方法

reverse(反轉)這語法用於把整個陣列中的成員順序整個反轉，就這麼簡單。以下為範例:

```
const fruitArray = ['蘋果', '芒果', '櫻桃', '香蕉' ]
fruitArray.reverse()

console.log(fruitArray)
//["香蕉", "櫻桃", "芒果", "蘋果"]
```

另一個情況是字串中的字元，如果要進行反轉的話，並沒有字串中的 `reverse` 方法，要用這個陣列的 `reverse` 方法加上字串與陣列的互相轉換的split與join方法，可以使用以下的函式:

```
function reverseString(str) {
    return str.split('').reverse().join('');
}
```

過濾與搜尋 filter與find/findIndex

filter(過濾)是使用一個回調(callback)函式作為傳入參數，將的陣列成員(值)進行過濾，最後回傳符合條件(通過測試函式)的陣列成員(值)的新陣列。它的傳入參數與用法與上面說的迭代方法類似，實際上也是另一種特殊用途的迭代方法:

```
const aArray = [1, 3, 5, 7, 10, 22]

const bArray = aArray.filter(function (value, index, array){
        return value > 6
    })

console.log(aArray) //[1, 3, 5, 7, 10, 22]
console.log(bArray) //[7, 10, 22]
```

find與findIndex方法都是在搜尋陣列成員(值)用的，一個在有尋找到時會回傳值，一個則是回傳索引值，當沒找到值會回傳undefined。它們一樣是使用一個回調(callback)函式作為傳入參數，來進行尋找的工作，回調函式的參數與上面說的迭代方法類似。

findIndex與最上面說的indexOf不同的地方也是在於，findIndex因為使用了回調(callback)函式，可以提供更多的在尋找時的彈性應用。以下為範例:

```
const aArray = [1, 3, 5, 7, 10, 22]
const bValue = aArray.find(function (value, index, array){
        return value > 6
    })
const cIndex = aArray.findIndex(function (value, index, array){
        return value > 6
    })

console.log(aArray) //[1, 3, 5, 7, 10, 22]
console.log(bValue) //7
console.log(cIndex) //3
```

陣列處理純粹函式

改寫原有的處理方法(或函式)為純粹函式並不困難，相當於要拷貝一個新的陣列出來，進行處理後回傳它。ES6後的展開運算子(...)可以讓語法更簡潔。

> 註: 以下範例並沒有作傳入參數的是否為陣列的檢查判斷語句，使用時請再自行加上。

> 註: 下列範例來自Pure javascript immutable arrays，更多其他的純粹函式可以參考這裡的範例。

push

```
//注意它並非回傳長度，而是回傳最終的陣列結果
function purePush(aArray, newEntry){
  return [ ...aArray, newEntry ]
}

const purePush = (aArray, newEntry) => [ ...aArray, newEntry ]
```

pop

```
//注意它並非回傳pop的成員(值)，而是回傳最終的陣列結果
function purePop(aArray){
  return aArray.slice(0, -1)
}

const purePush = aArray => aArray.slice(0, -1)
```

shift

```
//注意它並非回傳shift的成員(值)，而是回傳最終的陣列結果
function pureShift(aArray){
  return aArray.slice(1)
}

const pureShift = aArray => aArray.slice(1)
```

unshift

```
//注意它並非回傳長度，而是回傳最終的陣列結果
function pureUnshift(aArray, newEntry){
  return [ newEntry, ...aArray ]
}

const pureUnshift = (aArray, newEntry) => [ newEntry, ...aArray ]
```

splice

這方法完全要使用slice與展開運算符(...)來取代，是所有的純粹函式最難的一個。

```
function pureSplice(aArray, start, deleteCount, ...items) {
  return [ ...aArray.slice(0, start), ...items, ...aArray.slice(start + deleteCount) ]
}

const pureSplice = (aArray, start, deleteCount, ...items) =>
[ ...aArray.slice(0, start), ...items, ...aArray.slice(start + deleteCount) ]
```

sort

```
//無替代語法，只能拷貝出新陣列作sort
function pureSort(aArray, compareFunction) {
  return [ ...aArray ].sort(compareFunction)
}

const pureSort = (aArray, compareFunction) => [ ...aArray ].sort(compareFunction)
```

reverse

```
//無替代語法，只能拷貝出新陣列作reverse
function pureReverse(aArray) {
  return [ ...aArray ].reverse()
}

const pureReverse = aArray => [ ...aArray ].reverse()
```

delete

刪除(delete)其中一個成員，再組合所有子字串:

```
function pureDelete (aArray, index) {
   return aArray.slice(0,index).concat(aArray.slice(index+1))
}

const pureDelete = (aArray, index) => aArray.slice(0,index).concat(aArray.slice(index+1))
```

英文解說

常見問題

為什麼要那麼強調副作用？

副作用的概念早已經存在於程式語言中很久了，但在最近幾年才受到很大的重視。在過去，我們在撰寫這種腳本直譯式程式語言，最重視的其實是程式效率與相容性，因為同樣的功能，不同的寫法有時候效率會相差很多，也有可能這是不同瀏覽器品牌與版本造成的差異。

但現在的電腦硬體已經進步太多，所謂的執行資源的限制早就與10年前不同。效率在今天的程式開發早就已經不是唯一的重點，更多其他的因素都需要加入來一併考量，以前的應用程式也可能只是小小的特效或某個小功能，現在的應用程式將會是很龐大而且結構複雜的。所以，程式碼的閱讀性與語法簡潔、易於測試、維護與除錯、易於規模化等等，都會變成要考量的其他重點。純粹函式的確是未來的主流想法，當一個應用程式慢慢變大、變複雜，純粹函式可以提供的好處會變成非常明顯，所以一開始學習這個概念是必要的。

參考

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。

迴圈

迴圈(loop)是"重覆作某個事"的語句結構，用比較詳細的解釋，則是"在一個滿足的條件下，重覆作某件事幾次"這樣的語句。

迴圈語句經常會搭配陣列與物件之類的集合性資料結構使用，重覆地或循環地取出某些滿足條件的值，或是重覆性的更動其中某些值。

for語句

for語句需要三個表達式作為參數，它們各自有其功用，分別以分號(;)作為區分。for迴圈的基本語法如下:

```
for ([initialExpression]; [condition]; [incrementExpression])
  statement
```

- initialExpression: 初始(開始時)情況的表達式
- condition: 判斷情況或測試條件，代表可執行for中包含執行動作的滿足條件
- incrementExpression: 更新表達式，每次for迴圈中的執行動作語句完成即會執行一次這個表達式

一個簡單的範例如下:

```
for (let count = 0 ; count < 10 ; count++){
    console.log(count)
}
```

整個程式碼的解說是:

- 在迴圈一開始時，將 `count` 變數指定為數字 `0`
- 定義當 `count < 10` 時，才能滿足執行 `for` 中區塊語句的條件。也就是只有在 `count < 10` 時才能執行for中區塊語句。
- 每次當執行 `for` 中區塊的語句執行後，即會執行 `count++`

可自訂的表達式們

`for` 語句中共有三個表達式，分別代表迴圈進行時的設定與功用，這三個表達式的運用方式就可以視不同的應用情況而調整。所有的三個表達式都是可自訂的，也是可有可無。下面這個看起來怪異的 `for` 語句是一個無窮迴圈，執行這個語句會讓你的瀏覽器當掉:

```
//錯誤示範
for (;;){
    console.log('bahbahbah')
}
```

> 註: 用 `for(;;)` 這樣的語法，基本上失去 `for` 語句原本的意義，完全不建議使用。

> 注意: 迴圈是一個破壞性相當高的語句，如果不小心很容易造成整個程式錯誤或當掉。

第一個表達式是用作初始值的定義。它是可以設定多個定義值的，每個定義值之間使用逗號(,)作為分隔，定義值可使用的範圍只在迴圈內部的語句中:

```
for (let count = 0, total = 10 ; count < 10 ; count++){
    console.log(count, total)
}
```

> 註: 初始值同樣也可以定義在迴圈之外，但它的作用範圍會擴大到迴圈之外。

第二個表達式是用於判斷情況(condition)，也就是每次執行迴圈中區塊的語句前，都會來測試(檢查)一下，是不是滿足其中的條件，如果滿足的話(布林 `true` 值)，就執行迴圈中區塊的語句，不滿足的話(布林 `false` 值)，就跳出迴圈語句或略過它。這個表達式就是會造成無止盡的或重覆執行次數不正確的原兇，所以它是這三個表達式中要特別注意的一個。

由於它是一個標準的判斷情況(condition)表達式，如果有多個判斷情況的時候，要使用邏輯與(&&)和邏輯或(||)來連接，邏輯或(||)因為結果會是 `true` 的情況較為容易，所以要更加小心:

```
for (let count = 0, total = 10 ; count < 10 && total < 20 ; count++){
    console.log(count, total)
}
```

第三個表達式是用於更新的，常被稱為遞增表達式(incrementExpression)。不過它也不只用在遞增(increment)，應該說用在"更動(update)"或許會比較恰當，每次一執行完迴圈內部語句就會執行一次的表達式。它的作用相當於在迴圈內的區塊中的最後面加上這個語句，例如下面這個for迴圈和最上面一開始的範例的結果是一樣的:

```
for (let count = 0 ; count < 10 ;){
    console.log(count)
    count++
}
```

多個表達式也是可以的，同樣也是用逗點(,)分隔不同的表達式:

```
for (let count = 0, total = 30 ; count < 10 && total > 20 ; count++, total--){
    console.log(count, total)
}
```

這裡有個小小問題，是有關於遞增運算符(++)與遞減運算符(--)，它們運算元的位置差異，放在運算元的前面和後面差異是什麼？例如以下的範例:

```
let x = 1
let y = 1

console.log(x++) //1
console.log(x) //2

console.log(++y) //2
console.log(y) //2
```

所以放在運算元(值)前面(先)的遞增(++)或遞減(--)符號，就會先把值作加1或減1，也就會直接變動值。放後面的就是值的變動要在下一次再看得到。

> 口訣: 遞增減運算符(++/--) "錢仙"=(放)前(面)先(變動)

小結

實際上這三個表達式並沒有限定只能用哪些表達式，只要是合法的表達式都可以，只不過第二個表達式會作判斷情況，也就是轉換為布林值，這一點要注意的。有些程式設計師喜歡用這個特性寫出感覺很高超的語法，只是讓別人更難看懂在寫什麼而已，我完全不認同這種寫法，如何提供高閱讀性的程式碼才是最好的寫法。

至於每個表達式執行的情況，第一個表達式(初始化值)只會在讀取到 `for` 語句時執行一次，之後不會再執行。第二個表達式(判斷情況)會在讀取到 `for` 語句時執行一次，之後每次重覆開始時都會執行一次。第三個表達式會在有滿足條件下，執行到 `for` 語句區塊內部的語句的最後才執行。

多重迴圈

所謂的多重迴圈是巢狀的迴圈語句結構，也就是在迴圈的區塊語句中，再加上另一個內部的迴圈語句。典型的範例就是九九乘法表:

```
for (let i = 1 ; i < 10 ; i++){
  for (let j = 1 ; j < 10 ; j++){
    console.log(i + ' x ' + j + ' = ', i * j )
  }
}
```

> 註: 下面的 `while` 語句與 `do...while` 語句也可以寫為多重迴圈(巢狀迴圈)，類似上面的程式碼，就不再多重覆說明了。

while語句

相較於需要很確定重覆次數(重覆次數為可被計算的)的 `for` 迴圈， `while` 迴圈語句可以說它是 `for` 迴圈語句的簡單版本，你可以根據不同的使用情況選擇要使用哪一種。它只需要一個判斷情況(condition)的表達式即可，它的基本語法結構如下:

```
while (condition)
  statement
```

一個簡單的範例如下，這個範例與之前的for迴圈的範例功能是相等的:

```
let count = 0

while (count < 10) {
    console.log(count)
    count++
}
```

while語句沒什麼特別要說明的，它把更新用表達式的部份交給程式設計師自行處理，在while語句中的區塊語句中再加上。使用時仍要避免出現無窮迴圈或重覆次數不正確的情況。

do...while語句

`do...while` 語句是 `while` 語句的另一種變形的版本，差異只在於"它會先執行一次區塊中的語句，再進行判斷情況"，也就是會"保証至少執行一次其中的語句"，它的基本語法結構如下:

```
do
  statement
while (condition)
```

正常的使用情況與while語句沒什麼差異，只是位置上下顛倒再加個 `do` 而已:

```
let count = 0

do {
  console.log(count)
  count++
}
while (count < 10)
```

實際上 `while` 語句如果要相等於 `do...while` 的語句，應該是如下面這樣，也就是while語句前要先執行一次 `do...while` 中的語句才能算相等:

```
do_statment
while (condition)
  do_statment
```

因為 `do...while` 語句具有先執行語句再檢查，也就是保証會執行一次的特性，要視應用情況來使用，以簡單的實際例子來說明 `while` 與 `do...while` 語句的應用情況:

- `while`：當有一群人要進入電影院，需要對每個入場者收取門票與檢查，然後才能入場
- `do...while`：當有幾組參賽者參加歌唱比賽，需要先表演後再對每個參賽者評分

for...in語句

`for...in` 語句主要是搭配物件類型使用的，不要使用在陣列資料類型。它是用於迭代物件的可列舉的屬性值(enumerable properties)，而且是任意順序的。因為陣列資料在進行迭代時，順序是很重要的，所以陣列資料類型不會用這個語句，而且陣列資料類型本身就有太多好用的迭代方法。

另外， `for` 迴圈語句完全可以取代 `for...in` 語句，而且 `for` 迴圈語句可以保証順序，所以這個語句算是不一定要使用的，在這裡說明只是單純的比較其他的語句而已。

> 註: `for...in` 語句是任意順序的迴圈語句，使用時通常是用來檢測用的語句

> 註: 對於複雜的物件資料類型，單純使用JavaScript中的語言特性是常常有所不足，建議是使用其他的函式庫中的API，例如jQuery、Lodash或underscore.js。

for...of語句

`for...of` 語句是新的ES6語句，可以用於可迭代物件上，取出其中的值，可迭代物件包含陣列、字串、Map物件、Set物件等等。簡單的範例如下:

```
//用於陣列
let aArray = [1, 2, 3]

for (let value of aArray) {
  console.log(value)
}

//用於字串
let aString = "abcd";

for (let value of aString) {
  console.log(value);
}
```

break與continue

`break` (中斷)與 `continue` (繼續)是用於迴圈區塊中語句的控制，在 `switch` 語句中也有看過 `break` 這個關鍵字的用法，是一個跳出 `switch` 語句的語法。

`break` 是"中斷整個迴圈語句"的意思，可以中斷整個迴圈，也就是"跳出"迴圈區塊的語句，如果是在巢狀迴圈時是跳出最近的一層。 `break` 通常會用在迴圈語句中的 `if` 判斷語句裡，例如下面這個例子:

```
let count = 0

while (count < 10) {
    console.log(count)
    //count的值為6時將會跳出迴圈
    if(count === 6) break
    count++
}
```

`continue` 是"繼續下一次迴圈語句"的意思，它會忽略在 `continue` 之下的語句，直接跳到下一次的重覆執行語句的最開頭。因為 `continue` 有"隱含的 `goto` 到迴圈的最開頭程式碼"，它算是一個在JavaScript語言中的壞特性，它會讓你的判斷情況(condition)整個無效化，也可以破壞整體的迴圈語句執行結構，容易被濫用出現不被預期的結果，結論是能不用就不要使用。一個簡單的範例如下:

```
let count = 0
let a = 0

while (count < 10) {
    console.log('count = ', count)
    console.log('a = ', a)

    count++
    //count的值大於為6時將會不再遞增a變數的值
    if(count > 6) continue
    a++
}
```

> 註: 雖然JavaScript語言中並沒有 `goto` 語法，但迴圈語句中的 `continue` 搭配 `labeled語句` ，相當於其他程式語言中的 `goto` 語法。 `goto` 語法幾乎在所有的程式語言中，都是惡名昭彰的一種語法結構。

風格指引

- (Airbnb 18.3) 在控制語句的圓括號開頭(()前放一個空格(if, while等等)。在函式呼叫與定義的函式名稱與傳入參數之間就不需要空格。eslint: keyword-spacing jscs: requireSpaceAfterKeywords
- (Airbnb 7.3) 絕對不要在非函式區塊中宣告一個函式(if, while等等)。改用指定一個函式給一個變數。瀏覽器可以允許你可以這樣作，但是壞消息是，不同瀏覽器可能會轉譯為不同的結果。eslint: no-loop-func
- (Google) for-in迴圈經常會錯誤地被使用在陣列上。當使用陣列時，總是使用一般的for迴圈。
- 在迴圈中區塊語句裡，定義變數/常數是個壞習慣，應該是要定義在迴圈外面或for迴圈的第一個表達式中。

```
//壞的寫法
for (let i=0; i<10; i++){
  let j = 0
  console.log(j)
  j++
}

//好的寫法
for (let i=0, let j = 0; i<10; i++){
  console.log(j)
  j++
}
```

- 用於判斷情況的運算，應該要儘可能避免，先作完運算在迴圈外部或或for迴圈的第一個表達式中，效率可以比較好些。

```
//較差的寫法
for (let i=0 ; i < aArray.length ; i++) {
  //statment
}

//較好的寫法
for (let i=0, len = aArray.length ; i < len; i++) {
  //statment
}

//另一種寫法，這種寫法腦袋要很清楚才看得懂
for (let i = aArray.length; i--;){
  //statment
}

//較差的寫法
let i = 0
while (i < aArray.length) {
  //statment
  i++
}

//較好的寫法
let i = 0
let len = aArray.length

while ( i < len) {
  //statment
  i++
}
```

> 註: 上面有一說法是在陣列資料類型的迴圈中的第三表達式(更新用表達式)中，使用i--會比i++效率更佳。實際上在現在的瀏覽器上這兩種的執行速度是根本一樣，沒什麼差異。

結語

家庭作業

控制流程

由於程式碼的執行順序，是由最上方的程式碼開始，往下逐行執行。有些時候，我們需要在其中特定的位置，判斷在當時執行期間的情況值，來決定之後程式執行的走向，這稱之為控制流程(Control Flow/康錯 佛肉/)的語法結構。

舉例來說，電腦中的文字冒險類遊戲(例如: 美少女遊戲Galgame)中，主人翁在關鍵時刻必須對攻略對象，在故事進行的對話選項間作出選擇，依照一連串不同的選擇，有可以導向最後是好的結局(GoodEnding)或是壞的結局(BadEnding)。這種依選項(判斷情況)不同，而導致不同的執行結果的程式碼語法結構，就是控制流程的結構。

Expression(表達式)

Expression/依斯伯累遜/ (表達式)是以字面文字(literal/立的羅/)、變數/常數名稱、值(運算元)、 運算符，或其他Expression(表達式)的組合體，最終能運算而計算求出(evaluates)一個值。

> Expression(表達式)代表任何合法的可計算產出值的程式碼單位

簡單的表達式範例如下，這看起有點像某行程式碼的一部份，或只是單純的字面文字(固定值的意思，例如 數字 或 字串):

```
'Hello'
3.1415
x = 7
3 + 5
```

Side Effect(副作用)

你如果有去醫院看過醫生拿過藥，在藥包上都會註明吃了這個藥物後會產生的副作用，Side Effect/塞的 依費特/(副作用)就是這個意思。Side Effect(副作用)這個詞，經常會出現在很多進階使用的JavaScript框架或函式庫之中，它算是一種概念，在表達式中有使用這個概念來進行分類。Expression(表達式)就上面所說的，是用來求出值的，但常見的一些Expression(表達式)的作用在指定值，而當在指定原本的變數/常數的Expression(表達式)值時，如果會變動原本的變數/常數內的值，就稱為是"**具有Side Effect(副作用)**"的Expression(表達式)，例如像下面這些具有副作用的表達式範例:

```
counter++
x += 3
y = "Hello " + name
```

沒有副作用的表達式的範例，它們大概都只是字面文字(literal)或變數/常數名稱，或是產生一個新的運算結果值:

```
3 + 5
true
1.9
x
x > y
'Hello World'
```

Side Effect(副作用)並不是指好或不好的意思，而是它有可能會影響到其他環境的使用情況，在使用時要特別注意這樣。它除了在表達式中有這個概念，我們在函式中也會再見到它。這裡就先大概了解表達式是如何區分有無副作用的。

Statements(語句)

Statements/斯疊門/(語句、陳述)是在程式語言中，一小段功能性的程式碼，語句中包含了關鍵字與合法的語法(Syntax/新泰斯/)。在JavaScript語言中，傳統上是以半形分號(;)作為代表結束與分隔其他的語句，但也可以不使用半形分號(;)，而改以斷行來區分。撰寫一支程式，就如同在寫一篇文章時，其中會包含了各種描述語句。

> Statements(語句)可視為在JavaScript的最小獨立執行程式碼組合

```
//註解也是一種語句
```

```
//指定值也是一種語句
const x = 10

//block statement(區塊語句)
{
  statement_1
  statement_2
  ...
  statement_n
}

//function statement(函式語句)
function name() {
   [statements]
}
```

在JavaScript語言中，Expression(表達式)主要用來產生"值"，因為它的功用很特殊，通常會獨立出來說明稱之為Expression Statements(表達式語句)。

一般的Statements(語句)主要功能是執行動作或定義某種行為，例如之前說過的註解(Comment)就是一種語句。

Statements(語句)還可以依不同情況的使用進行分類，以下列出:

- 控制流程(Control flow)
- 定義(Declarations)
- 函式與類別(Functions and classes)
- 迭代/迴圈(Iterations/依特瑞遜/)
- 其他(Others)

> 註: 此分類參考MDN的分類方式，ECMAScript標準分類並不是這樣。

> 註: Iterations/依特瑞遜/這個字詞，中文一般是翻譯為"迭代"這個字，這個中文字詞太過專業，對初學者來說是有看沒有懂。比較好的理解是它有"重覆作某件事"的意思，也就是和"迴圈"、"重覆"的意思相近。

控制流程

控制流程(flow control)語句是一種特殊的語句，它與程式的執行流程有關，會因為其中的判斷結果不同，導致不同的執行結果。

區塊(block)語句

區塊(block)語句可以組合多行語句，或用於分隔不同語句，區塊(block)語句經常使用於控制流程的語句之中。使用花括號(curly brackets)標記({})，將多行語句包含在其中:

```
{
  statement_1
  statement_2
  ...
  statement_n
}
```

if...else語句

`if/伊芙/...else/埃額斯/` 是常見的控制流程的語句，在中文的意思就是"如果...要不然"。所以整體的結構就像是中文的意思"如果aaa命題為真情況下，就xxx，要不然就ooo"。這個aaa就是一個判斷情況(condition)，判斷情況使用的都是比較運算符(Comparison operators)與邏輯運算符(Logical Operators)，而判斷情況結果則是會用布林的 `true` /觸/ 與 `false` /佛斯/ 值來判斷，但要特別注意之前說明的"falsy/佛西/"情況，這些在判斷情況中會自動轉成布林的 `false` 。

一個簡單的 `if...else` 語句的範例如下:

```
const x = 10

if (x > 100) {
    console.log('x > 100')
```

```
} else {
    console.log('x < 100')
}
```

> 註: `if...else` 並不是寫了 `if` 就一定要有 `else` ，也可以單獨只用 `if` 語句，但 `else` 不能單獨使用。

`if...else` 語句有幾個常見的延申結構，例如多個判斷情況時可以再其中加入 `else if` 。不過當你在寫這些判斷情況時，最好是讓它有真的會出現的情況，以免寫出根本不會有的判斷情況，成為程式碼的多餘部份:

```
const x = 10

if (x > 100) {
    console.log('x > 100')
} else if ( x < 50){
    console.log('x < 100 but x > 50')
} else {
    console.log('x < 50')
}
```

另一種是巢狀的語法結構，也就是進入某個判斷情況下，可以在裡面再寫出另一個 `if...else` 語句，例如:

```
const x = 10

if (x > 100) {
    if (x > 500){
        console.log('x > 500')
    }else{
        console.log('x > 100 but x < 500')
    }
} else if ( x < 50){
    console.log('x < 100 but x > 50')
} else {
    console.log('x < 50')
}
```

在判斷後的執行語句，如果是只有一行的情況下，就可以不需要使用區塊語句({})。也可以用空一格然後把判斷情況與結果語句寫在同一行，不過要注意這裡不要寫過長的語句，以免造成閱讀上的困難。例如:

```
const x = 10

//去除block statement
if (x > 100)
    console.log('x > 100')
else
    console.log('x < 50')

//寫成一行
if (x === 10) console.log('x is 10')
```

而在使用 `if` 加上 `else` 時，為了簡化語句，使用三元運算符(Ternary Operator)(?:)，來讓程式碼更簡潔，這也只能用於簡單的判斷情況與執行語句時:

```
const x = 10

(x > 100) ? console.log('x > 100') : console.log('x < 50')
```

> 注意: 建議只使用三元運算符在單一行的語句上，雖然它可以使用在多行與巢狀結構。

由於三元運算符的語法很簡單，它也常被配合用在指定值的語句中，例如:

```
const foo = value1 > value2 ? 'baz' : null
```

這個像我們之前說過的，用邏輯或運算符(||)來指定變數/常數預設值的語句。不過如果是單純的判斷某個值是否存在，然後設定它為預設值，用邏輯或運算符(||)是比較好的作法，例如:

```
//相當於 const foo = a ? a : b
const foo = a || b
```

> 心得提示: 因為人生還很長，而我們需要寫的程式碼還有很多，所以你會本書上看到有很多感覺上像是偷懶或密技的簡短語法，或許它並不是出現在標準中，也不是課堂上會教的"正規"寫法，但的確是廣泛被使用的一些語法。

比較運算符

比較運算符我們在前面的內容中已經有介紹一部份，還有一些相等比較的概念。以下將它大略分成2個分類來說明:

- 相等比較運算: 相等(Equality)(==)，完全一致(Identity)(===)。不相等(Inequality)(!=)，不完全一致(Non-identity)(!==)。
- 關係比較運算: 大於(>)，小於(<)，大於等於(>=)，小於等於(<=)

相等比較運算時，需要遵守標準中所定義的"抽象相等比較演算法"((==)與(!=)符號的比較)與"嚴格相等比較演算法"((===)與(!==)符號的比較)的規定。抽象相等比較演算法((==)與(!=)符號的比較)的規則如下(假設x是左邊的運算子，而y是右邊的運算子):

- Type(x)與Type(y)的類型相同時，則進行嚴格相等比較
- 數字與字串比較時，字串會自動轉成數字再進行比較
- 其中有布林值時， `true` 轉為1， `false` 轉為+0
- 當物件與數字或字串比較時，物件會先轉為預設值以及轉成原始資料類型的值，再作比較

至於嚴格相等比較演算法(也就是(===)與(!==)符號的比較)，由於一定要比較原始資料類型的類型，只要類型不同就一定是回傳為 `false` 。除了類型相同，值也要相等，才會有回傳 `true` 的情況，什麼值轉換成什麼值再比較就根本不需要。

> 註: 物件的情況會比較特別，我們會在說明物件類型的章節再詳細說明。

你如果之前已經有"falsy"與"truthy"的概念，就會理解到很多比較運算的最終結果，都會以這個為基礎轉變為布林的 `false` 或 `true` 值。

在 `if` 中的判斷情況表達式中可以直接運算出布林值，這就是依照"falsy"與"truthy"的概念。或是再搭配 邏輯反相Logical NOT符號(!)來回傳反轉的布林值，例如:

```
if (undefined) console.log('true') //false

if (null) console.log('true') //false

if (+0) console.log('true') //false

if (!'') console.log('!\'\' true') //true

if (123) console.log('123 true') //true

//下面的還沒教到，是空物件與空陣列
const a = {}
const b = []

if (a) console.log(a, 'true')
if (b) console.log(b, 'true')
```

邏輯運算符

邏輯運算符實際上在前面的課程都已經介紹過了，只有三個而已:

- 邏輯與Logical AND(&&)
- 邏輯或Logical OR(||)
- 邏輯反相Logical NOT(!)

邏輯與Logical AND(&&) 與 邏輯或Logical OR(||) 兩個符號可以組合多個不同的比較運算，然後以邏輯運算的"與"與"或"來作最後的布林值的運算。不過要注意的是，它們的運算回傳值，在JavaScript中並非布林值，而是**最終的值**，轉換為布林值是判斷情況中的作用。

> 註: 邏輯與Logical AND(&&)的結果只有同時為真時才能為真。也就是"真&&真 為 真"，其他都為"假"。 註: 邏輯或Logical OR(||)的結果只要其一為真，結果就為真。

這兩個運算符通常會搭配 群組運算符(())，也就是括號標記(())，這在數學運算上是用來作為優先運算的區分，或是用來區隔不同的比較運算。以下為範例:

```
const x = 50
const y = 60
const z = 100

if ((x > 10) && (x < 100)) console.log(true)
if (( ((x + y) > 10) || (x === 50) ) && (z == 100)) console.log(true)
```

> 註: (&)符號英文為And或Amphersand。(|)符號的英文在電腦上通常稱為Pipe，管道的意思。(!)在英文中稱為Exclamation mark，驚嘆號的意思。

風格指引

- (Airbnb 15.1) 優先使用(===)與(!==)而非(==)與(!=)
- (Airbnb 15.2) 像if的條件語句內會使用ToBoolean的抽象方法強制轉為布林類型，遵循以下規範：
  - 物件 轉換為 true
  - Undefined 轉換為 false
  - Null 轉換為 false
  - 布林 轉換為 該布林值
  - 數字 如果是 +0, -0, 或 NaN 則轉換為 false，其他的皆為 true
  - 字串 如果是空字串 '' 則轉換為 false，其他的皆為 true
- (Airbnb 15.3) 使用簡短寫法（註: 範例中這兩種判斷情況的簡短寫法很常見)

```
// 壞寫法
if (name !== '') {
  // ...stuff...
}

// 好的寫法
if (name) {
  // ...stuff...
}

// 壞寫法
if (collection.length > 0) {
  // ...stuff...
}

// 好的寫法
if (collection.length) {
  // ...stuff...
}
```

- (Airbnb 15.6) 不應該使用巢狀的三元運算語句，一般都只會用單行的表達式。
- (Airbnb 15.7) 避免不必要的三元運算語句。

```
// 壞寫法
var isYes = answer === 1 ? true : false

// 好寫法
var isYes = answer === 1

// 壞寫法
var isNo = answer === 1 ? false : true

// 好寫法
var isNo = answer !== 1

// 壞寫法
var foo = bar ? bar : 1

// 好寫法
var foo = bar || 1
```

- (Airbnb 16.1) 具有多行語句的區塊，需要使用花括號(braces)({})框起來

```
// 壞寫法
if (test)
  return false

// 好寫法
if (test) return false

// 好寫法
if (test) {
  return false
}

// 壞寫法
function foo() { return false }

// 好寫法
function bar() {
  return false
}
```

- (Airbnb 16.2) 如果你在多行區塊時使用if與else，將else放在if區塊的結尾花括號(})後的同一行。

```
// 壞寫法
if (test) {
  thing1()
  thing2()
}
else {
  thing3()
}

// 好寫法
if (test) {
  thing1()
  thing2()
} else {
  thing3()
}
```

switch語句

switch/斯餵取/語句有一種講法與用法，是相當於多個 `if...else` 的組合語句，也就是用於判斷情況有很多種不同的回傳值的組合情況，不過這個理解是有些問題。實際上，它最常被使用的是在完全一致相等值(===)比較的情況下，很少用在相關比較運算的情況下。基本上它的語法結構如下:

```
switch (expression) {
  case value1:
    //符合運算得到value1的執行語句
    break
  case value2:
    //符合運算得到value2的執行語句
    break
  ...
  case valueN:
    //符合運算得到valueN的執行語句
    break
  default:
    //符合運算得到其他值的執行語句
    break
}
```

舉一個簡單的範例來說，像下面這樣的 `if...else` 語句:

```
const x = 10

if (x > 100){
  console.log('x > 100')
```

```
} else if (x < 100 && x >50) {
  console.log('x < 100 && x >50')
} else {
  console.log('x < 50')
}
```

轉換為switch語句會變成像下面這樣。 `switch(true)` 代表當 `case` /K斯/中的比較運算需要為布林值的 `true` 時，才能滿足而執行其中包含的語句:

```
const x = 10

switch (true) {
    case (x >100):
        console.log('x > 100')
        break
    case (x < 100 && x >50):
        console.log('x < 100 && x >50')
        break
    default:
        console.log('x < 50')
        break
}
```

這相當於下面這個if...else的語句:

```
const x = 10

if ((x > 100) === true) {
    console.log('x > 100')
}else if ((x < 100 && x >50) === true) {
    console.log('x < 100 && x >50')
} else {
    console.log('x < 50')
}
```

而 `switch` 語句最常被使用的情況，是用於判斷相等值的情況下，在這時候 `case` 語句裡的比較結果的相等於完全一致(Identity)(===)的比較結果，在滿足的情況下才會執行包含在 `case` 其中的執行語句，例如:

```
const x = 10

switch (x) {
    case 100 :
        console.log('x is 100')
        break
    case 50:
        console.log('x is 50')
        break
    case 10:
        console.log('x is 10')
        break
    default:
        console.log(x)
        break
}
```

> 注意: 因為case語句中的值會以一致相等運算(===)來比較，所以資料類型也要完全相等才行，上面的範例如果把 `case 10:` 改為 `case '10':` ，就會輸出 `10` ，而不是 `x is 10`

`break` /博累克/關鍵字雖然在語法上是可選的，但 `case` 語句沒有搭配 `break` 會一直執行到出現 `break` 或是 `switch` 整個語句的結尾，這會出現不正確的結果，像下面這個範例:

```
const x = 50

switch (x) {
    case 100 :
        console.log('x is 100')
```

```
        break
    case 50:
        console.log('x is 50')
        //這邊少一個break
    case 10:
        console.log('x is 10')
        break
    default:
        console.log(x)
        break
}
```

最後的結果會是像下面這樣，通常這個結果並不是我們想要的:

```
x is 50
x is 10
```

> 註: 這算是一個有陷阱的設計，在eslint的no-fallthrough規則頁面上可以看到更多的說明

> 註: 所以不管如何，只要case語句中有包含其它需要執行的語句，一定需要以break作為結尾。但最後一個case或default語句可以不需要break。

判斷時有多個 `case` 情況而執行同一個語句時，會使用像下面這個範例的語法，這個語法結構也是很常見的用法，例如:

```
const fruit = '芒果'

switch (fruit)
{
   case '芭樂':
   case '香蕉':
       console.log(fruit, '是四季都出產的水果')
       break
   case '西瓜':
   case '荔枝':
   case '芒果':
   default:
       console.log(fruit, '是只有夏季出產的水果')
       break
}
```

風格指引

- (Airbnb 15.5)當case與default區塊中包含了字面宣告時（例如：let、const、function 及 class）時使用花括號({})來建立區塊語句。(註: 這是為了要區隔出每個case中的各自字面宣告區塊，以免造成重覆宣告的錯誤)

```
// 壞寫法
  switch (foo) {
    case 1:
      let x = 1
      break
    case 2:
      const y = 2
      break
    case 3:
      function f() {}
      break
    default:
      class C {}
  }

  // 好寫法
  switch (foo) {
    case 1: {
      let x = 1
      break
    }
    case 2: {
      const y = 2
```

```
      break
    }
    case 3: {
      function f() {}
      break
    }
    case 4:
      bar()
      break
    default: {
      class C {}
    }
  }
```

- (Airbnb 18.3) 在控制語句(if、while等等)的開頭花括號符號({)前面多空一格空白，函式的呼叫或定義則不需要。
- `defalt` 語句習慣固定是放在 `switch` 語句中的最後一段的位置，雖然它也不是一定要放在那裡。
- `switch` 語句的 `case/default` 語句，假使是位於最後一段(通常是 `default` 語句)，它的 `break` 在功能上並非必要，習慣加上只是為了讓程式碼更具一致性。

英文解說

Expression名詞，有"表情"、"表達"的意思，表情通常指的是人臉部的表情，這個字詞常常用在專業技術性的呈現或表達。至於像常會用的"表情符號"並不是這個字詞，英文字詞是使用emoticon或emoji。emoticon是emotion加上icon的新字詞，emotion則有"情感"、"情緒"的意思，近似於feeling(感受)的意思。Expression Emoticon這個組合字詞，通常是指"和人臉部表情有關的表情符號"，哭哭笑笑一類。不過表情符號的範圍比較廣，而且有很多是和動作、物品或標識有關。

Statement名詞，有"聲明"、"語句"、"陳述"等意思，它也可以當作財務或商業上的"報表"、"結算表"來使用，看起來這個字詞是用在正經八百的文件內容或訊息上。另外，它的字根是state，這個字的動詞有"宣佈"、"聲明"、"規定"的意思，名詞還有"州"或"國家"的意思，在電腦中最常拿來作"狀態"的名詞使用。

Literal名詞，有"文字"、"字面"的意思，也就是"按照字面上的意思就是這樣"。在電腦專業領域通常用這個字詞來作為"固定值"(fixed value)的記號，例如String literal稱為字串字面量，Numeric literal稱為數字字面量，其他還有陣列、物件、函式的字面量。而相對於Literal(字面)的就是 變數/常數 記號。

結語

本章一開始說明了兩個重要的名詞定義，表達式(Expression)與語句(Statement)。前面所學的變數/常數，以及資料類型的部份，將組合成為完整的程式碼語句的一部份。本章的重點是在於控制流程語句 `if...else` ，以及 `switch` 語句的應用，以及判斷情況的概念。

另外，在ES6後還有另一個可以用在控制流程的結構 - Promise(承諾)，由於它需要更多基礎的知識，我們將會在"特性"單元中，用獨立的一個章節來說明它。

家庭作業

作業一

公司交給你一個案子，要撰寫一支對網站上線上填寫表單進行檢查欄位的程式，以下是這個表單的欄位與要檢查的規則:

- 姓名(fullname): 最多4個字元，必填
- 手機號碼(mobile): 手機號碼必需是10個數字
- 出生年月日(birthday): 年1970-2000，月份1-12，日期1-31
- 住址(address): 最少8個字元，最多50個字元，必填
- Email(email): 最少10個字元，最多50個字元，必填

請問要如何寫出每個欄位的判斷檢查的程式碼。

函式與作用域

函式(function)

函式(function/訪遜/)是JavaScript的非常重要的特性。函式用於程式碼的重覆使用、資訊的隱藏與複合(composition)。我們經常會把一整組的功能程式碼，寫成一個函式，之後可以重覆再使用，JavaScript在執行時的呼叫堆疊也是以函式作為單位。

> 注意: 依據ECMAScript標準的定義，函式的 `typeof` 回傳值是'function'，而不是'object'。由此可見在標準中定義的'object'類型，只是針對"單純"的物件定義而言，但具有函式呼叫(call)實作的物件，將會歸類為'function'，因為它們的內部都有建構函式的特性，例如Date、String、Object這些內建物件，它們的 `typeof` 回傳值都是'function'。 `typeof` 的回傳值只能作為參考用，有很多複雜的應用下並沒有辦法辨別得出是什麼樣的物件。

> 註: 那麼要如何精確又有效的檢查一個變數/常數是否為函式類型？請參考這篇How can I check if a javascript variable is function type?的問答。

函式定義

函式的基本語法結構如下:

```
//匿名函式
function() {}

//有名稱的函式
function foo() {}
```

函式的名稱也是一個識別符，命名方式如同變數/常數的命名規則。

而匿名函式並沒有函式的名稱，通常用來當作一個指定值，指定給一個變數/常數，被指定後這個變數/常數名稱，就成了這個函式的名稱。實際上，匿名函式也有其他的用法，例如拿來當作其他函式的傳入參數值，或是進行一次性執行。

函式使用 `return` 作為最後的回傳值輸出，函式通常會有回傳值，但並非每種函式都需要回傳值，也有可能利用輸出的方式來輸出結果。以下兩種方式對於函式都是可以使用的宣告(定義)方式，使用帶有名稱的函式稱為"函式定義"的方式，而另一種用變數/常數指定匿名函式的稱為"函式表達式"的方式:

```
//函式定義 - 使用有名稱的函式
function sum(a, b){
    return a+b
}

//函式表達式 - 常數指定為匿名函式
const sum = function(a, b) {
    return a+b
}
```

函式的呼叫是使用函式名稱加上括號(())，以及在括號中傳入對應的參數值，即可呼叫這個函式執行。例如:

```
const newValue = sum(100,0)
console.log(sum(99, 1))
```

ES6中有一種新式的函式語法，稱為"箭頭函式(Arrow Function)"，使用肥箭頭符號(Flat Arrow)(=>)，它是一種匿名函式的縮短寫法，下面這個寫法相當於上面的 `sum` 函式定義:

```
const sum = (a, b) =>  a + b
```

箭頭函式(Arrow Function)因為語法簡單，而且可以綁定 `this` 變數，所以算得上最受歡迎的ES6新功能，現在在很多程式碼中被大量使用。我們在特性篇中裡會專門有一章的內容來說明箭頭函式(Arrow Function)。

> 註: `this` 變數與物件有關，在物件的章節會說明。

> 註: 箭頭函式在"特性篇"有一篇專門的教學文章

傳入參數

函式的傳入參數是需要討論的，它是函式與外部環境溝通的管道，也就是輸入資料的部份。

不過，你可能會在函式的"傳入參數"常會看到兩個不同的英文字詞，一個是parameter(或簡寫為param)，另一個是argument，這常常會造成混淆，它們的差異在於:

- parameters: 指的是在函式的那些傳入參數名稱的定義。我們在文章中會以"傳入參數定義名稱"來說明。
- arguments: 指的是當函式被呼叫時，傳入到函式中真正的那些值。我們在文章中會以"實際傳入參數值"來說明。

傳入參數預設值

關於函式的傳入參數預設值，在未指定實際的傳入值時，一定是 `undefined` ，這與變數宣告後但沒有指定值很相似。而且在函式定義時，我們並沒有辦法直接限定傳入參數的資料類型，所以在函式內的語句部份，一開始都會進行實際傳入參數值的資料類型檢查。

有幾種方式可以用來在函式內的語句中，進行預設值的設定，例如用 `typeof` 的回傳值來判斷是否為 `undefined` ，或是用邏輯或(||)運算符的預設值設定方式。用來指定預設值的範例:

```
//用邏輯或(||)
const link = function (point, url) {
    let point = point || 10
    let url = url || 'http://google.com'
    ...
}

//另一種設定的方式，typeof是回傳類型的字串值
const link = function (point, url) {
    let point = typeof point !== 'undefined' ? point : 10
    let url = typeof url !== 'undefined' ? url : 'http://google.com'
    ...
}
```

> 注意: 邏輯或(||)運算符設定預設值，雖然語法簡單，但有可能不精準。它會在實際傳入參數值只要是"falsy"，就直接指定為預設值。

在ES6中加入了函式傳入參數的預設值指定語法，現在可以直接在傳入參數時就定義這些參數的預設值，這個作法是建議的用法:

```
const link = function (point = 10, url = 'http://google.com') {
    ...
}
```

> 註: 只有 `undefined` 的情況下才會觸發預設值的指定值。

> 註: 有預設值的參數在習慣上都是擺在傳入參數列表的"後面"，雖然這並不是個強制的作法只是個習慣。

以函式作為傳入參數

前面有說明函式可以作為變數/常數的指定值。不僅如此，在JavaScript中函式也可以當作實際傳入參數的值，將一個函式傳入到另一個函式中作為參數值，而且在函式的最後也可以回傳函式。這種函式的結構稱之為"高階函式(Higher-order function)"，是一種JavaScript程式語言的獨特特性，高階函式可以讓在函式的定義與使用上能有更多的彈性，它也延申出很多不同的應用結構。你可能常聽到JavaScript的callback(回呼、回調)結構，它就是高階函式的應用。

習慣上，因為函式在定義時，它的傳入參數並沒辦法限定資料類型，所以當要定義傳入參數將會是個函式時，通常會用fn或func作為傳入參數名稱，以此作為辨別。不過，當你在撰寫一個函式時，最好是要加上傳入值的，以及回傳值的註解說明。以下為一個簡單的範例:

```
const addOne = function(value){
    return value + 1
}

const addOneAndTwo = function(value, fn){
    return fn(value) + 2
}
```

```
console.log(addOneAndTwo(10, addOne)) //13
```

無名的傳入參數(unnamed arguments)

這是函式的一種隱藏機制，實際上對於傳入的參數值是有一個隱藏在背後的物件，名稱為 `arguments` ，它會對傳入參數實際值進行保存，可以直接在函式內的語句中直接取用。 `arguments` 雖是一個物件資料類型，但它有"陣列"的一些基本特性，不過缺少大部份陣列中的方法，所以被稱作"pseudo-array"(偽陣列)。以下為一個簡單的範例:

```
function sum() {
  return arguments[0]+arguments[1]
}

console.log(sum(1, 100))
```

不過，如果你在函式的傳入參數定義名稱中，使用了 `arguments` 這個參數名稱，或是在函式中的語句裡，定義了一個名稱為 `arguments` 的自訂變數/常數名稱，這個隱藏的物件就會被覆蓋掉。總之它的行為相當怪異，有一說是說它一開始設計時就錯了，隱藏的 `arguments` 物件對開發者來說，並不像隱藏版的密技，而是比較像是隱藏版的陷阱。

> 註: 關於arguments的詳細介紹，可以參考The JavaScript arguments object…and beyond

不固定傳入參數(Variadic)與其餘(rest)參數

像下面這個範例中，原先sum函式中，定義了要有三個傳入參數，但如果真正在呼叫函式時傳入的參數值(arguments)並沒有的情況下，或是多出來的時候，會發生什麼情況？

前面有說到，沒有預設值的時候會視為 `undefined` 值，而多出來的情況，是會被直接略過。有的時候需要一種能夠"不固定傳入參數"的機制，在各種函式應用時，才能比較方便。

```
function sum(x, y, z) {
  return x+y+z
}

console.log(sum(1, 2, 3))  //6
console.log(sum(1, 2))    //NaN
console.log(sum(1, 2, 3, 4))  //6
console.log(sum('1', '2', '3'))  //123
console.log(sum('1', '2')) //12undefined
console.log(sum('1', '2', '3', '4'))  //123
```

雖然上一節有說過的，有個隱藏的 `arguments` 物件，它可以獲取到所有傳入的參數值，然後用類似陣列的方式來使用，但它的設計相當怪異(有一說是設計錯誤)，使用時要注意很多例外的情況，加上使用前根本也不需要定義，很容易造成程式碼閱讀上的困難。另外，在一些測試報告中，使用 `arguments` 物件本身比有直接使用具有名稱傳入參數慢了數倍。所以結論是， `arguments` 物件的是不建議使用它的。

那麼，有沒有其他的機制可以讓程式設計師能處理不固定的傳入參數？

在ES6中加入了其餘參數(rest parameters)的新作法，它使用省略符號(ellipsis)(...)加在傳入參數名稱前面，其餘參數的傳入值是一個標準的陣列值，以下是一個範例:

```
function sum(...value) {
    let total = 0
    for (let i = 0 ; i< value.length; i++){
        total += value[i]
    }
    return total
}

console.log(sum(1, 2, 3))  //6
console.log(sum(1, 2))    //3
console.log(sum(1, 2, 3, 4))  //10
console.log(sum('1', '2', '3'))  //123
console.log(sum('1', '2')) //12
console.log(sum('1', '2', '3', '4'))  //1234
```

如果要寫得更漂亮、簡潔的的語法，直接使用Array(陣列)本身的好用方法，像下面這樣把原本的範例重寫一下:

```
function sum(...value) {
    return value.reduce((prev, curr) => prev + curr )
}
```

> 註: reduce(歸納)是陣列的方法之一，它可以用來作"累加"

其餘參數只是扮演好參數的角色，代表不確定的其他參數名稱，所以如果一個函式中的參數值有其他的確定傳入參數名稱，其餘參數名稱應該要寫在最後一個位子，而且一個函式只能有一個其餘參數名稱:

```
function(a, b, ...theArgs) {
  // ...
}
```

其餘參數與 `arguments` 物件的幾個簡單比較:

- 其餘參數只是代表其餘的傳入參數值，而 `arguments` 物件是代表所有傳入的參數值
- 其餘參數的傳入值是一個標準陣列，可以使用所有的陣列方法。而 `arguments` 物件是"偽"陣列的物件類型，不能使用陣列的大部份內建方法
- 其餘參數需要定義才能使用， `arguments` 物件不需要定義即可使用，它是隱藏機制

內部(巢狀)函式

函式中的語句中，還可以包含其他的函式，這稱為內部函式(inner)，或是巢狀函式(nested)的結構。以下為一個簡單的範例:

```
function addOuter(a, b) {

    function addInner() {
        return a + b
    }

    return addInner()
}

addOuter(1, 2) //3
```

這樣的程式碼有點類似下面的寫法，不過你可以仔細的比較一下這兩個程式碼之間，傳入參數值的差異:

```
function addOuter(a, b) {
    return addInner(a, b)
}

function addInner(a, b) {
    return a + b
}

addOuter(1, 2) //3
```

內部函式可以獲取到外部函式所包含的環境值，例如外部函式的傳入參數、宣告變數等等。而內部函式又可以成為外部函式的回傳值，所以當內部函式接收到外部函式的環境值，又被回傳出去，內部函式間接變成一種可以讓函式對外曝露包含在函式內部環境值的溝通管道，這種結構稱之為"閉包(closure)"。

內部函式在JavaScript中被廣泛的使用，因為它可以形成所謂"閉包"(closure)的結構。

> 註: "閉包"(closure)在特性篇中有一個獨立的章節來說明。

作用範圍(scope)

"作用範圍(scope)"或稱之為"作用域"，指的是"變數或常數的定義與語句的可見(被存取得到)的範圍"，作用範圍可簡單區分為本地端的(local)與全域的(global)。

JavaScript程式語言的作用範圍，基本上是使用"函式作用範圍(function scope)"的，或稱之以函式為基礎(function-based)的作用範圍。也就是說只有使用函式才能劃出一個本地端的作用範圍，其他的區塊像if、for、switch、while等等，雖然有使用區塊語句({...})，但卻是無法界定出作用範圍。

因此，當你在所有函式的外面宣告變數或常數，這個變數或常數就會變成"全域的"作用範圍的一員，稱為"全域變數或常數"，也就是說在程式碼裡的任何地方都可以被存取得到。

> 註: 在JavaScript語言中，你應該要把"函式"也當作一種"值"來看待，它可以被指定到一個變數/常數，也可以作為函式回傳值。而在作用範圍中的行為也和"值"類似。

在ES6後的新作法，加入"區塊作用範圍(block scope)"概念，也就是使用具有區塊的語句，例如上述的if、for、switch、while等等，都可以劃分出作用範圍，這是一個很棒的改進。那要怎麼作呢？就是以 `let` 與 `const` 取代原本 `var` 的宣告變數的方式， `var` 可以完全捨棄不使用。

如果是使用 `var` 來定義變數，程式碼中的變數x並不是在函式中定義的，所以會變為"全域變數"：

```
if (true) {
  var x = 5
}

console.log(x) //5
```

對比使用 `let` 來宣告變數，程式碼中的y位於區塊中，無法在外部環境獲取得到：

```
if (true) {
  let y = 5
}

console.log(y) //y is not defined
```

其他函式特性

回調(callback)

回調(callback)是一種特別的函式結構，也因為JavaScript具有"高階函式(Higher-order function)"的特性，意思是說在函式中可以用另一個函式當作傳入參數值，最後也可以回傳函式。

一般而言，函式使用回傳值(return)作為最後的執行語句。但回調並不是，回調結構首先會定義一個函式類型的傳入參數，在此函式的最後執行語句，即是呼叫這個函式傳入參數，這個函式傳入參數，通常我們稱它為回調(callback)函式。回調函式經常使用匿名函式的語法，直接寫在函式的傳入參數中。

```
function showMessage(greeting, name, callback) {
    console.log('you call showMessage')
    callback(greeting, name)
}

showMessage('Hello!', 'Eddy', function(param1, param2) {
    console.log(param1 + ' ' + param2)
})
```

由於回調函式是一個函式類型，通常會在使用它的函式中，作一個基本檢查，以免造成程式錯誤，本章節的最前面有說明過了，函式的 `typeof` 回傳值是'function'，以下為改寫過的程式碼，改寫過後不論是不是有傳入回調函式，都可以正常運作，也就是回調函式變成是一個選項:

```
function showMessage(greeting, name, callback) {
     console.log('you call showMessage')

    if (callback && typeof(callback) === 'function') {
        callback(greeting, name)
    }
}
```

回調(callback)提供了使用此函式的開發者一種彈性的機制，讓程式開發者可以自行定義在此函式的最後完成時，要如何進行下一步，這通常是在具有執行流程的數個函式的組合情況。實際上，回調函式實現了JavaScript中非同步(asynchronously)的執行流程，這使得原本只能從頭到底(top-to-bottom)的程式碼，可以在同時間執行多次與多種不同的程式碼。在實際應用情況時，回調結構在JavaScript程式中大量的被使用，它也變成一種很明顯的特色，例如以下的應用中很常見:

- HTML中的DOM事件
- AJAX
- 動畫
- Node.js

提升(Hoisting)

簡單的來說，提升是JavaScript語言中的一種執行階段時的特性，也是一種隱性機制。不過，沒先定義與指定值就使用，這絕對是個壞習慣是吧？變數/常數沒指定好就使用，結果一定是不是你要的。

`var` 、 `let` 和 `const` 會被提升其定義，但指定的值不會一併提升上去，像下面這樣的程式碼:

```
console.log(x) //undefined
var x = 5

console.log(y) //undefined
let y = 5
```

最後的結果出乎意料，竟然只是沒指定值的 `undefined` ，而不是程式錯誤。實際上這程式碼裡的變數被提升(Hoisting)了，相當於:

```
var x
console.log(x)
x = 5

let y
console.log(y)
y = 5
```

函式定義也會被提升，而且它變成可以先呼叫再定義，也就是整個函式定義內容都會被提升到程式碼最前面。不過這對程式設計師來說是合理的，在很多程式語言中都可以這樣作:

```
foo() //可執行

function foo(){
  console.log('Hello')
}
```

不過使用匿名函式的指定值方式(函式表達式, FE)，就不會有整個函式定義都被提升的情況，只有變數名稱被提升，這與上面的變數宣告方式的結果一致:

```
foo() //錯誤: foo is not a function

let foo = function(){
  console.log('Hello')
}
```

結論如下:

- 所有的定義(var, let, const, function, function*, class)都會被提升
- 使用函式定義時，在函式區塊中的這些定義也會被提升到該區塊的最前面
- 當函式與變數/常數同名稱而提升時，函式的優先程度高於變數/常數。
- 遵守好的風格習慣可以避免掉變數提升的問題

全域作用範圍污染

全域作用範圍污染(global scope pollution)，整體來說，也是壞的程式特性+壞的程式寫作習慣，所造成的不良後果。例如沒有經過 var 宣告的變數，會自動變為全域作用範圍，或是是在把變數宣告在全域作用範圍中。過多的變數常常會造成記憶體無法回收，或是全域變數與函式中的變數常常互相衝突。

全域作用範圍污染在JavaScript中，一直是一個長久以來經常會發生的問題，尤其是在程式愈來愈龐大，整體的組織與結構沒有一開始就預作規劃時。ES6中針對作用範圍作了很多的標準上的改進，但不論程式特性如何進步，維護一個良好的寫作習慣，可能比程式本身的功能還重要幾百倍。

所以，好的程式設計師，在撰寫程式時應遵守一些建議的風格習慣，好好地理解作用範圍的概念，這樣就可以避免全域作用範圍污染情況發生。

匿名函式與IIFE

匿名函式還有另一個會被使用的情況，就是實現只執行一次的函式，也就是IIFE結構。IIFE是Immediately-invoked function expressions的縮寫，中文稱之為"立即呼叫的函式表達式"，IIFE可以說是JavaScript中獨特的一種設計模式，它是被某些聰明的程式設計師研究出來的一種結構，它與表達式的強制執行有關，有兩種語法長得很像但功能一樣，這兩種都有人在使用:

```
(function () { … })()
(function () { … }())
```

IIFE在執行環境一讀取到定義時，就會立即執行，而不像一般的函式需要呼叫才會執行，這也是它的名稱的由來 - 立即呼叫，唯一的例外當然是它如果在一個函式的內部中，那只有呼叫到那個函式才會執行。

```
(function(){
    console.log('IIFE test1')
}())

function test2(){
    (function(){
        console.log('IIFE test2')
    }())
}

test2()
```

IIFE的主要用途，例如分隔作用範圍，避免全域作用範圍的污染，避免在區塊中的變數提升(Hoisting)。IIFE也可以再進而形成一種Module Pattern(模組模式)，用來封裝物件的公用與私有成員。許多知名的函式庫例如jQuery、Underscore、Backbone一開始發展時，都使用模組模式來作為擴充結構。

不過，模組模式的結構存在於JavaScript已有很久一段時間，算是前一代主要的設計模式與程式碼組織方式，現在網路上看到的教學文，大概都是很有歷史的了。而現今的JavaScript已經都改為另一種更具彈性的、更全面化的稱為Module System(模組系統)的作法，例如AMD、CommonJS與Harmony(ES6標準的代號)，這在特性篇會有一個獨立的章節再作介紹。

純粹函式與副作用

在表達式的章節中，我們有講到在表達式中副作用(Side Effect)的分別，函式也有這種區分方式，不過對於函式來說，具有副作用代表著可能會更動到外部環境，或是更動到傳入的參數值。函式的區分是以 純粹(pure)函式 與 不純粹(impure)函式 兩者來區分，這不光只有無副作用的差異，還有其他的條件。

純粹函式(pure function)即滿足以下定義的函式:

- 給定相同的輸入(傳入值)，一定會回傳相同輸出值結果(回傳值)
- 不會產生副作用
- 不依賴任何外部的狀態

一個典型的純粹函式的例子如下:

```
const sum = function(value1, value2) {
  return value1 + value2;
}
```

套用上面說的定義，你可以用下面這樣理解它是不是一個純粹函式:

- 只要每次給定相同的輸入值(例如1與2)，就一定會得到相同的輸出值(例如3)
- 不會改變原始輸入參數，或是外部的環境，所以沒有副作用
- 不依賴其他外部的狀態(變數之類的)

那什麼又是一個不純粹的函式？看以下的範例就是，它需要依賴外部的狀態值(變數值):

```
let count = 1;

let increaseAge = function(value) {
  return count += value;
}
```

在JavaScript中不純粹函式很常見，像我們一直用來作為輸出的 `console.log` 函式，或是你可能會在很多範例程式看到的 `alert` 函式，都是不純粹函式，這類函式因為沒有回傳值，都是用來作某件事而已。像 `console.log` 會更動瀏覽器的主控台(外部環境)的輸出，也算是一種副作用。

純粹函式具有以下的優點:

- 程式碼可以簡單化，閱讀性提高
- 較為封閉與固定，可重覆使用性高
- 易於單元測試(Unit Test)、除錯

不過有許多內建的或常用的函式都是免不了有副作用的，例如這些應用:

- 會改變傳入參數變數(物件、陣列)的函式
- 時間性質的函式
- I/O相關
- 資料庫相關
- AJAX

例如而每次輸出值都不同的不純粹函式一類，最典型的就是 `Math.random` ，這是產生隨機值的內建函式，既然是隨機值當然每次執行的回傳值都不一樣。

總之，並不是說有副作用的函式就不要使用，而且要理解這個概念，然後儘可能在你自己的寫的函式上使用純粹函式，以及讓必要有副作用的函式得到管理與控制。現在已經有一些新式的函式庫或框架，會特別要求在某些地方只能使用純粹函式，而具有副作用的不純粹函式只能在特定的情況下才能使用，這就需要先有這樣的概念與特別注意了。

風格指引

- (Airbnb 7.3、Google) 永遠不要在非函式區塊(if, while等等)裡面，定義一個函式。而是把函式指定給一個變數(註: 函式表達式)
- (Airbnb 18.3) 在控制語句(if, while等等)的圓括號()開頭加上一個空白字元。函式在呼叫或定義時，函式名稱與傳入參數列則不需要空白。
- (idiomatic.js 7.B) 提前回傳可以增加程式碼可閱讀性，對於效率沒有明顯差異。

```
// 不好的寫法:
function returnLate( foo ) {
  var ret;

  if ( foo ) {
    ret = "foo";
  } else {
    ret = "quux";
  }
  return ret;
}

// 好的寫法:

function returnEarly( foo ) {
```

```
  if ( foo ) {
    return "foo";
  }
  return "quux";
}
```

常見問答

函式要用那種定義方式比較好？

由上面的內容來說，有三種定義函式的方式，一種是傳統的函式定義(FD)，另一種是用常數或變數指定匿名函式的方式(函式表達式, FE)，最後一種是新式的箭頭函式，這三種的範例如下:

```
//函式定義
function sum(a, b){
  return a+b
}

//函式表達式
const sum = function(a, b) {
  return a+b
}

//箭頭函式
const sum = (a, b) => a+b
```

那麼，到底那一種是比較建議的方式？

首先，由於第二種方式(函式表達式)完全可以被箭頭函式取代，箭頭函式又有另外的好處(綁定 `this`)，所以它幾乎可以不用了。

而第一種方式(函式定義)有一些特點，所以它會被用以下的情況:

- 全域作用範圍
- 模組作用範圍
- Object.prototype的屬性值

函式定義的優點:

- 函式定義名稱可以加入到執行期間的呼叫堆疊(call stack)中，除錯方便
- 函式定義可以被提升，也就是可以在定義前呼叫(請參考上面的說明內容)

除此之外，都使用第三種方式，也就是箭頭函式。

> 請參考When should I use Arrow functions in ECMAScript 6?

arguments物件還要用嗎？

當然是不要用。這東西是有設計缺陷的，除非你對它很了解，不然用了也可能會出問題，不過話說回來，如果你對它很了解的話，就不會想用它了。改用"其餘參數"就可以了。

IIFE語法結構還要用嗎？

看情況而定。如果是有一些函式庫例如jQuery中的擴充方式會用到，這當然避免不了。如果是其他的已經採用新式的模組系統，可能是根本不需要。

英文解說

Scope/史溝波/ 中文有"視野"、"導彈範圍"的意思。也就是相當於程式語言中，看不看得到(能不能存取得到)的意思。一般中文會翻譯成"作用域"或"作用範圍"，有點難懂的文言文。與作用範圍(scope)相關的還有一個Namespace(命名空間)，這是一種語法結構或組織方法，讓程式設計師可以把不同的識別符與程式碼敘述，放到不同的"空間"之中，以免造成衝突或混亂。不過，JavaScript語言中並沒有內建的命名空間(Namespace)的特性。

Context/康鐵斯/ ，中文有"上下文"、"環境"的意思。在程式語言中指的是程式碼的執行上下文內容，它與作用範圍相關，但不相等於作用範圍。這個名詞有時會和Scope一起拿出來比較，在JavaScript語言中它也是很重要的一個概念。我們在物件中將會再說明它的意思。以下是基本的比較:

- Scope是屬於以函式為基礎的(function-based)。而Context則是以物件為基礎的(object-based)。
- Scope指的是在程式碼函式中的變數的可使用範圍。而Context指的是 `this` ，也就是指向擁有(或執行)目前執行的程式碼的物件。

家庭作業