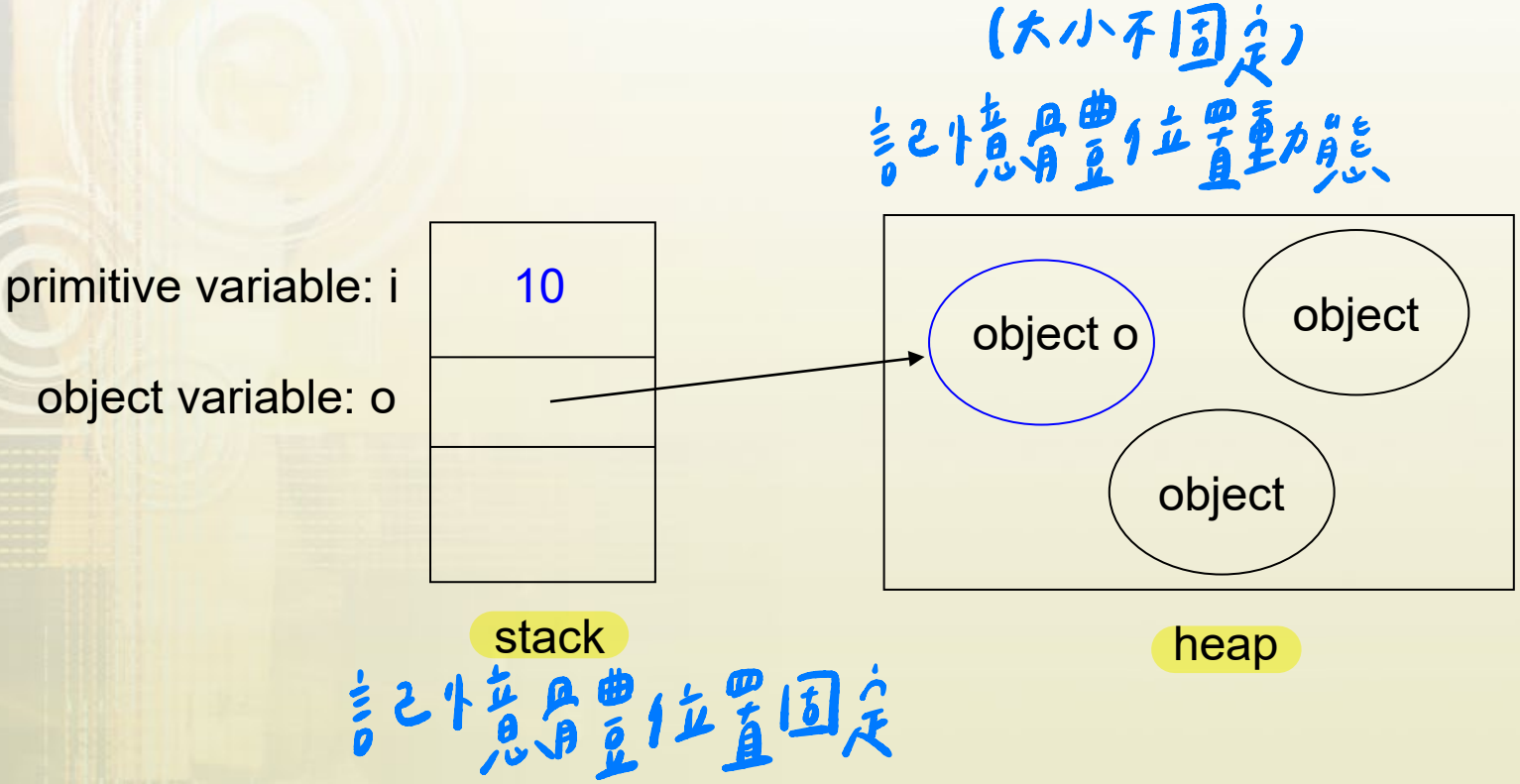
JavaScript Reference Types=物件

Primitive versus Reference Types

- primitive types 直接存
  - number, string, boolean
  - primitive values are stored on the stack memory
- reference types 有屬性、方法(是一件事物)
  - also called object type/definition
  - group of data and function definitions
  - each object has some Property and Method
    - e.g. document.write()
      document is a object, write() is the method of document
  - are stored on the heap memory

Primitive versus Reference Types

- when a object is assigned to a variable, only the memory address of object are stored to variable



What is Object

- An object is a collection of related data(property, member data) and/or functionality(method, member function) 把相關的東西打包：易於維護

Object	Properties	Methods
	car.name = Fiat	car.start()
	car.model = 500	car.drive()
	car.weight = 850kg	car.brake()
	car.color = white	car.stop()

source: https://www.w3schools.com/js/js_objects.asp

- accessed the object's properties and methods using dot notation.

Reference Types

- Primitive Wrapper Types
  - Number, Boolean, String
- Native Reference Types
  - Array Type
  - Date Type, Math Type, …
  - Function Type
  - Object Type
- Custom Types (User Defined Object)

物件 < 方法 屬性

function 是 reference type

是一種資料型態 (可當參數傳)

String

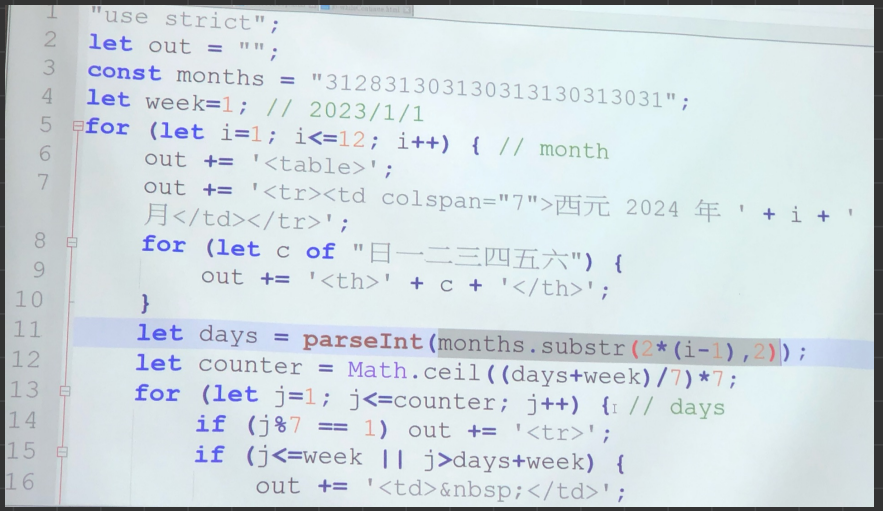
- e.g String.html 示範字串物件的各種方法與屬性
- note: substr(m, n) 和 substring(m, n) 的差異。如果 text = "我願是千萬條江河"，text.substr(3,5)會傳回 "千萬條江河"(第4個字元開始，取5個字元)，text.substring(3,5)會傳回 “千萬“(第4個字元開始，第5個字元結束)。
- 有關於字串的比對，只提到了 indexOf() 和 lastIndexOf() 兩個方法，事實上 JavaScript 對於字串的比對和代換有許多強大的功能，例如 search、match、replace 等函數，這些功能統稱「通用表示法」，後面會仔細介紹。

String Properties and Methods

性質或方法	說明
length	傳回字串的長度
toUpperCase()	換成大寫字母
toLowerCase()	換成小寫字母
concat()	字串並排（等效於使用加號）
charAt(n)	抽出第 n 個字元（n=0 代表第一個字元）
charCodeAt(n)	抽出第 n 個字元（n=0 代表第一個字元），並轉換成 Unicode
substr(m, n)	傳回一個字串，從位置 m 開始，且長度為 n
substring(m, n)	傳回一個字串，從位置 m 開始，結束於位置 n-1
indexOf(str)	尋找子字串 str 在原字串的第一次出現位置
lastIndexOf(str)	尋找子字串 str 在原字串的最後一次出現位置

String Properties and Methods

性質或方法	說明	等效的HTML標籤
big()	增大字串的字型	<big>...</big>
small()	減小字串的字型	...
blink()	閃爍字串（不適用於 IE 瀏覽器）	<blink>...</blink>
bold()	變粗體	...
italics()	變斜體	...
fixed()	變等寬字體	<tt>...</tt>
strike()	槓掉字串	<strike>...</strike>
sub()	變下標	_{...}
sup()	變上標	^{...}
fontcolor()	設定字串的顏色	...
fontsize()	設定字串的字型大小	...

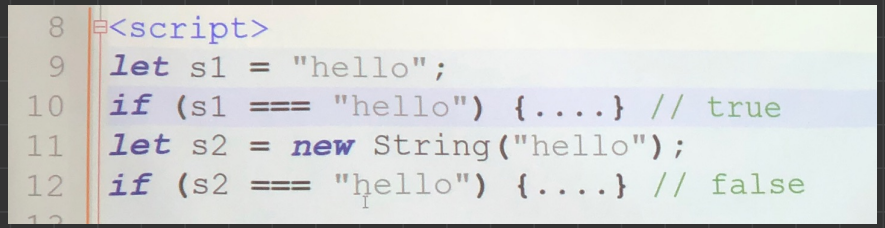


```
"use strict";
let out = "";
const months = "312831303130313130313031";
let week=1; // 2023/1/1
for (let i=1; i<=12; i++) { // month
    out += '<table>';
    out += '<tr><td colspan="7">西元 2024 年 ' + i + '
    月</td></tr>';
    for (let c of "日一二三四五六") {
        out += '<th>' + c + '</th>';
    }
    let days = parseInt(months.substr(2*(i-1),2));
    let counter = Math.ceil((days+week)/7)*7;
    for (let j=1; j<=counter; j++) { // days
        if (j%7 == 1) out += '<tr>';
        if (j<=week || j>days+week) {
            out += '<td> </td>';
```

mystring.bold 物件

mystring.small() 方法

是String的方法，屬性



```
<script>
let s1 = "hello";
if (s1 === "hello") {....} // true
let s2 = new String("hello");
if (s2 === "hello") {....} // false
```

物件本相比：會把其所有特性拿出來比

拿屬性來比較：實數比較

Why Primitive Wrapper Types?

隱藏在系統中

- Every time a primitive value is read, an object of the corresponding primitive wrapper type is created behind the scenes
- The Wrapper objects have useful properties and methods, e.g. toString().
- e.g.

```
let s = "hello";
console.log(s.toUpperCase()); // HELLO
```

本身不能用substring

系統偷偷產生 S String 物件，用完即丟

Number

- Returns different object values to their numbers. If the value cannot be converted to a legal number, NaN is returned. If no argument is provided, it returns 0.
  - Number(true) → 1, Number(false) → 0
  - Number(null) → 0
  - Number("011") → 11
  - Number("0xf") → 15
  - Number("string") → NaN
  - Number("") → 0
- e.g. Number.html

字串轉數值

Number methods

函式格式	說明
x.toString([radix])	將數值 x 轉成特定基底 radix 的字串。
x.toFixed(n)	將數值 x 轉成小數點以下 n 位有效數字的小數點表示法。
x.toExponential(n)	將數值 x 轉成小數點以下 n 位有效數字的科學記號表示法。
x.toPrecision(n)	將數值 x 轉成共具有 n 位有效數字。

Number Properties

- some useful properties

常數表示法	說明
Number.MIN_VALUE	傳回能在 JScript 中表示最接近零的數字。大約等於 5.00E-324。
Number.MAX_VALUE	傳回能在 JScript 中表示的最大值。大約等於 1.79E+308。
Number.NEGATIVE_INFINITY	傳回一個能在 JScript 中表示、且比最大負數 (-Number.MAX_VALUE) 還要小的值。
Number.POSITIVE_INFINITY	傳回能在 JScript 中表示且大於最大數 (Number.MAX_VALUE) 的值。
Number.NaN	一個特殊值，可指出算術運算式的傳回值不是一個數字。

Number toString() 範例

- e.g. NumberRadix.html 利用 toString() 函式，顯示不同基底的表示方式

```
document.writeln("<td>"+x.toString());
document.writeln("<td>"+x.toString(2));
document.writeln("<td>"+x.toString(8));
document.writeln("<td>"+x.toString(16));
```

  - 如果 toString() 沒有參數，預設以十進位為底

```
document.writeln("<td class=\"ccc\">"+x.toString(16).toUpperCase());
```

原本就是 string，可用 method

```
34 document.write("自訂物件：" + student +"<br>");
35 document.write("自訂物件：" + student.toString()+"<br>");
```

其實有偷偷放 toString

Note: toString() Example

- 利用 toString() 將各種物件轉成字串 toString.html

物件	toString() 的結果
Array（陣列）	將 Array 的元素轉換為字串，形成以逗號串連起來的結果，此結果 與 Array.toString() 和 Array.join() 得到的結果相同
Boolean（布林）	如果布林值為 True，會傳回 "true"；否則會傳回 "false"
Date（日期）	傳回顯示日期的文字形式
Error（錯誤）	傳回包含錯誤訊息的字串
Function（函數）	傳回函數的定義
Number（數字）	傳回數字的文字表示法
String（字串）	傳回 String 物件的值
自訂物件	傳回 "[object Object]"

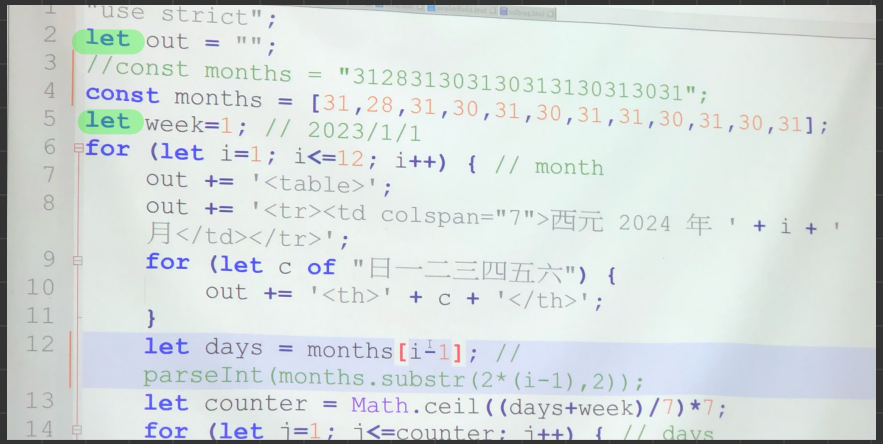
BigInt()

- store big integer values that are too big to be represented by a normal JavaScript Number.
- e.g.
  let a = 123456789012345678 9+1; // Number
  let b = 123456789012345678 9n+1n; // BigInt
  let c = BigInt("123456789012345678 9")+1n;
- .valueOf()
- BigInt.asIntN() static method truncates a BigInt value to the given number of least significant bits and returns that value as a signed integer.
- BigInt.asUintN() to an unsigned integer value

Reference Types - Array Type

Array Type

- arrays are ordered lists of data
- each item of array can hold any type of data
- first way to create array
  - e.g. var myArray = new Array(); // 產生一個空的陣列
  - e.g. var myArray = new Array(10);
    - create an array containing ten items each item gets undefined value
  - e.g. var myArray = new Array("item1", "item2", "item3");
  - the new operator can be omitted
- 2nd way to create array
  - var myArray = [];
  - var myArray = [,,,,,]
    - create 5 items in Firefox, 6 items in IE, not recommended
  - e.g. var myArray = ["item1", "item2", "item3"];



```
"use strict";
let out = "";
//const months = "312831303130313130313031";
const months = [31,28,31,30,31,30,31,31,30,31,30,31];
let week=1; // 2023/1/1
for (let i=1; i<=12; i++) { // month
    out += '<table>';
    out += '<tr><td colspan="7">西元 2024 年 ' + i + '
    月</td></tr>';
    for (let c of "日一二三四五六") {
        out += '<th>' + c + '</th>';
    }
    let days = months[i-1]; //
    parseInt(months.substr(2*(i-1),2));
    let counter = Math.ceil((days+week)/7)*7;
    for (let j=1; j<=counter; j++) { // days
```

2用陣列

2陣列

substr返回一字符串從指定位置開始到指定字符數字符

Array Index

- 要使用陣列變數時，需先宣告，但可以不用設定陣列的元素個數。我們可以使用索引 (Index) 來存取每一個元素的值，索引從 0 開始，例如陣列 myArray 的第一個元素為 myArray [0]，第五個元素為 myArray [4]，依此類推。
- e.g. array/arrayList.html

```
var myArray = new Array();
myArray[0] = "This is a test"; // 加入第 1 個元素
myArray[1] = 3.1415926; // 加入第 2 個元素
myArray[2] = "The last element"; // 加入第 3 個元素
```

```
    document.writeln("myArray[0] = " + myArray[0] + "<br>");
    document.writeln("myArray[1] = " + myArray[1] + "<br>");
    document.writeln("myArray[2] = " + myArray[2] + "<br>");

    myArray = ["資訊系", "電機系", "材料系"]; // same type
    // myArray = ["This is a test", 3.1415926, "The last element"]; // different types

    // 列舉陣列中的所有元素及其索引
    for (prop in myArray)
        document.write("<br>myArray[" + prop + "] = " + myArray[prop]);
    for (item of myArray)
        document.write("<br>"+item);
```

myArray[0] = 資訊系
myArray[1] = 電機系
myArray[2] = 材料系
資訊系
電機系
材料系

in/of差別

```
    var person = [];
    person["firstname"] = "john"; // person.firstname = 'john';
    person["lastname"] = "Doe";
    for (prop in person)
        document.write("<br>person[" + prop + "] = " + person[prop]);
```

index不可以是數字外的->自動轉換成物件

物件無法用of窮舉(只可用in)
陣列才可用of窮

for...in:除了承屬性的陣列,儘可能不用,否則會把除了承屬性一併車前出

Array Listing

- 列出陣列中元素

```
myArray = ["教務處", "學務處", "總務處"];
for (prop in myArray)
    console.log("myArray[" + prop + "] = " + myArray[prop]);
```

  - 陣列的索引就是它的屬性，所以 prop 依序為 0, 1, 2
  - 在取用陣列的元素時，還是必須使用 myArray[2] 或 myArray["2"] 等，而不能使用 myArray.2。
- [ES6] 或者改用 for/of 直接取用元素本身

```
for (item of myArray)
    console.log(item);
```

Array length property

- The length property sets or returns the number of elements in an array.
  - 也就是陣列的 length 屬性可讀也可寫
- 修改 length 屬性可以更改陣列大小
  - 減少 length 則 (new_length-1) 之後的元素將被刪除
  - 增加 length 則增加 (old_length-1) 之後的元素，型態為 undefined
  - e.g. arrayLength.html

用陣列方式存取物件屬性

不可用物件存取陣列

- There are two ways to access object properties: dot notation and bracket notation(array-like).
  - 使用 document.xxx 或 document[“xxx”] 來存取屬性 xxx，得到的結果是相同的。
  - (PS: xxx 必須是合法識別字)
- e.g. 列出所有 document 物件的屬性

```
for (prop in document) {
   document.write("<br>document."+prop+" = "+document[prop]);
}
```

物件後的屬性要是合法識別字

[note] arrays with named indexes

- JavaScript does not support arrays with named indexes.
- In JavaScript, arrays always use numbered indexes.
- If you use a named index, JavaScript will redefine the array to a standard object.

- [note] Arrays with named indexes are called associative arrays (or hashes).

Array methods

方法	說明
concat()	傳回一個由兩個或兩個以上陣列並排而成的新陣列
join()	傳回一個字串值，它是由陣列中的所有元素串連在一起所組成，並且用特定的分隔字元來分隔
pop()	移除陣列的最後一個元素，並將它傳回
push()	附加新元素到陣列尾部，並傳回陣列的新長度
reverse()	傳回一個元素位置反轉的陣列
shift()	移除陣列的第一個元素，並將它傳回
slice()	傳回陣列的一個區段
splice()	移除陣列中的元素，並依需要在原位插入新元素，然後傳回被刪除的元素
sort()	傳回一個元素已排序過的陣列
toString()	傳回一個物件（或陣列）的字串表示法
unshift()	在陣列開始處插入指定的元素，並傳回此陣列

Array toString(), join()

- 輸出陣列物件時，toString() 會將陣列轉換成由逗號隔開的字串，或使用 join() 來指定輸出的格式
- e.g. arrayJoin.html
  - 程式碼重點
    - document.writeln("myPet.toString()=“+myPet.toString()+"
");
    - document.writeln("myPet.join()="+myPet.join()+"
");
    - document.writeln("myPet.join(',')="+myPet.join(',')+"
");
    - document.writeln("myPet.join('+')="+myPet.join('+')+"
");
  - 說明
    - 從範例可以看出，myPet.toString() 和 myPet.join() 得到的結果是一樣的。

myPet = 鼠,牛,虎,兔
myPet.toString() = 鼠,牛,虎,兔
myPet.join() = 鼠,牛,虎,兔
myPet.join(', ') = 鼠, 牛, 虎, 兔
myPet.join(' + ') = 鼠 + 牛 + 虎 + 兔

Array split(), concat()

- 將字串拆成陣列：split()
- 陣列的串接：concat()
- e.g. arrayConcat.html
  - 程式碼重點

```
array1=str1.split('、'); // 將字串拆成陣列
array2=str2.split('、'); // 將字串拆成陣列
array3=array1.concat(array2); // 串接兩個陣列
```

- 說明
  - 字串物件由 split() 中的字串"、"分割成陣列。
  - Concat 可以使兩陣列結合在一起形成新的陣列

Array splice()

- 置換陣列中元素
- e.g. arraySplice.html
  - 程式碼重點

```
myPet = ["鼠", "牛", "虎", "兔", "龍", "蛇", "Cat",
"Bird", "狗", "豬"];
myPet.splice(6, 2, "馬", "羊", "猴", "雞");
```

  - 說明
    - splice(index,howmany,element1,...,elementX)
    - delete from index to index+howmany-1, then insert the element1, …, elementX

delete array element

- difference between using the delete operator and using the Array.prototype.splice method?
  - myArray = ['a', 'b', 'c', 'd']
    - myArray.splice(0, 2) // ["c", "d"]
    - delete myArray[0] // [undefined, "b", "c", "d"] or [empty, "b", "c", "d"] delete沒刪掉陣列
  - why deleting an element from an array (thus making it sparse) won't reduce its length.

Array sort(), reverse()

- 陣列中元素的排序，可使用陣列方法 sort() 和 reverse()
- e.g. arraySort1.html
- 說明
  - 要注意的是 sort() 會先將數值轉成字串，再進行字串的排序。
  - 若要進行數值的排序，必須自訂比較函數，並將此函數傳進 sort()，稍後說明。

排序後的陣列：
myArray[0] = 20
myArray[1] = 3.1415926
myArray[2] = 4
myArray[3] = First
myArray[4] = false
myArray[5] = third

字串一個一個字相比

Array pop(), push(), shift(), unshift()

- 陣列元素的運作：pop(), push(), shift(), unshift()
  - pop(), push() 類似堆疊，於陣列後方操作
  - shift(), unshift() 於陣列啟始處操作
- e.g. arrayPop.html
  - 程式碼重點

```
popped = myPet.pop();
elementCount = myPet.push("龍", "蛇")';
shifted = myPet.shift()";
myPet.unshift("馬", "羊")';
```

```
var myPet = ["鼠", "牛", "虎", "兔"];
document.writeln("myPet = " + myPet + "<br>");
// pop()
document.writeln("<p>After「popped = myPet.pop()」<br>");
popped = myPet.pop();
document.writeln("myPet = " + myPet + "<br>");
document.writeln("poped = " + popped + "<br>");
// push()
document.writeln('<p>After「elementCount = myPet.push("龍", "蛇")」<br>');
elementCount = myPet.push("龍", "蛇");
document.writeln("myPet = " + myPet + "<br>");
document.writeln("elementCount = " + elementCount + "<br>");
// shift()
document.writeln("<p>After「shifted = myPet.shift()」<br>");
shifted = myPet.shift();
document.writeln("myPet = " + myPet + "<br>");
document.writeln("shifted = " + shifted + "<br>");
// unshift()
document.writeln('<p>After「myPet.unshift("馬", "羊")」<br>');
myPet.unshift("馬", "羊");
document.writeln("myPet = " + myPet + "<br>");
```

myPet = 鼠,牛,虎,兔

After「popped = myPet.pop()」
myPet = 鼠,牛,虎
poped = 兔

After「elementCount = myPet.push("龍", "蛇")」
myPet = 鼠,牛,虎,龍,蛇
elementCount = 5

After「shifted = myPet.shift()」
myPet = 牛,虎,龍,蛇
shifted = 鼠

After「myPet.unshift("馬", "羊")」
myPet = 馬,羊,牛,虎,龍,蛇

Array Iteration Methods

call back func.

- Array.forEach()
- Array.map()
- Array.filter()
- Array.reduce(), Array.reduceRight()
- Array.every()
- Array.some()
- Array.indexOf(), Array.lastIndexOf()
- Array.find(), Array.findIndex()

- https://www.w3schools.com/js/js_array_iteration.asp

Reference Types - Math Type

數學物件

Math

- e.g. math.html 示範山數學物件的一些 properties and methods

方法	說明
abs(x)	取一個數 x 的絕對值
ceil(x)	傳回大於輸入值 x 的最小整數
floor(x)	傳回一個比輸入值 x 小的最大整數
log(x)	計算以 e (2.71828) 為底的自然對數值
exp(x)	傳回以 e (2.71828) 為底的冪次方值
pow(a, n)	計算任意 a 的 n 次方
sqrt(x)	求出一個數 x 的平方根
round(x)	四捨五入至整數

Math methods

方法	說明
max(a, b)	傳回兩個數 a, b 中較大的數
min(a, b)	傳回兩個數 a, b 中較小的數
random()	隨機產生一個介於 0~1 的數值
sin(x)	正弦函數
cos(x)	餘弦函數
tan(x)	正切函數
asin(x)	反正弦函數
acos(x)	反餘弦函數
atan(x)	反正切函數

Math Example

- mathRandom.html 利用亂數選擇字串陣列的元素
- 程式碼重點

  index = Math.floor(Math.random()*text.length);
- 說明
  - Math.random() 會傳回一個介於 0 和 1 之間的亂數。
  - 因此 Math.random()*text.length 會產生一個介於 0 和 text.length 之間的亂數（帶有小數）。
  - 最後，Math.floor(Math.random()*text.length) 會產生一個介於 0 和 text.length-1 之間的整數（包含頭尾），所以可以用來選取 text 陣列中的一個元素。

Reference Types - Date

Date Example -- date.html

- new Date() creates a new date object with the current date and time
- Date objects are **static**. The computer time is ticking, but date objects are not.
  - 也就是說 Date 物件內容不會隨時間而改變

Date methods

方法	說明
toString()	以標準字串來表示日期物件
toLocaleString()	以地方字串（依作業系統而有所不同）來表示日期物件
getYear()	取得年份
getMonth()	取得月份（需注意：0 代表一月，因此例如若是八月，結果就是 7）
getDate()	取得日期
getHours()	取得時數
getMinutes()	取得分鐘數
getSeconds()	取得秒數
getDay()	取得星期數（例如若是星期四，結果就是 4）

Function

Function

- a block of code designed to perform a particular task.
- benefits:
  - re-used
    - save code: code is repeated many times with only a few minor modifications
  - modular design: programming in a modular style
  - black-box
    - given a function, we need only know "what it does". There is no need to know "how it does". (maybe we do care if it does the job efficiently.)
  - team collaboration

First-Class Function

- A programming language is said to have First-Class functions when functions in that language are treated like any other variable.
  For example, in Javascript, a function
  - can be passed as an argument to other functions,
  - can be returned by another function
  - can be assigned as a value to a variable.

可當變數傳給其他函數，或當回傳值

Function Definition 1/2

- Syntax

```
function functionName(Arguments) {
    statements
    ...
    return (return_value) // 非必要
}
```

- 括號裡的引數 (Input Arguments)，可以沒有；若有多個則以逗號分開。
- 若有需要，函數最後可用 return 來傳回值 (數值、字串，或其他型態的資料）至呼叫此函數的程式。

Function Definition 2/2

- 函數的定義，通常寫在 <head> 及 </head> 之間，以確保 HTML 主體在被呈現前，所有相關的 JavaScript 函數都已被載入，並隨時可被執行
- 定義函數並不代表函數的執行，只有在程式中呼叫函數的名稱後，才會執行該函數。
- 一般來說，我們希望函數的定義出現的位置和它被呼叫之處能越接近越好，以方便程式管理，在這種情況下，只要函數定義出現在其被呼叫之前即可[note: function hoisted]

Function Definition - Example

- e.g. function.html

```
<script type="text/javascript">
        function show(name, message) {
            alert(name + "," + message);
        }
        show("John", "Hi");
        show("Boy", "Wake up!!");
        show("Girl", "Wake up!!");

        function max(x, y) {  if ( x < y ) return y;  else return x;  }
        var a=1, b=2;
        var c=max(a, b);
</script>
```

Example - Array 的自訂比較函數

陣列的自訂比較函數必須：

- 具有兩個參數 e.g. myComp(arg1, arg2)
- return 一個數值來表示兩個參數間的關係
  - <0, 表示 arg1 排在 arg2 之前
  - =0, 表示 arg1 等於 arg2
    - no guarantee of stability
  - >0, 表示 arg1 排在 arg2 之後
  - 如果 return 一個非數值，do nothing! (no error)
- e.g. arraySort2.html

```
function comparisonFunction(a, b){ return(a-b); }
```

Variable Scope 1/4

變數可因其有效範圍的不同，分成兩類

- Local variables (區域/局部 變數)
  - **var** operator makes the variable local to the scope in which it is defined. (see next slide)
    - 必須在變數第一次使用時加上 **var**
- Global variables (全域變數)
  - if **var** operator is omitted, means to define a global variable (is not recommended)
  - 在整個程式中都可以看的見、 而且每一個函數都可以用的變數。

Variable Scope 2/4

so the story is

- variable defined outside the function is global variable, no matter use operator **var**
- variable defined in the function
  - is local variable if use operator **var**
  - is global variable if do not use operator **var** (is not recommended)
- 如果在函數內有一個變數名稱與全域變數名稱相同，則區域變數優先權 (precedence) 高於全域變數
- 區域變數有助於降低記憶體的浪費，提高系統使用效率
- 請養成用 **var** 宣告變數的好習慣

Variable Scope Example

- e.g. variableScope.html
  - 程式碼重點

```
function 內：   var x=5; // 局部變數
                y=8; // 全域變數
function 外：   x = 10; // 全域變數
                y = 10; // 全域變數
```

- 說明
  - 函數內的變數 x 和外面的 x 雖然名稱一樣，但是用 var 宣告，執行函數後外面的變數 x 值不受影響。
  - 函數內的變數 y 和函數外面 y 的名稱一樣，且執行函數完後，外面的變數 y 會變成 8。
  - 為了減少除錯的時間，所有函數的內部變數，在第一次使用時最好加上 var，以確認其有效範圍只在此函數內。

Variable Scope Types

There are four (technically three) types of scope in javascript:

- window scope (global)
  - These variables can be used anywhere in your script at any time. (global variable)
- function scope
  - These variables are only available inside of a particular function.
- class (arguably same as function)
  - These variables are only available to members of a particular class.
- block (new in ECMAScript 6) [註] - 裡面有{}是-block
  - These variables only exist within a particular code block (i.e., between a set of '{' and '}').

Block Scope

```
for (var i=1; i<10; i++) { … }
console.log(i);  // you will get 10
```

- In Javascript, the variable i still exists outside the for loop

```
for (let i=1; i<10; i++) { … }
console.log(i);  // undefined
```

- Block-Level Scope: the variable i will be destroyed after the loop execution

Block Scope - let, const

- The let and const block bindings introduce lexical scoping to JavaScript. These declarations are not hoisted and only exist within the block in which they are declared. you cannot access variables before they are declared, even with safe operators such as typeof.
- If you use let or const in the global scope, a new binding is created in the global scope but no property is added to the global object. e.g. :

```
var hi = "Hi!";
console.log(window.hi); // work!
let hi = "Hi!";
console.log(window.hi); // error!
```

Call by Value or Call by Reference

指標,記憶體位置

- All function arguments in ECMAScript are passed by value
- Q: What happens if function changes its argument value?
- for primitive types - nothing
  strings, numbers, boolean values, and null are unchanged in the original variable.
  - called by value
  - the actual value of variable is passed to function argument

Call by Value or Call by Reference

- for reference type, the variable store reference values, if object arguments are passed, the address of object be passed. So it is equivalent to be passed by reference
  - called by reference (called by address)
  - A function can change the properties of an object

Function Arguments array

- in addition to set the variable argu1, argu2…, all parameters are stored in variable **arguments**, which is an array
- arguments.length is parameter count
- arguments[i] is the ith paramenter value
  - function functionName(argu1, argu, ….) {
    statements; …
  - }
- 所以在Javascript中呼叫函數時，如果超過函數定義的引數個數時，仍然可執行。

Function Arguments Example

- e.g. arguments.html

```
function sum() {
    let i,total=0;
    for (i=0; i<arguments.length; i++) {
        total = total + arguments[i];
    }
    return total;
}
document.write(sum(10,10)+"<br>");
document.write(sum(10,20,30)+"<br>");
```

recursive function 遞迴函數

- recursion is a technique where a function invokes itself.
  - function functionName(arguments){
    if (…) // 結束條件
    functionName(arguments); // recursive call
    }
- 函數呼叫自己，直到某個條件達成時才停止。
- 如果沒有結束條件，這個函數就會永無止盡的呼叫，形成無窮遞迴。
- 遞迴易設計可以精簡程式碼，但耗CPU資源(但在某些特殊演算法上效果較迴圈佳)。

Classification of Recursion

- Direct Recursion
  - e.g. factorial n! = n * (n-1)!
- Indirect Recursion
  - mutual recursion: function A call function B, then function B call function A.
    e.g. Cyclic Hanoi Towers

recursive function -- factorial.html

- computer the factorial of n in recursion
  - function factorial(n){
    if (n<=1) return(1);
    return (n*factorial(n-1)); // invoke factorial() again
    }
    document.write("5! = ", factorial(5))

- This kind of recursion is called linear recursion, because the stack grows linearly with n.

recursive function -- fibonacci.html

- computer the n'th Fibonacci number in recursion
  - function fibonacci(n){
    if (n<=1) return(1);
    return (fibonacci(n-1)+ fibonacci(n-2));
    }
    document.write("f10 = ", fibonacci(10))

- This kind of recursion is called tree recursion, because the run-time stack grows and shrinks like a tree.

Recursion versus Iteration

	Recursion	Iteration
pro	easy to design	fast
con	time and space overhead	hard to design

- factorial and Fibonacci have obvious iterative solution. Iteration is better solution for them.
- how about McCarthy's 91 function:

  f(n) = n-10, if n>100

  = f(f(n+11)), otherwise

  - recursion or iteration?

f(100)=f(f(111))=f(101)=91
f(91)=f(f(102))=f(92)=f(93)=…=f(100)

f(n) = n-10, if n>100
= 91, otherwise

Recursion versus Iteration

- usually, time is more important than space. we can allocate more memory (tabulation) to store intermediate data, it can improve the recursion performance. e.g.
  combination c(m,n) = c(m-1,n)+c(m-1,n-1)
  = (c(m-2, n)+c(m-2,n-1))+(c(m-2,n-1)+c(m-2,n-2)),
  It's not necessary to compute c(m-2,n-1) twice.

Function Definition more 1/2

- 1. typical way
  - function functioName(argu1, argu2, ….) {…}
    - e.g. function sum(x, y) { return (x+y); }
  - loaded into the scope before code execution
- 2. function expression
  - var functioName=function(argu1, argu2, ….) {…}
    - e.g. var sum=function(x, y) { return (x+y); };
  - create anonymous function, then assign to a variable
  - unavailable until the function expression are executed

Function Definition more 2/2

- 3.defined by Function object constructor
  - var functionName = new Function(argu1String, argu2String, …, bodyString);
    - e.g. var show = new Function("alert('hello!')");
    - e.g. var sum = new Function("x", "y", "return (x+y)");
  - This way is not recommended.
- 4. Arrow Function [ES6]
  - short syntax of expressions function. e.g.
    var sum = (x,y) => x+y;
    is equivalent to
    var sum = function(x,y) {return x+y;}

Function Definition more [note]

- unusual way
  var functionName = function anotherName(listOfVariableNames) { function-body };
- In this particular case, because the function is being assigned, and not defined normally, the name anotherName can be used by the code inside the function to refer to the function itself, but the code outside the function cannot see it at all

Function Concept Summary

- function is an object.
- each function is an instance of the Function type.
- function name is a simple reference to function object.
- function name can be reassigned to new function object

```
var fact = factorial;
document.write("fact(5) = " + fact(5) + "<br>"); // 120
factorial = null;
document.write("fact(5) = " + fact(5) + "<br>"); // error

var factorial = function(n){        // 以遞迴方式進行階乘函數的計算
    if (n==0) return(1);    // 結束條件
    return(n*factorial(n-1));   // 遞迴呼叫
}
document.write("factorial(5) = " + factorial(5) + "<br>");
var fact=factorial;
document.write("1st:fact(5)=" + fact(5) + "<br>");
factorial=null;
//document.write("2nd:fact(5)=" + fact(5) + "<br>");

var factorial = function inside(n){     // 以遞迴方式進行階乘函數的計算
    if (n==0) return(1);    // 結束條件
    return(n*inside(n-1));  // 遞迴呼叫
}
```

要確保值不會被改到
→型別 inside

Arrow Function Limitation

- No this, super, arguments object
- Cannot be called with new
  - Arrow functions do not have a [[Construct]] method.
  - no new.target bindings
- No prototype
  - since you can't use new on an arrow function, there's no need for a prototype.

Function Hoisted

- no master where functions definition are declared, they get hoisted to the top of program block behind the scenes. (include function name and function definition)
- but for the function expression, function implementation do not get hoisted. (only the variable get hoisted.)
- e.g. hoisted.html

[note] let and const are not hoisted

- The let and const are block bindings. These declarations only exist within the block in which they are declared and are not hoisted.

用let宣告還未存在, 用var宣告存在

var. 宣告建議都寫在最前面

Object & Prototype Linkage

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)。

在 JavaScript 中，每個物件都有一個 prototype (原型)，物件可以從原型上繼承屬性和方法，達到復用程式碼的效果，這就是所謂的 prototypal inheritance (原型繼承)。除此之外，原型也能繼承其他物件，因此物件可以繼承一層又一層的屬性和方法，這形成了所謂的 prototype chain (原型鏈)。本篇文章將介紹 prototype、prototype chain，以及 prototypal inheritance 與 class inheritance (類別繼承) 的差異。

JavaScript Prototype 原型

JavaScript 中的每個物件都有一個隱藏的屬性 [[Prototype]]，我們稱它為 prototype (原型)。Prototype 只能是一個物件或是 `null`。我們有一個非標準的方法可以存取 prototype：`__proto__`。

例如，我們可以用 `__proto__` 將 `dog` 物件的 prototype 指定為 `animal` 物件：

```
const animal = {};
const dog = {};

dog.__proto__ = animal;
```

將 A 物件的 prototype 設定為 B 物件，就是 A **繼承 (inherit)** 了 B。

繼承關係還可以更長，例如我們可以再創造一個物件繼承 `dog`：

```
const goofy = {
  __proto__: dog
};
```

那麼設定完物件的 prototype 以後可以幹嘛呢？

接下來要介紹 prototypal inheritance (原型繼承) 的特性了，讓我們一起往下看！

Prototypal Inheritance 原型繼承

Prototype 的功能是：當我們在一個物件查詢某個屬性或方法，找不到的時候，我們會到它的 prototype 裡去查詢。換句話說：

> *JavaScript 的物件能夠「繼承」其 prototype 的屬性或方法。*

例如，我們在 `dog` 物件裡找不到 `isAnimal` 屬性，於是我們到它的 prototype，也就是 `animal` 物件裡查詢 `isAnimal` 屬性，結果找到了：

```
const animal = {
  isAnimal: true
};

const dog = {
  __proto__: animal
};

console.log(dog.isAnimal) // true
```

同樣的原理也適用於 object method (物件方法)，例如我們可以在 `animal` 物件上定義 `eat()` 方法，並且呼叫 `dog.eat()`：

```
const animal = {
  eat() {
    console.log('Eat!');
  }
};

const dog = {
  __proto__: animal
};

dog.eat(); // Eat!
```

Prototypal Inheritance 原型繼承 vs. Class Inheritance 類別繼承

當一個物件繼承自另一個 prototype 物件，它就可以「繼承」prototype 物件上的屬性和方法。這就是所謂的 prototypal inheritance，原型繼承。

那麼使用 prototypal inheritance 原型繼承有什麼好處呢？簡單地說，「繼承」是一種代碼復用的手段。大部分的程式語言可以透過 Class (類別) 繼承達到這個效果，例如：假設我們有 `Dog` 和 `Cat` 兩種物件，都是動物但又各自有些不同的地方，那麼我們可以定義一個 `Animal` class 實作了所有動物的共通點，再定義 `Dog`、`Cat` 類別在 `Animal` 的基礎上各自增加特性。

而 JavaScript 中，我們可以透過 prototype 達成同樣的效果。和 class 繼承最大的差別在於：JavaScript 中物件是繼承自 prototype，而 prototype 本身也是一個物件。繼承自 prototype 就好比你在創造物件時有一個可以效仿的實體，而 class 則像是一張參考的藍圖。

> *延伸閱讀：[教學] 深入淺出 JavaScript ES6 Class (類別)*

Constructor Function (建構函式) 的 Prototype

我們知道 JavaScript 可以用 `new` 運算子加上 constructor function (建構函式) 建立新物件：

```
function Animal(name) {
  this.name = name;
}

const dog = new Animal('Barley');
```

如果想知道用 `new` 建立新物件的詳細原理，可以看一下這篇：

F.prototype.constructor 屬性

`F.prototype` 預設會擁有 `constructor` 屬性。我們可以透過 `constructor` 屬性得知一個物件如何被創造出來的。甚至還可以用來創造新物件！

```
const dog = new Animal('Barley');
const cat = new dog.constructor('Chris');
```

如何利用 Prototypal Inheritance (原型繼承) 模擬 class inheritance

為了達到程式碼復用，我們可能會想讓 `Dog` 可以繼承 `Animal` 上的屬性和方法。常見的物件導向語言可以讓 child class (子類別) 繼承 parent class (父類別)，也就是類似 `class Dog extends Animal` 之類的方式。那使用 prototype 的 JavaScript 該如何達到類似的效果呢？

首先我們要定義 `Dog` constructor，這裏假設 `Dog` 額外帶有一個 `breed` 屬性，用來表示狗的品種。我們需要在 `Dog` constructor 中呼叫 `Animal` constructor：

```
function Dog(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
}
```

然而只有屬性的物件並不是太有用，我們希望建立出來的新物件有一些方法 (method) 可以呼叫。那我們該如何幫新物件增加方法呢？這時候 prototype 就可以派上用場了。

直接說結論：我們得將方法定義在 constructor function 的 `prototype` 屬性上。例如，我們希望建立的新物件有 `eat()` 方法，那我們就得定義 `Animal.prototype.eat`：

```
Animal.prototype.eat = function() {
  console.log('Eat!');
}

const dog = new Animal('Barley');
dog.eat(); // Eat!
```

如果你只想知道怎麼定義一個有方法的物件，那看到這邊就可以了。

但是如果你想知道這個寫法的原理是什麼的話，我們就來一起往下看吧！

F.prototype

在 JavaScript 中，constructor 的 `prototype` 屬性是一個特殊的屬性，當我們把一個 function (這裏假設是 `F`) 當成 constructor 使用時，`F.prototype` 會多一個特殊的用途，讓 JavaScript engine 知道：

> *當我建立新物件的時候，新物件的 prototype 要等於 `F.prototype`。*

舉上面的例子來說，`dog` 物件的 prototype 是 `F.prototype`；換句話說，`dog` 繼承自 `F.prototype`。

我們可以測試 `dog.__proto__` 屬性來印證：

```
const dog = new Animal('Barley');
dog.__proto__ === Animal.prototype; // true
```

簡單地說，因為建立的新物件會繼承 `F.prototype`，所以我們在 `F.prototype` 上定義的方法或屬性，也可以被建立的新物件存取。

這就是為什麼我們要將方法定義在 `F.prototype` 上。

F.prototype 的預設值

`F.prototype` 如果沒有特別指定，預設值會是一個物件，帶有 `constructor` 屬性，指向 constructor 本身：

```
function Animal() {}

console.log(Animal.prototype); // { constructor: Animal }
console.log(Animal.prototype.constructor === Animal) // true
```

F.prototype.constructor 屬性

`F.prototype` 預設會擁有 `constructor` 屬性。我們可以透過 `constructor` 屬性得知一個物件如何被創造出來的。甚至還可以用來創造新物件！

```
const dog = new Animal('Barley');
const cat = new dog.constructor('Chris');
```

如何利用 Prototypal Inheritance (原型繼承) 模擬 class inheritance

為了達到程式碼復用，我們可能會想讓 `Dog` 可以繼承 `Animal` 上的屬性和方法。常見的物件導向語言可以讓 child class (子類別) 繼承 parent class (父類別)，也就是類似 `class Dog extends Animal` 之類的方式。那使用 prototype 的 JavaScript 該如何達到類似的效果呢？

首先我們要定義 `Dog` constructor。這裏假設 `Dog` 額外帶有一個 `breed` 屬性，用來表示狗的品種。我們需要在 `Dog` constructor 中呼叫 `Animal` constructor：

```
function Dog(name, breed) {
  Animal.call(this, name);
  this.breed = breed;
}
```

`Animal.call(this, name)` 是為了執行 `Animal` constructor 內所有的初始化動作，包含讓建立的新物件帶有 `Animal` 建立物件的屬性 (這裡指的是 `name`)。透過 `Animal.call(this, name)` 我們不用把重複的代碼全部貼到 `Dog`，達到程式碼復用的效果。

現在有另外一個問題：我們沒辦法存取 `Animal` 定義的方法：

```
const dog = new Dog('Barley', 'Golden Retriever');
dog.eat(); // Uncaught TypeError: dog.eat is not a function
```

為什麼呢？答案在於 `Dog.prototype` 在沒有特別指定的情況下是預設的 prototype，上面查詢不到任何 `Animal` 的方法。怎麼辦呢？解法很簡單，我們只要讓 `Dog.prototype` 繼承 `Animal.prototype` 就行了：

```
Dog.prototype = Object.create(Animal.prototype);
```

如果不熟悉 `Object.create()` 的讀者，可以看一下這篇唷。

> *延伸閱讀：[教學] JavaScript new、Function Constructor (建構函式) 及 Object.create()*

這個做法還會衍生一個問題，就是 `Dog.prototype.constructor` 的值會變成 `Animal`，因為 `Dog.prototype` 繼承 `Animal.prototype`，而 `Animal.prototype.constructor === Animal`。

解法是我們要幫 `Dog.prototype` 手動加上 `constructor` 屬性：

```
Object.defineProperty(Dog.prototype, 'constructor', {
  value: Dog,
  enumerable: false, // so that it does not appear in 'for in' loop
  writable: true
});
```

Object Literal

- can create object with an object literal

```
var student = {name:"Timmy", id:"0200102"};
```

  - 註: 這種物件格式很像 JSON (JavaScript Object Notation) 格式，但是 JSON 要求屬性名稱一律要使用引號，JavaScript 不要求

- 註: 使用 object literal 的好處是可以指定含有空格的屬性，但是若要存取此屬性，則必須使用 obj["propertyName"] 的方式來進行，而不能使用 obj.propertyName 的方式。

built-in Object type

- or create object with an built-in Object type
  - Object is the fundamental reference type
  - all objects are instances of Object type
- create an instance of Object, e.g.

  var o = new Object();
- customize object, e.g. object.html:

  var student = new Object();

  student.name = "Timmy";

  student.id = "0200102";

  先產生一個原型物件，然後增加 property，產生客制化物件。

```
<script>
var student = new Object();
student.name = "Timmy";
student.id = "0200102";

//JSON (Javascript Simple Object Notation)
var student1 = {"name":"Timmy", "id":"0200102"};
// property 不含空白時，可不用引號
var student2 = {name:"Mary", id:"0200103"};
var student3 = {name:"Alice", id:"0200104"};

// 列舉物件中的所有屬性
for ( field in student ) {
    document.write( "student." + field + " = " +
        student[field] // or eval("student."+field)
        + "<br>" );
}

// 以 in 運算子測試物件的欄位是否存在
field="name";
if (field in student) {
    document.write(field + " is a field of student<br>");
    document.write(field," value: ",student[field],"<br>");
}
</script>
```

student.name = Timmy
student.id = 0200102
name is a field of student
name value: Timmy

[note] in operator

- 以 in 運算子測試物件的欄位是否存在
  - in 除了搭配 for 使用外，也可以用來檢查物件是否含有某屬性，如果這個屬性存在，就會回傳 true，不存在則回傳 false。
- e.g. object.html

```
if (field in student)
    document.write(field + " is a field of
student<br>");
```

Factory Pattern

- 使用函數搭配原型 Object 產生物件
- e.g. factory.html

```
function student(name, id) {
    var o = new Object();
    o.name = name;
    o.id = id;
    return o;
}
var student1 = student("Timmy", "0200102");
```

生物件也可以定義方法

```
12          var o = new Object();
13          o.name = inputName;
14          o.studentID = inputStudentID;
15          o.age = inputAge;
16          o.display = function () {
17              document.writeln(
18                  "<br>姓名 : " + o.name
19                  + "<br>學號 = " + o.studentID
20                  + "<br>年齡 = " + o.age
21                  + "<br>");
22          };
23          return o;
24      }
```

```
10  <script>
11  function student(inputName, inputStudentID, inputAge) {
12      var o = new Object();
13      o.name = inputName;
14      o.studentID = inputStudentID;
15      o.age = inputAge;
16      o.display = function () {
17          document.writeln(
18              "<br>姓名 : " + o.name
19              + "<br>學號 = " + o.studentID
20              + "<br>年齡 = " + o.age
21              + "<br>");
22      };
23      return o;
24  }
25
26  var student1 = student("Alex","0200102",23);
27  var student2 = student("Joey","0200103",20);
28  var student3 = student("Kelvin","0200104",22);
29
30  student1.display();
31  student2.display();
32  student3.display();
33  </script>
```

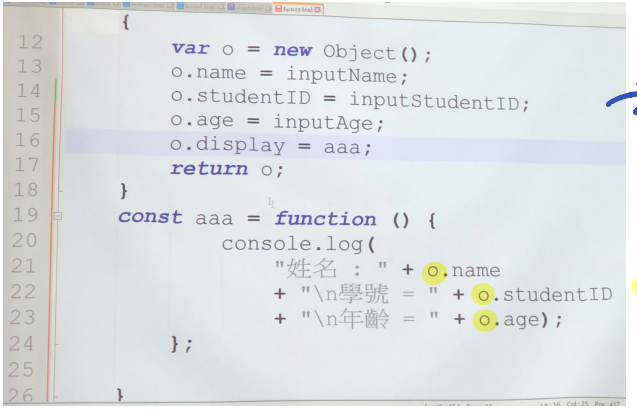
姓名 : Alex
學號 = 0200102
年齡 = 23

姓名 : Joey
學號 = 0200103
年齡 = 20

姓名 : Kelvin
學號 = 0200104
年齡 = 22

```
        var o = new Object();
        o.name = inputName;
        o.studentID = inputStudentID;
        o.age = inputAge;
        o.display = function () {
            document.writeln(
                "<br>姓名 : " + o.name
                + "<br>學號 = " + o.studentID
                + "<br>年齡 = " + o.age
                + "<br>");
        };
        return o;
    }
```

多個要寫很多次



```
    {
        var o = new Object();
        o.name = inputName;
        o.studentID = inputStudentID;
        o.age = inputAge;
        o.display = aaa;
        return o;
    }
    const aaa = function () {
        console.log(
            "姓名 : " + o.name
            + "\n學號 = " + o.studentID
            + "\n年齡 = " + o.age);
    };

}
```

→ 統一函數重複用

o可改用this，因呼叫時this指的是自己

Contructor Pattern

- e.g. constructor.html

```
function Student(name, id) {
    this.name = name;
    this.id = id;
    this.display = function () { … };
}
var student1 = new Student("Timmy", "0200102");
```

- 說明
  - 使用 Constructor (建構子) 概念來產生物件
  - this 代表 new 所產生的物件

The basic difference is that a constructor function is used with the `new` keyword (which causes JavaScript to automatically create a new object, set `this` within the function to that object, and return the object):

```
var objFromConstructor = new ConstructorFunction();
```

A factory function is called like a "regular" function:

```
var objFromFactory = factoryFunction();
```

But for it to be considered a "factory" it would need to return a new instance of some object: you wouldn't call it a "factory" function if it just returned a boolean or something. This does not happen automatically like with `new`, but it does allow more flexibility for some cases.

In a really simple example the functions referenced above might look something like this:

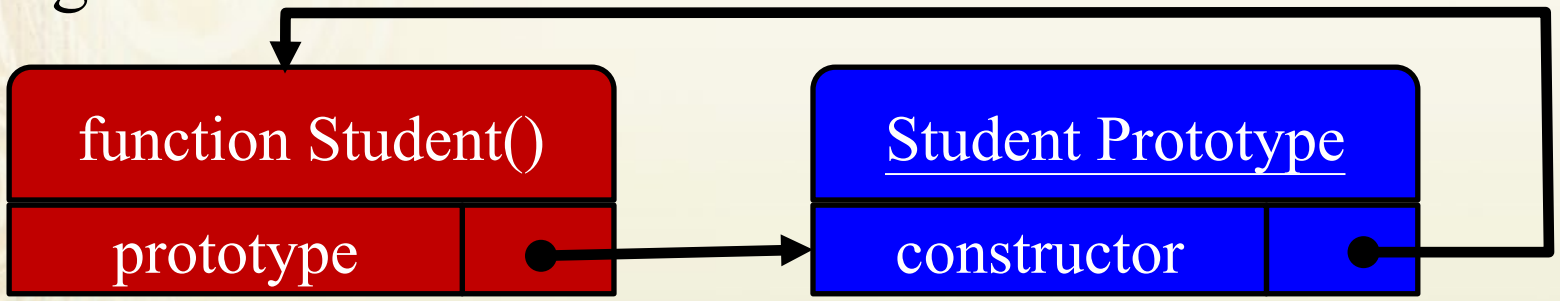
```
function ConstructorFunction() {
   this.someProp1 = "1";
   this.someProp2 = "2";
}
ConstructorFunction.prototype.someMethod = function() { /* whatever */ };

function factoryFunction() {
   var obj = {
      someProp1 : "1",
      someProp2 : "2",
      someMethod: function() { /* whatever */ }
   };
   // other code to manipulate obj in some way here
   return obj;
}
```

Constructor Pattern's Imperfection

- The major downside to constructors is that methods are created once for each instance.
  - e.g. each instance of Circle get its own instance of area() that output the circle area.
- one of solutions is to move the function definition outside of the constructor.
  - cons: create function in the global scope
- another solution is to use prototype pattern

Prototype Object of Function

- each function is created with a prototype property, which refer to a Prototype object
- e.g.




```
function Student(…) { … };
Student.prototype.display = function () {…};
Student.prototype.extension = "others";
```

```
38    console.log(student1 instanceof Student); //true
39    console.log(student1 instanceof Object);  //true
40
41    console.log(student1.constructor == Student);  //true
42    console.log(student1.constructor == Object);  //false
```

看Student是否是student1

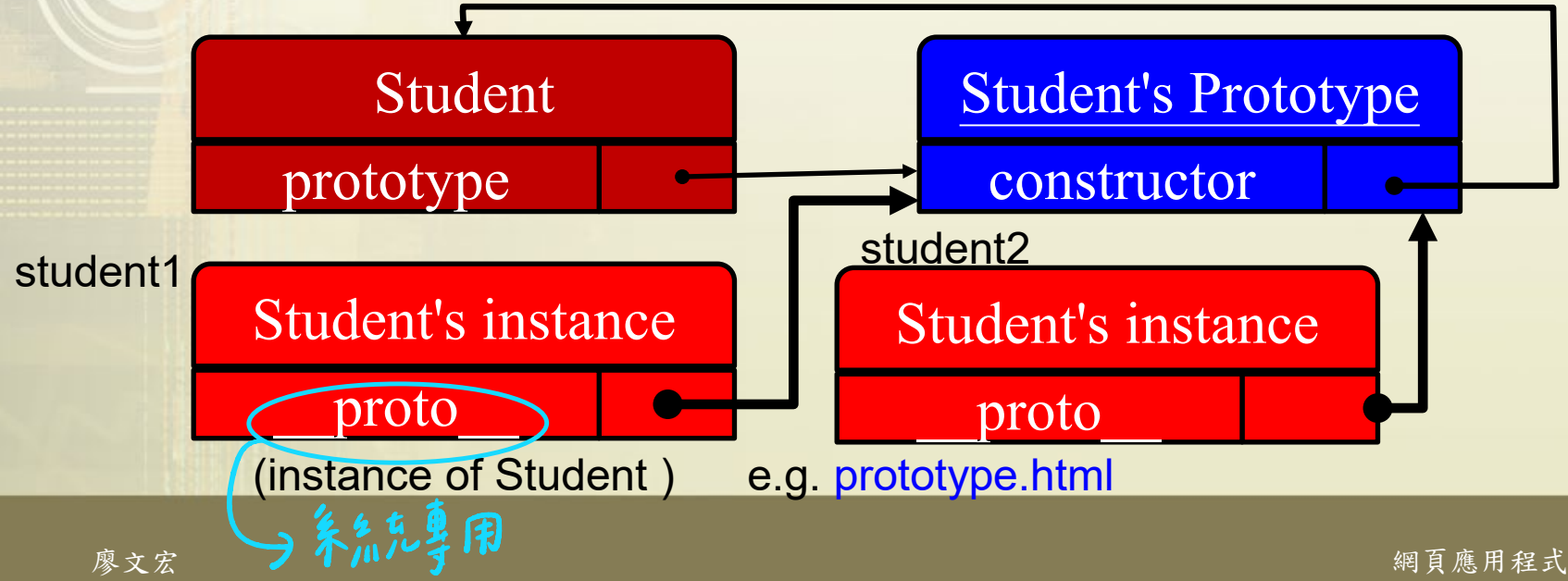
constructor可找到原型

How prototype work

在定義物件(系統會自動)就會產生物件

- every instance has an internal pointer to the constructor's prototype object.
  - this pointer may be named __proto__, dependent on browser
  - so all instances share the prototype object

產生Student1沒有constructor但其內自動內建指標會找,找到就有3



```
    <h2>Constructor Pattern</h2>
    <hr>
    <script>
    function Student(inputName, inputStudentID, inputAge) {
        this.name = inputName;
        this.studentID = inputStudentID;
        this.age = inputAge;
        this.display = function () {
            document.writeln(
                "<br>大名 : " + this.name
                + "<br>學號 = " + this.studentID
                + "<br>年齡 = " + this.age
                + "<br>"
            );
        };
    }

    var student1 = new Student("Alex","0200102",23);
    var student2 = new Student("Joey","0200103",20);
    var student3 = new Student("Kelvin","0200104",22);

    student1.display();
    student2.display();
    student3.display();

    console.log(student1.display == student2.display);  //false

    console.log(student1 instanceof Student); //true
    console.log(student1 instanceof Object);  //true

    console.log(student1.constructor == Student);  //true
    console.log(student1.constructor == Object);  //false
```

大名 : Alex
學號 = 0200102
年齡 = 23

大名 : Joey
學號 = 0200103
年齡 = 20

大名 : Kelvin
學號 = 0200104
年齡 = 22

```
    <title>Prototype Pattern</title>
</head>
<body>
<script>
    // 1. example Student
    function Student(name, id) {
        this.name = name;
        this.id = id;
    }
    Student.prototype.display = function () {
        document.writeln(
            "<br>大名 : " + this.name
            + "<br>學號 = " + this.id
            + "<br>"
        );
    };
    var student1 = new Student("Alex","0200102");
    var student2 = new Student("Joey","0200103");
    var student3 = new Student("Kelvin","0200104");

    student1.display(); // Student.prototype.display()
    student2.display();
    student3.display();

    console.log(student1.display == student2.display);  //true

    console.log(student1 instanceof Student); //true
    console.log(student1 instanceof Object);  //true

    console.log(student1.constructor == Student);  //true
    console.log(student1.constructor == Object);  //false
    console.log(Student.prototype.constructor == Student);  //true

    console.log(Student.prototype.isPrototypeOf(student1));  //true
    console.log(Student.prototype.isPrototypeOf(student2));  //true

    console.log(student1.hasOwnProperty("display"));  //false
    // Student.display(); // TypeError: undefined is not a function
    console.log("display" in student1);  //true
```

大名 : Alex
學號 = 0200102

大名 : Joey
學號 = 0200103

大名 : Kelvin
學號 = 0200104

```html
<!DOCTYPE HTML>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Prototype Pattern</title>
</head>
<body>
<script>
    // 1. example Student
    function Student(name, id) {
        this.name = name;
        this.id = id;
    }
    Student.prototype.display = function () {
        document.writeln(
            "<br>大名 : " + this.name
            + "<br>學號 = " + this.id
            + "<br>"
        );
    };
    var student1 = new Student("Alex","0200102");
    var student2 = new Student("Joey","0200103");
    var student3 = new Student("Kelvin","0200104");

    student1.display(); // Student.prototype.display()
    student2.display();
    student3.display();

    console.log(student1.display == student2.display);  //true

    console.log(student1 instanceof Student); //true
    console.log(student1 instanceof Object);  //true

    console.log(student1.constructor == Student);  //true
    console.log(student1.constructor == Object);  //false
    console.log(Student.prototype.constructor == Student);  //true

    console.log(Student.prototype.isPrototypeOf(student1));  //true
    console.log(Student.prototype.isPrototypeOf(student2));  //true

    console.log(student1.hasOwnProperty("display"));  //false
    // Student.display(); // TypeError: undefined is not a function
    console.log("display" in student1);  //true
    //function hasPrototypeProperty(obj, prop){
    //  return !obj.hasOwnProperty(prop) && (prop in obj);
    //}

    document.writeln("<hr>");

    // possible application:
    // I. output array string, defaultly joined by '|'
    var a=[1,2,3]; // new Array(1,2,3);
    document.writeln(a,"<br>"); // a.toString()
    document.writeln(a.join('|'),"<br>"); // coded for every array
    // change the default joining format
    Array.prototype.toString = function () { return this.join('|') }
    document.writeln(a,"<br>"); // now, write without join()
    var b = [4,5,6];
    document.writeln(b,"<br>"); // now, write without join()
    // II. hidden the function code
    function sayHi() { alert("hi"); }
    document.writeln(sayHi, "<br>"); //sayHi.toString()
    document.writeln(sayHi.toString(), "<br>"); //outputs "function sayHi() {alert("hi");}"
    Function.prototype.toString = function () { return "Function codes are hidden"; };
    document.writeln(sayHi, "<br>"); //outputs "Function codes are hidden"

    // 2. example Person
    function Person(name) {
        this.name = name;
    }
    Person.prototype.getName = function() {return this.name;};
    Person.prototype.friends = ["Alice", "Sandy"];
    var person1 = new Person("John");
    var person2 = new Person("Mary");
    person1.friends.push("Michael");
console.log(person2.friends);
/*
    // user-defined prototype object
    Person.prototype = {
        getName: function () { return this.name; },
        friends: ["Alice", "Sandy"],
        //constructor: Person,
    }
    var person1 = new Person("John");
    var person2 = new Person("Mary");
    console.log("user-defined prototype object:");
    console.log(person1.getName());
    console.log(person1.constructor == Person);  //false
    console.log(person1.constructor == Object);  //true
    console.log(Person.prototype.constructor == Object);  //true

    person1.friends.push("Michael");
    console.log(person2.friends); // "Alice", "Sandy", "Michael"
    console.log(person1.friends === person2.friends); // true

    // 3. example Circle
    const PI = 3.1415926; // ES6; ES5: var PI = 3.1415926;

    function Circle(radiux) {
        this.radiux = radiux;
    }
    Circle.prototype.area = function () {
        return this.radiux * this.radiux * PI;
    }
    var circle1 = new Circle(3);
    var circle2 = new Circle(5);
    document.writeln(circle1.area(), "<br>");
    document.writeln(circle2.area(), "<br>");
    document.writeln(circle1.area == circle2.area, "<br>");
*/
</script>
```

元素可是陣列

```html
<!DOCTYPE html>
<html>
<head>
<title>Prototype Chaining</title>
<script type="text/javascript">

    function Person(name = 'Adam') {
        this.name = name;
    }
    Person.prototype.getName = function(){return this.name;};

    function Student(name, id){
        //this.name = name;
        Person.call(this, name);
        this.id = id;
    }

    //inherit from Person
    Student.prototype = new Person();
    console.log(Student.prototype.getName()); //Adam

    var inst = new Student("Eve","9911001");
    console.log(inst.getName());   // "Eve"
    console.log(inst instanceof Student);  // true
    console.log(inst instanceof Person); // true
    console.log(inst instanceof Object); // true
    console.log(inst instanceof Function); // false  <--
    console.log(Student instanceof Function); // true

    console.log(Student.prototype.isPrototypeOf(inst)); //true
    console.log(Person.prototype.isPrototypeOf(inst)); //true
    console.log(Object.prototype.isPrototypeOf(inst)); //true
    console.log(Function.prototype.isPrototypeOf(inst)); //false  <--
    console.log(Function.prototype.isPrototypeOf(Student)); //true

    //override existing method
    Student.prototype.getName = function (){
        return this.id;
    };
    console.log(inst.getName());   // "9911001"
    delete Student.prototype.getName;
    console.log(inst.getName());   // "Eve"
</script>
</head>
</html>
```

How prototype work constructor / prototype 比較

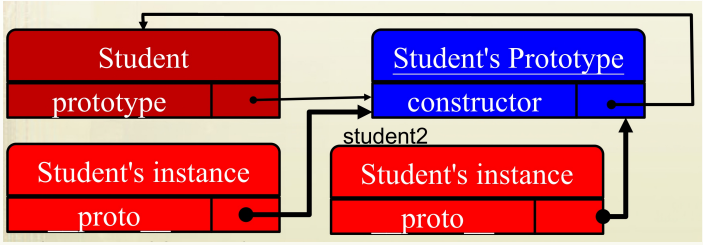
- Whenever a property is accessed, the search begins on the object instance itself. If not found, continues to search the __proto__ object.
- This is how prototypes are used to share properties and methods among multiple object instance.

自己沒有就會去找原型

本來查物件有沒有屬性：用 hasOwnProperty

X用 in 會找原型有沒有

How prototype work [note]



- e.g.

```
var foo = function (){ };
foo.hi = function(){console.log('foo.hi');};
foo.prototype.hi = function(){console.log('prototype.hi');};
foo.hi(); // foo.hi
var f = new foo();
f.hi(); // prototype.hi
```

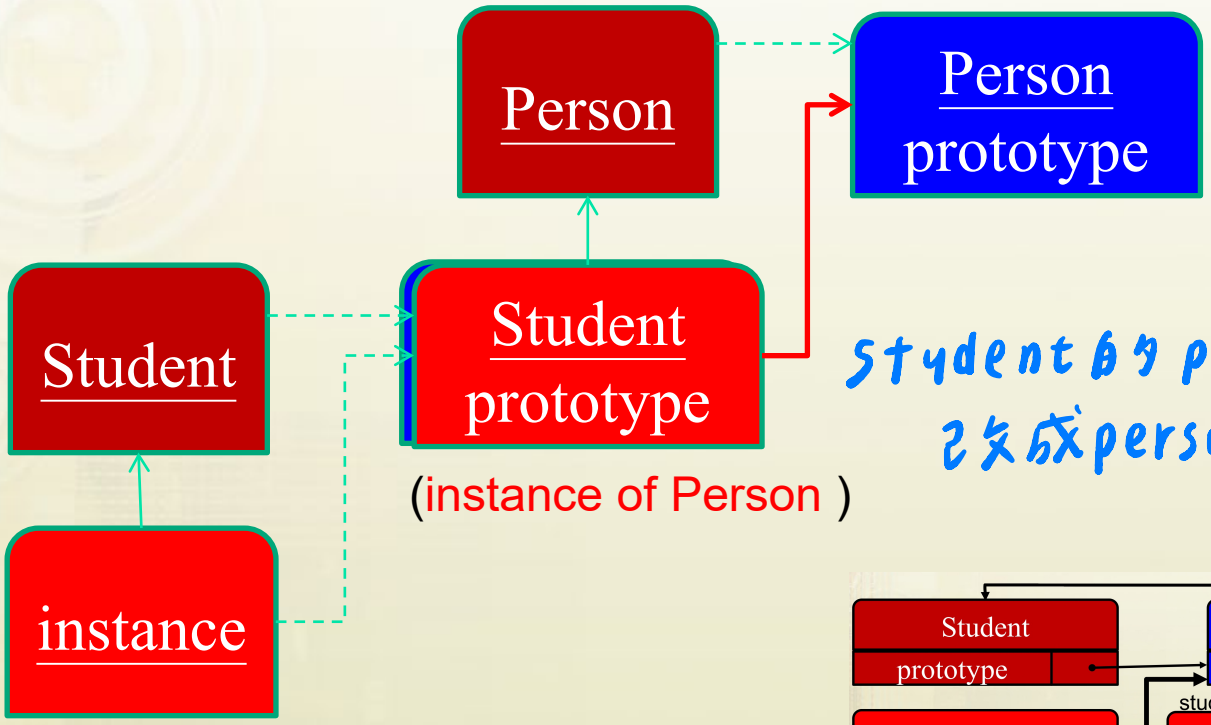
紅色 hi

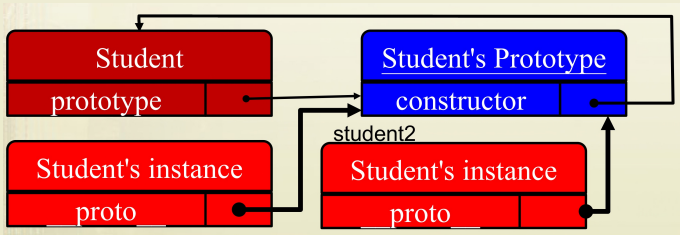
藍色 hi (f自己本身沒有hi)

- hasOwnProperty() determines if a property exists on the instance. It returns true only if the property on the instance itself, not on prototype.
  e.g. f.hasOwnProperty("hi")
- e.g. if (hi in f)
  - in operator returns true if the property is accessible, means the property can be on the instance or on the prototype.

prototype chain 會隔代繼承

- 可利用 prototype 特性模擬物件的繼承
  e.g. prototypeChain.html

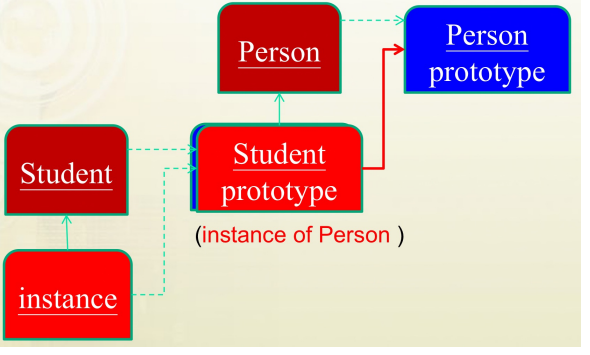




```html
<!DOCTYPE html>
<html>
<head>
<title>Prototype Chaining</title>
<script type="text/javascript">

    function Person(name = 'Adam') {
        this.name = name;
    }
    Person.prototype.getName = function(){return this.name;};

    function Student(name, id){
        //this.name = name;
        Person.call(this, name);
        this.id = id;
    }

    //inherit from Person
    Student.prototype = new Person();
    console.log(Student.prototype.getName()); //Adam

    var inst = new Student("Eve","9911001");
    console.log(inst.getName());   // "Eve"
    console.log(inst instanceof Student);  // true
    console.log(inst instanceof Person); // true
    console.log(inst instanceof Object); // true
    console.log(inst instanceof Function); // false  <--
    console.log(Student instanceof Function); // true


    console.log(Student.prototype.isPrototypeOf(inst)); //true
    console.log(Person.prototype.isPrototypeOf(inst)); //true
    console.log(Object.prototype.isPrototypeOf(inst)); //true
    console.log(Function.prototype.isPrototypeOf(inst)); //false  <--
    console.log(Function.prototype.isPrototypeOf(Student)); //true

    //override existing method
    Student.prototype.getName = function (){
        return this.id;
    };
    console.log(inst.getName());   // "9911001"
    delete Student.prototype.getName;
    console.log(inst.getName());   // "Eve"
```

getName 由紅至少能才找至少(找2次)

display(找一次)



Class [ES6]

- e.g. class.html
- A class body can only contain methods, but not properties
- Class declarations are not hoisted
- Class be invoked only via new, not via function call
- two ways to define a class:
  class declarations and *class expressions*.

```
<script>
    class Person {
        constructor(name) {
            this.name = name;
        }
        getName() {
            return this.name;
        }
    }
    class Student extends Person {
        constructor(name, id) {
            super(name);
            this.id = id;
        }
        display() {
            return "****"+super.getName()+"****"+this.id;
        }
    }

    var student = new Student("Adam", "2011001");
    console.log(student.getName());
    delete Student.prototype.getName;
    console.log(student.getName());
</script>

```

class要先定義