03. More on JavaScript

Ric Huang / NTUEE

(EE 3035) Web Programming

More on JavaScript

- Object prototype and inheritance
- More on functions
- Arrow function
- “this” and bind()
- More on types and variables
- More on array and array methods
- Other useful JS grammars/concepts/tips
- Class in JavaScript

Object Prototype and Inheritance

Recall: Prototype-Based object construction

- Object: collection of properties
  - property: key(or name)-value pair
  - value can be a function ==> method

```
var Student = function (name, id) { ... }
var a = new Student("Ric", 101);
var b = new Student("Mary", 301);
a.score = 100;  // property 'score' only
                // belongs to 'a'
```

Recall: Object can be created in many ways

1. Using object initializers

```
var obj = { property_1:   value_1, ... }
```

2. Using constructor function

```
function Car(make, model, year)
{ this.make = ... }
// is the same as
var Car = function(make, model, year)
{ this.make = ... }
// Then you can use "new" to create objects
var car1 = new Car("Toyota", "Wish", 2015);
```

Recall: Object can be created in many ways

3. Using Object.create method

```
// Using an existing object as prototype
var obj = { value: 10 }
var newObj = Object.create(obj);
console.log(newObj.value); // 10
newObj.value = 20;      // assign a new value
```

Closer Look at Constructor Function

- Tell the differences...

```
// #1 Function call to a ref to an anonymous function
var Car = function(make) { this.make=make; }
var car = Car("BMW");
console.log(car);           // undefined
console.log(car.make);      // ERROR
```

```
// #2 Constructor call to a ref to an anonymous function
var Car = function(make) { this.make=make; }
var car = new Car("BMW");
console.log(car);           // { make: "BMW" }
console.log(car.make);      // "BMW"
```

Closer Look at Constructor Function

- Tell the differences...

```
// #3 Ref to a ref to an anonymous function
var Car = function(make) { this.make=make; }
var car = Car;
console.log(car);           // is a function
console.log(car.make);      // undefined
```

```
// #4 Constructor call to a ref to an anonymous function
var Car = function(make) { this.make=make; }
var car = new Car;          // missing parameter
console.log(car);           // is an object
console.log(car.make);      // undefined
```

Closer Look at Constructor Function

- Tell the differences...

```
// #5 Create an object from another object as
//    a prototype
var Car = function(make) { this.make=make; }
var car = new Car("BMW");
console.log(car);           // { make: "BMW" }
console.log(car.make);  // "BMW"

var myCar = Object.create(car);
console.log(myCar); // { } <- no own property
console.log(myCar.make);// "BMW" <- inherited
                        //          property
```

Object Inheritance

- All objects in JavaScript inherit from at least one other object.
- The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor.
    - Don't get confused with the “inheritance” in class-based language(e.g. C++).
    - “class” in JS is just a syntactical sugar (supported in ES6, covered later). JavaScript remains prototype-based.

Prototype Chain

- Each object has a private property which holds a link to another object called its prototype.
- That prototype object has a prototype of its own, and so on until an object is reached with null as its prototype. By definition, null has no prototype, and acts as the final link in this prototype chain.
- Nearly all objects in JavaScript are instances of Object which sits on the top of a prototype chain.

Inheriting Properties

- An object inherits all the properties from the objects along the prototype chain

```
var Car = function(make, model) {
  this.make = make; this.model = model; }
var car = new Car("BMW", "X5");

var myCar = Object.create(car);
myCar.model = "X3"; // over-write and
                    // make it own property
myCar.year = 2018;  // own property

console.log(myCar.driver); // undefined
```

Object.keys(obj) vs. for (var i in obj)

- Object.keys(obj) returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of the object obj.

```
Object.keys(myCar);   // ["model", "year"]
```

- for (var i in obj) traverses all enumerable properties of an object and its prototype chain.

```
for (var i in myCar)
  console.log(i);  // "model", "year", "make"
```

List of the enumerable properties

```
var Obj = function() { this.a = 10; this.b = 20; }
var obj1 = new Obj();
Object.keys(obj1);   // [ "a", "b" ]

var obj2 = Object.create(obj1);
obj2.c = 30;
Object.keys(obj2);   // [ "c" ]

console.log(obj2.a); // 10
obj2.a = 40;         // overwrite and make it own
                     // property
Object.keys(obj2);   // [ "c", "a" ]
```

You can add a property through prototype

```
var Car = function(make) { this.make = make; }
var car = new Car("BMW");
var myCar = Object.create(car);
myCar.year = 2020;

Car.prototype.model = "X5";
console.log(myCar.model);   // "X5"
```

- However, “model” is a property in Car.__proto__

```
Object.keys(myCar);  // ["year"]
Object.keys(car);    // ["make"]
for (var i in myCar) // "year", "make", "model"
  console.log(i)
```

Inheriting “methods”

- Recall: property in an object can be a function

```
var Car = function(make) {
  this.make = make;
  this.drive = function () {
    console.log(this.make + " is driving");}
}
var car = new Car("BMW");
car.drive();     // "BMW is driving"

var myCar = Object.create(car);
myCar.make = "Lexus";
myCar.drive();   // "Lexus is driving"

Car.prototype.park = function () { // add a function
    console.log(this.make + " is parking"); }
myCar.park();    // "Lexus is parking"
```

More on Functions

Nested function and closure

- Recall: “var” is local variable with function scope
- Recall: There can be functions in a function
  - “…the inner function can see the variables in the outer function, but not the other way around.”
- BUT, the inner function can be accessed only from statements in the outer function.
- Therefore, the inner function forms a closure (閉包)
  - The inner function inherits the arguments and variables from the outer function, and the outer function can call the inner function only through its arguments
- Encapsulation for the vars of the inner function.

Examples of “Closure”

```
function addSquares(a, b) { // a^2 + b^2
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
a = addSquares(2, 3); // returns 13
b = addSquares(3, 4); // returns 25
c = addSquares(4, 5); // returns 41
```

- 呃？那為什麼不直接 return a*a + b*b 就好？

Return the closure inner function

```
function addN(x) {
  function increment(y) {
    return y + x;
  }
  return increment;
}
var add3 = addN(3);     // give me a function
                        // that adds 3
var    a = add3(5);     // returns 8
var    b = addN(3)(5);  // returns 8
```

- 其實上述例子比較現代化的寫法是用 “arrow function”

Arrow function (=>)

- Arrow function is just like lamda function in Python (more to cover later)
- 語法：

```
(argument1,... argumentN) => {
  // function body
}
```

- 但如果 function body 只有 “return expression”，則可簡化成 — // 沒有 { }, 也不可有 “return”

```
(argument1,... argumentN) => expression
```

Arrow function (=>)

- 而如果 argument 只有一個，則可以省略 ()

```
argument => expression
```

- 如果沒有 argument, 或是有兩個或以上的 arguments, 則 () 不可省略

- 但如果 expression 是 object definition, 則要加上 ()

```
argument => ({ property: value })
```

Arrow function examples

```
const hw = () =>
             { console.log("Hello, world!"); }
hw();                     // "Hello, world!"
const add = (a, b) => a + b;
add(3, 5);                // 8
const mapFirst = array => ({first: array[0]})
mapFirst([3, 1, 4]);  // { first: 3 }
```

- 前面 closure 的例子用 arrow function 改寫...

```
function addN(x) { return (y => y + x); }
```

- 或者是全用 arrow functions...

```
let addN = x => (y => y + x);
```

Note: Arrow function is anonymous

1. Assign it to another variable // as a reference

```
const add = (a, b) => a + b;
```

2. Call it on the spot

```
(() => { console.log("Hello, world!"); })()
(() => console.log("Hello, world!"))()
```

3. Pass it as a parameter

```
[1, 3, 5].map(e => e*2);  // [2, 6, 10]
```

Callback functions in Array methods

```
[1, 3, 5].map(e => e*2);  // [2, 6, 10]
```

- 前面例子的 arrow function 是一個 callback function 的例子
- Other useful callback functions for Array —

```
arr.map(e => e*2);                // return newArr
arr.filer(e => e.length < 10);    // return newArr
arr.every(e => e.length < 10);    // return true/false
arr.some(e => e.length < 10);     // return true/false
arr.reduce((e1, e2) => e1 + e2);// reduce to one
                                  // element
arr.forEach(e => console.log(e));
```

Callback function

- In general, a callback function f() is passed as a functional object to another function g(args), like g(f, ...args).
- Note that callbacks are often used to continue code execution after an asynchronous operation has completed — these are called asynchronous callbacks

```
function getImg(imgGot, url) {
  var httpState = someAsyncFunToGetImg(url);
  imgGot(httpState); // callback function to
                     // inform the caller
}
```

Use arrow function as a callback

// Note: "setInterval(f, ms)" calls f() per ms milliseconds

```
function Person() {
  this.age = 0;
  setInterval(() => { this.age++; }, 1000);
}
var p = new Person(); // p.age will increase
```

- However, if we rewrite the arrow function as a regular function, it won’ t work…

```
function Person() {
  this.age = 0;
  setInterval
  (function growUp() { this.age++; }, 1000);
} // p.age remains 0
```

“this” in function

- "this" in function refers to the caller object

```
function Person() {
  this.age = 0;
  setInterval(function growUp()
  { console.log(this); this.age++; }, 1000);
}
```

- Who calls “setInterval()” ? ==> Window

“this” in function

- How about this?

```
function Person() {
  this.age = 0;
  function growUp() { this.age++; }
  setInterval(growUp(), 1000);
}
```

- The same. It's Window that calls setInterval

“this” in function

- How about this?

```
function Person() {
  this.age = 0;
  setInterval
  (function growUp() { age++; }, 1000);
}
```

[Beware] You got an error
 “Uncaught ReferenceError: age is not defined at growUp” every second!!

- One way to fix this... with "that"

```
function Person() {
  var that = this; // that is a local variable
  that.age = 0;
  setInterval(function growUp() {
    that.age++;    // refer to that local var
  }, 1000);
}
var p = new Person();
```

A more “elegant” old way is to use “bind()”…

- “Function.prototype.bind(thisArg[, … otherArgs])” creates a new function that use “thisArg” as “this” for the bound function

```
function Person() {
  this.age = 0;
  function growUp() { this.age++; }
  setInterval(growUp.bind(this), 1000);
}
var p = new Person();
```

How arrow function works for “this”?

- An arrow function does not have its own this; the this value of the enclosing execution context is used.

```
function Person() {
  this.age = 0;
  setInterval(() => { this.age++; }, 1000);
}
var p = new Person(); // p.age will increase
```

More on Types and Variables

Recall: Using “var” to declare a variable

- If a variable is declared outside any function, it is a global variable and is visible in the entire program.
- If a variable is declared inside a function, it is visible only in that function.
- If a function is defined inside another function, the inner function can see the variables in the outer function, but not the other way around.

Recall: Using “let” and “const”

- "let" defines a block-scoped local variable

```
if (true) { let y = 5; }
console.log(y);  // ReferenceError: y is not
defined
```

- "const" defines “read-only” variables. However, the properties inside a const object are NOT constrained

```
const obj = { aa: 10, bb: "Hello" };
obj = 30;       // ERROR
obj.aa = 30;  // This is OK!!
```

All numbers are double (floating numbers)

- Shift (<<, >>) operator: 將數字轉成最多 32-bit 的 signed int 來進行操作
- Numeric values or string?

```
"30" + 8  ==> "308"
"30" - 8 ==> 22
"30" * 8 ==> 240
"30" / 8 = 3.75
"30" << 8 ==> 7680
"30" >> 8 ==> 0
```

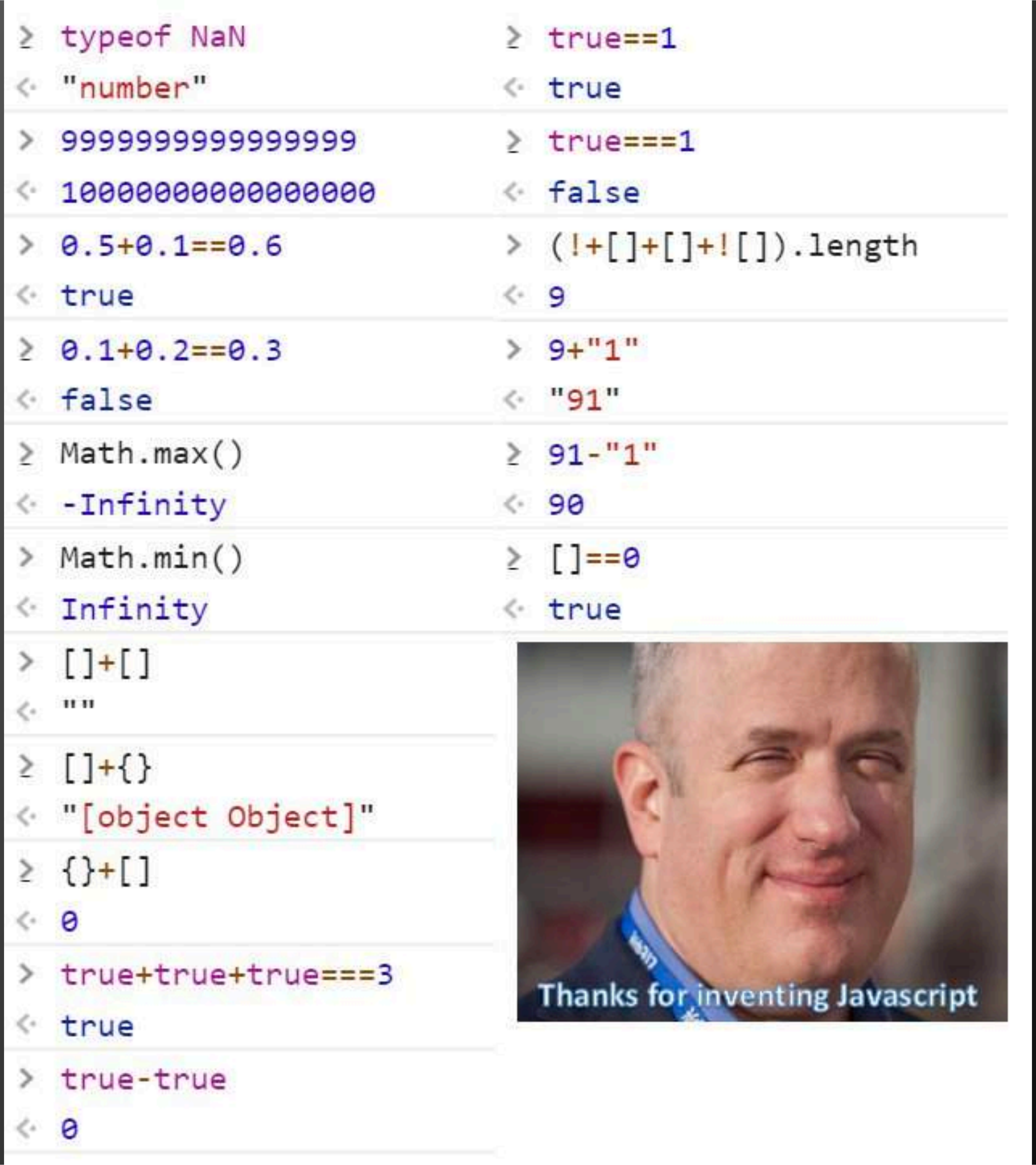
Converting strings to numbers

- parseInt(string, base) // base = [2,36]

```
parseInt("15") = 15 // default is decimal
parseInt(0xF) = parseInt("0xF")
= parseInt("0xF", 16) = 15
parseInt(017) = parseInt("017", 8)
= parseInt("015") = 15
parseInt(017, 8) = parseInt("015", 8) = 13
parseInt("15*3", 10) ==> parseInt("15", 10)
==> 15
parseInt("321", 2) ==> NaN
```

- See also: parseFloat(“string”);

What the hack...



```
> typeof NaN
<· "number"
> 9999999999999999
<· 10000000000000000
> 0.5+0.1==0.6
<· true
> 0.1+0.2==0.3
<· false
> Math.max()
<· -Infinity
> Math.min()
<· Infinity
> []+[]
<· ""
> []+{}
<· "[object Object]"
> {}+[]
<· 0
> true+true+true===3
<· true
> true-true
<· 0
```

```
> true==1
<· true
> true===1
<· false
> (!+[]+[]+![]).length
<· 9
> 9+"1"
<· "91"
> 91-"1"
<· 90
> []==0
<· true
```

More on object properties

Don’t put " " in property name declaration

- Even though these are the same:

```
var a = { i: 10, "jj": 20 }

var a = { i: 10, jj: 20 }
```

- But this is an error

```
console.log(a."jj"); // ERROR!!

console.log(a.jj);   // OK!
```

Access the property value

1. By ‘.’
   - Property name must be a valid JS identifier
   - “s.createdAt” is OK, but “s.created\ at” is not OK

2. By [ ]
   - Property name can be any valid JS string
   - s['created at'] = '2019.03.06 下午';
   - s[''] = "nothing"; // even empty string

Access the property value

```
ric.name;          // “Ric”
ric."name";        // ERROR!!
ric[name];         // undefined, if var name is
                   // undefined
ric["name"];       // “Ric”
ric[0];            // undefined, if ric is not an
                   // array
```

- 使用 [] 來存取 object 的 property 有一個好處是可以 “動態的設定” property name

```
var a;
if (someExpression) { a = "A"; }
else { a = "B"; }
// assume someExpression is false
ric[a] = 70;
console.log(ric.A);  // undefined
console.log(ric.B);  // 70
```

- Property name can almost be anything…

```
var car = {
  manyCars: { a: "Saab", "b": "Jeep"},
         7: "Mazda",
        "": "An empty string",
       "!": "Bang!"
};
car.manyCars.a;  // "Saab"       car.manyCars."a";  // ERROR!!
car.manyCars.b;  // "Jeep"       car.manyCars."b";  // ERROR!!
car.manyCars[a]; // undefined    car.manyCars["a"]; // "Saab"
car.manyCars[b]; // undefined    car.manyCars["b"]; // "Jeep"
car."7";   // ERROR!!            car.7;   // ERROR!!
car["7"];  // "Mazda";           car[7];  // "Mazda"
car."";    // ERROR!!            car[""]; // "An empty string"
car."!";   // ERROR!!            car.!;   // ERROR!!
car["!"];  // "Bang!"
```

Built–in “Math” Object

```
// The built-in Math object provides many
// useful methods:
Math.PI;           // 3.14159...
Math.sin(radian);  // e.g. Math.sin(Math.PI/
4) = 0.7071...
Math.pow(2, 5);    // 32
Math.floor(22/3);  // 7
Math.round(-3.49); // -3
Math.min(3, 5, 7); // 7
Math.random();     // Between [0, 1)
Math.sqrt(179);    // ~13.38
```

Built-in “Date” Object

```
// current local time (up to second)
var now = new Date();
var XMasDinner
= new Date("December 25, 2019 18:30:00");
var XMasDinner
= new Date(2019, 11, 25, 18, 30, 0)
var myBirthday
= new Date("December 02, 2001");
var myBirthday
= new Date(2001, 11, 02);  // 11 is December
// Output: "Sun Dec 02 2001 00:00:00 GMT+0800
(台北標準時間)"
```

Built-in "Date" Object

- Useful Date methods (ref)

```
Date.now()

// Date.propotype getter methods
getDate(), getDay(), getHours(), getYear(),
getTime(), getUTCDate()…

// Date.propotype setter methods
setDate(), setHours(), setYear(), setTime(),
setUTCDate()…

// Date.propotype conversion methods
toString(), toDateString(), toJSON(),
toLocaleDateString()…
```

Built-in “Date” Object

- Year vs. FullYear

```
var date = new Date(98, 1);
// Sun Feb 01 1998 00:00:00 GMT+0000 (GMT)
date.setYear(98); // Obsolete, although working
// Sun Feb 01 1998 00:00:00 GMT+0000 (GMT)
date.setFullYear(98);
// Sat Feb 01 0098 00:00:00 GMT+0000 (BST)
```

- Compute the elapsed time

```
var start = Date.now();
doSomethingForALongTime();
var end = Date.now();
var elapsed = end - start;
```

```
var start = new Date();
doSomethingForALongTime();
var end = new Date();
var elapsed = end.getTime()
            - start.getTime();
```

More on Array and Array Methods

Array construction

- Array can be constructed in 3 different ways

```
// These are the same
var arr = [ 0, 1, 2, 3 ];
var arr = new Array(0, 1, 2, 3);  // 不建議
var arr = Array(0, 1, 2, 3);
```

Array.length

```
var arr = ['one', 'two', 'three'];
arr[2];              // three
arr['length'];       // 3; same as "arr.length"
arr['foo'];          // undefined
arr.foo = "bar";     // As an object, arr can
                     // have properties
arr['foo'];          // "bar"
```

```
var cats = [];
cats[30] = ['Dusty'];
console.log(cats.length); // 31
```

Element or length?

```
var arr = [ 42 ];         // arr[0] = 42
var arr = [ "42" ];       // arr[0] = "42"
var arr = [ 1, "42" ];    // arr[0] = 1,
                          // arr[1] = "42"
var arr = Array(42);      // arr.length = 42,
                          // arr[0] = undefined
// The above is the same as:
var arr = [];
arr.length = 42;
```

Iterate through an Array

```
var array = ['first', 'second', , 'fourth'];
for (var i = 0; i < array.length; i++)
  console.log(array[i]);
// Output: first, second, undefined, fourth
```

- Or using forEach() with arrow function ==> Will skip undefined element!!

```
var array = ['first', 'second', , 'fourth'];
array.forEach(i => console.log(i));
// Output: first, second, fourth
```

Other useful Array methods

```
var newArr = arr.concat(elements);  // [ arr, elements ]
var newArr = arr.join(', '); // join elements with ', '
arr.push(5,6,7);             // to the end
var element = arr.pop();     // remove and return the last
var firstElem = arr.shift(); // remove and return the first
var length = arr.unshift(8,9,0); // add to the front
// extracts a section of an array and returns a new array
var newArray = arr.slice(startIdx, uptoIdx);
arr.reverse();
arr.sort();
```

Other Useful JavaScript Grammars/Concepts/Tips

各種 for

- 你知道的 for

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

- for (⋯ in ⋯)
  // loop through property names

```
for (propertyName in object) statement
```

- for (⋯ of ⋯)
  // loop through property values on
  // iterable object

```
for (propertyValue of object) statement
```

for(... in ...) vs. for (... of ...)

```
var arr = [3, 5, 7];
arr.foo = 'hello';

for (var i in arr) {
   console.log(i); // "0", "1", "2", "foo"
}

for (var i of arr) {
   console.log(i); // 3, 5, 7; NO "hello"
}
```

“in” operator

- The “in” operator returns true if the specified property is in the specified object

```
// Array
var trees = ['redwood', 'bay', 'cedar', 'oak', 'maple'];
0 in trees; 3 in trees;      // returns true
6 in trees; 'bay' in trees; // returns false
'length' in trees;           // returns true

// built-in objects
'PI' in Math;              // returns true
var myString = new String('coral');
'length' in myString;   // returns true

// Custom objects
var mycar = { make: 'Honda', model: 'Accord', year: 1998 };
'make' in mycar;  // returns true
'model' in mycar; // returns true
```

Shift operator

- "<<" performs signed left-shifting operation

```
9 << 2; /* 36 */          -9 << 2; /* -36 */
```

- ">>" performs signed right-shifting operation

```
// Thinking in 2's complement, with (+/-)
sign preserved
9 >> 2; /* 2 */     -9 >> 2; /* -3 */
```

- ">>>" performs zero-filled right-shifting operation

```
// Thinking in 2's complement,
// with 0's filled from left
9 >>> 2; /* 2 */    -9 >>> 2; /* 1073741821 */
```

Spread (...) operator

- Spread (···) operator allows array, string, or function to expand the argument specification in place for 0 or more operands

```
// for function calls
myFunction(...iterableObj);

// for array or string
[...iterableObj, '4', 'five', 6];

// for object literals (new in ES2018)
let objClone = { ...obj };

// Example:
function myFunction(v, w, x, y, z) { }
var args = [0, 1];
myFunction(-1, ...args, 2, ...[3]);
```

Rest (...) operator

- Rest syntax looks exactly like spread syntax but is used for destructuring arrays and objects, while "spread" is used for expanding arguments

```
// Syntax
function f(a, b, ...restArgs)... // restArgs is an array

// Examples
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a); console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}
myFun("one", "two", "three", "four", "five", "six");

function f(...[a, b, c]) { return a + b + c;}
f(1)          // NaN (b and c are undefined)
f(1, 2, 3)    // 6
f(1, 2, 3, 4) // 6 (the fourth parameter is not destructured)
```

Destructuring Assignment

- The destructuring assignment is to unpack values from arrays, or properties from objects, into distinct variables.

```
var a, b, rest;
[a, b] = [10, 20];
console.log(a);      // expected output: 10
console.log(b);      // expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(rest);  // expected output: [30,40,50]
```

Template Literals (Strings)

- 有的時候我們用字串變數來 access 一些物件、Array 內的 properties/values, 在程式中需要動態的決定字串的值，像是 [“bg-green” , “bg-red” , “bg-yellow”], 我們希望傳進 function 的參數可以是：
  - someFunction(“bg-dynamicallyDecided”), 其中 “dynamicallyDecided” 希望可以動態決定
- Template literals 的語法是：

```
`string text ${expression} string text`
* 其中 expression 可以是一個變數，甚是是一個算式
```

- 範例：

```
var a = 5, b = 10;
console.log(`Fifteen is ${a + b} and not ${2 * a + b}.`);
// "Fifteen is 15 and not 20."
```

“instanceof” operator

- The “instanceof” operator returns true if the specified object is of the specified object type

```
var theDay = new Date(1995, 12, 17);
if (theDay instanceof Date) {
  // statements to execute
}
```

“typeof” operator

- The “typeof” operator returns a string indicating the type of the unevaluated operand

```
var myFun = new Function('5 + 2');
var shape = 'round';
var size = 1;
var foo = ['Apple', 'Mango', 'Orange'];
var today = new Date();

typeof myFun;          // returns "function"
typeof shape;          // returns "string"
typeof size;           // returns "number"
typeof foo;            // returns "object"
typeof today;          // returns "object"
typeof doesntExist;    // returns "undefined"
```

Exception Handling

- 許多時候，當你的 code 有錯誤，你只會得到一個白螢幕，或者是許多的小菜包
  - 當然，console.log() 是你 debug 的好朋友
  - 但往往要不是發現的時候太晚了，就是還要開 console 才能看得到
- 用 Exception Handling 來第一時間抓到錯誤！

Exception Handling。語法

```
try {
    doSomething();
} catch(e) {  // e 可能為系統的錯誤訊息，或者是
              // throw 出來的 exp
    takeSomeAction(e);
} finally {   // 不管有沒有 error 都會被 call
    takeCareTheFinalStep();
}
function doSomething() {
   // if something wrong, throw an error
   throw errorExpression;
}
```

Exception Handling ◦ Example

```
try {
    const a = 10;
    doSomething(a);
} catch(e) {
    alert(e);
}
function dosomething(a) { a++; }
```

Class in JavaScript

JavaScript Classes in ES-6

- JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugars over JavaScript's existing prototype-based inheritance.
- The class syntax does not introduce a new object-oriented inheritance model to JavaScript.

Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

Class Declarations

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

- Compared to —

```
function Rectangle(height, width) {
  this.height = height;
  this.width = width;
}
```

No class hoisting!!

```
const p = new Rectangle(); // ReferenceError!!

class Rectangle {}
```

Class Methods

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}

const a = new Rectangle(10, 20);
a.calcArea();  // 200
```

Compared to —

```
function Rectangle(height, width) {
  this.height = height;
  this.width = width;
  this.calcArea = function () {
    return this.height * this.width;
  }
}

const a = new Rectangle(10, 20);
a.calcArea();
```

With get/set methods

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() { return this.calcArea(); }
  // Method
  calcArea() { return this.height * this.width; }
}

const a = new Rectangle(10, 20);
a.calcArea();  // 200
a.area;        // 200
```

What's the difference between getter/setter and a standard function call?

- "A difference between using a getter or setter and using a standard function is that getters/setters are automatically invoked on assignment. So it looks just like a normal property but behind the scenes you can have extra logic (or checks) to be run just before or after the assignment."

Ref: https://stackoverflow.com/questions/42342623/why-use-getters-and-setters-in-javascript

Static Methods

- Static methods are called without instantiating their class and cannot be called through a class instance.
- Static methods are often used to create utility functions for an application.

Static Method Example

```
class Point {
  constructor(x, y) { this.x = x; this.y = y; }
  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.hypot(dx, dy);
  }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
console.log(Point.distance(p1, p2));
// 7.0710678118654755
```

Be careful about "this", again!

- In the above example, since there is no caller object when “Point.distance(p1, p2)” is called, no this is defined!

Class Inheritance

- Using the keywords extends and super

```
class Animal {
  constructor(name) { this.name = name; }
  speak() { console.log(this.name + ' makes a noise.'); }
}
class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class constructor and
                 // pass in the name parameter
  }
  // Overload parent class' method
  speak() { console.log(this.name + ' barks.'); }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

Recall: Object Inheritance

- All objects in JavaScript inherit from at least one other object.
- The object being inherited from is known as the prototype, and the inherited properties can be found in the prototype object of the constructor.
  - Don’t get confused with the “inheritance” in class-based language(e.g. C++).
  - “class” in JS is just a syntactical sugar (supported in ES6, covered later). JavaScript remains prototype-based.

Recall: Object Inheritance

- myDog inherits "Animal" through the method "Object.create(new Animal("mitzie")"
- myDog.name is an inherited properties but does NOT reside in "myDog"

```
function Animal(name) {
  this.name = name;
  this.speak = function () {
    console.log(this.name + ' makes a noise.');
  }
}
var myDog = Object.create(new Animal("mitzie"));
myDog.speak();            // "mitzie makes a noise"
console.log(myDog);       // { } <- no own property
console.log(myDog.name);// "mitzie" <- inherited
                        //            property
```

Recall: Object Inheritance

- console.log(myDog), you will see —

```
myDog
    Animal {}
          __proto__: Animal
                name: "mitzie"
                speak: ƒ ()
                __proto__: Object
```

- This is an object inheritance, NOT a function inheritance, so you cannot do —

```
var d = new myDog("someName");
```

Recall: Object Inheritance

- But if we change it to:

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

- Dog is now a function
- However, cannot do "`var d = new Dog("Mitzie")`"

So, what does class inheritance do, exactly?

In the previous class inheritance example...

```
class Animal {
  constructor(name) { this.name = name; }
  speak() { console.log(this.name +
                    'makes a noise.'); }
}
class Dog extends Animal {
  constructor(name) {
    super(name); // call the super class
                 // constructor and pass
                 // in the name parameter
  }
  // Overload parent class' method
  speak() { console.log(this.name +
                       ' barks.'); }
}

let d = new Dog('Mitzie');
d.speak(); // Mitzie barks.
```

- console.log(d), you will see —

```
d
  Dog {name: "Mitzie"}
    name: "Mitzie"
    __proto__: Animal
      constructor: class Dog
      speak: ƒ speak()
      __proto__: Object
```

- "name" is a property in "Dog", not in "Animal"
  => Different from previous object inheritance example

class inheritance is equivalent to the following function inheritance...

```
function Animal(name) {
  this.name = name;
  this.speak = function () {
    console.log(this.name + ' makes a noise.');
  }
}
function Dog(name) {
  Animal.call(this, name);
  this.speak = function() {
    console.log(this.name + ' barks.');
  }
}
Dog.prototype = Object.create(Animal.prototype);
let d = new Dog('Mitzie');
d.speak();
```

FYI: Function.prototype.call()

- The call() method of Function calls a function with a given this value and arguments provided individually.

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

- With "Dog.prototype = Object.create(Animal.prototype)"

```
d
  Dog {name: "Mitzie",
       speak: ƒ}
    name: "Mitzie"
    speak: ƒ ()
    __proto__: Animal
      __proto__:
        constructor:
              ƒ Animal(name)
        __proto__: Object
```

- Without "Dog.prototype = Object.create(Animal.prototype)"

```
d
  Dog {name: "Mitzie",
       speak: ƒ}
    name: "Mitzie"
    speak: ƒ ()
    __proto__:
      constructor:
            ƒ Dog(name)
      __proto__: Object
```

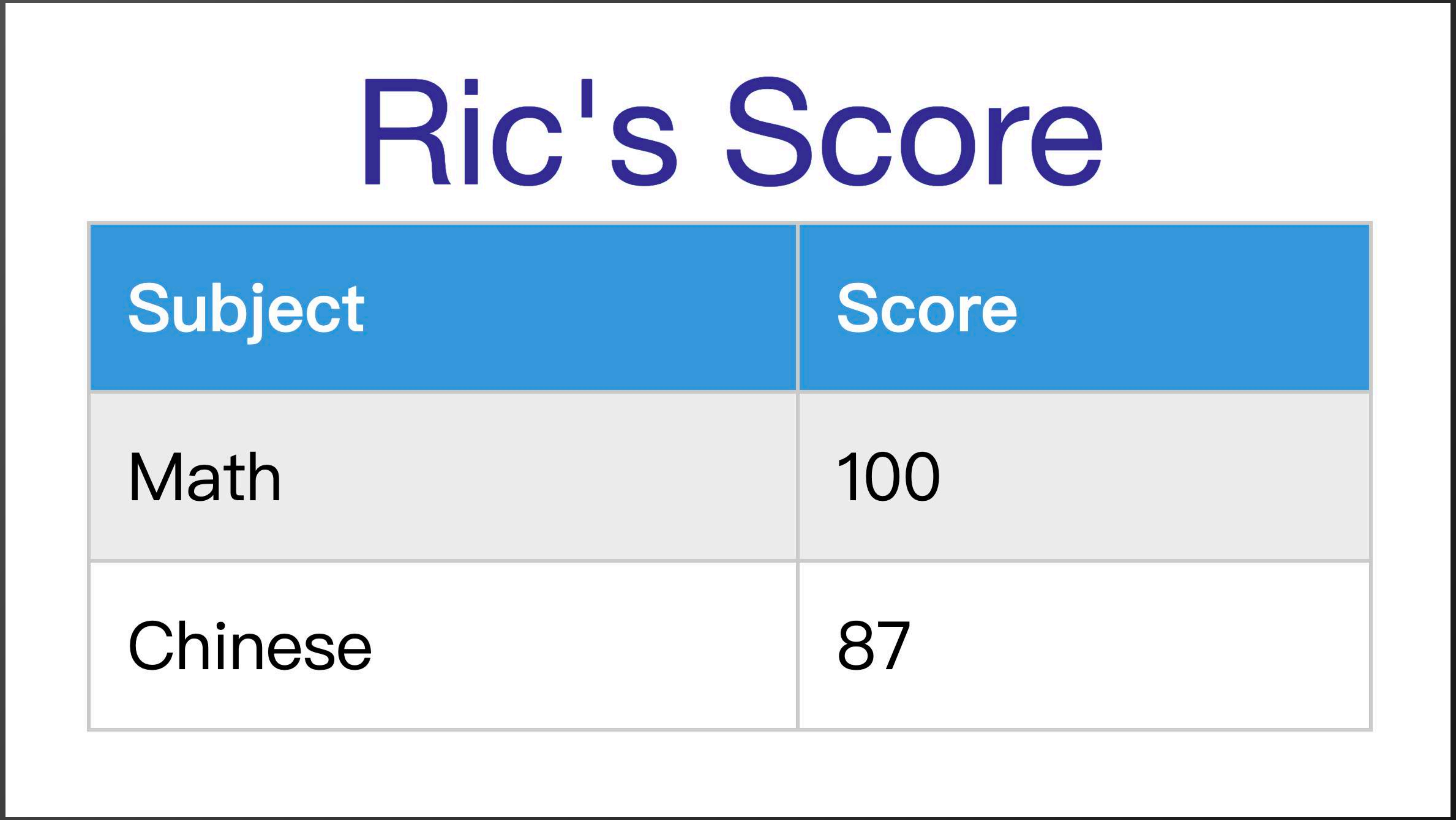
Calling super class's method

- Again, using super

```
class Cat {
  constructor(name) { this.name = name; }
  speak() { console.log
            (`${this.name} makes a noise.`); }
}
class Lion extends Cat {
  speak() { super.speak();
            console.log(`${this.name} roars.`);}
}
let l = new Lion('Fuzzy');
l.speak();
// Fuzzy makes a noise.
// Fuzzy roars.
```

You've learned "class"
in JavaScript.
How can it help you?

Let's construct a simple table



Ric's Score

Subject	Score
Math	100
Chinese	87

- Download pure HTML implementation from here.

Think: If there are going to be more columns and rows, the HTML file can become very long and thus hard to maintain.

Ideally, we would like to have a JS file like...

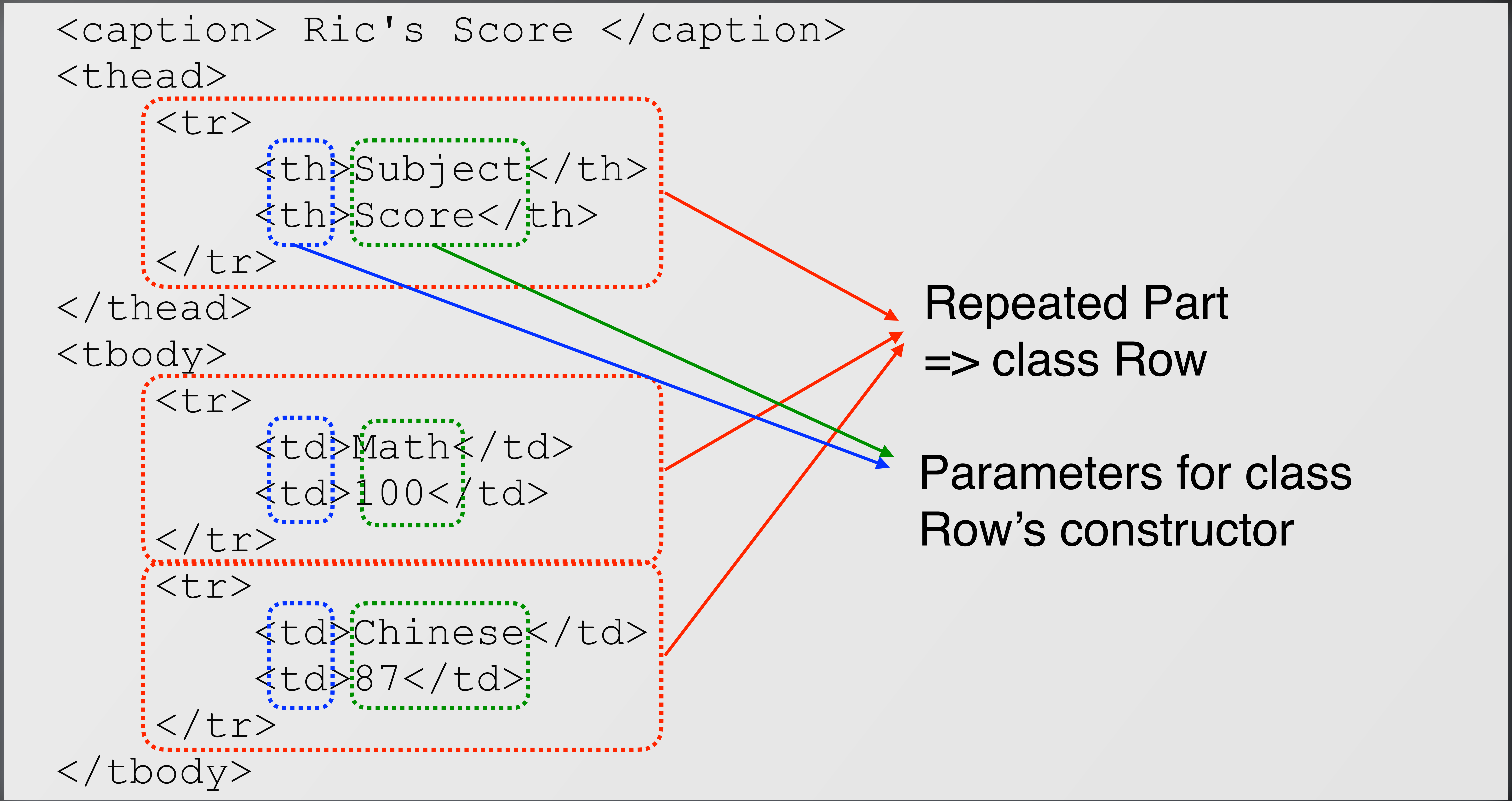
```
const who = "Ric";
const columnIndex = ["Subject", "Score"];
const scoreCard = {
    name: `${who}`,
    records: [
        ["Math", 100],
        ["Chinese", 87]
    ],
};

// Define a class,
// and based on the data above,
// create the table content
```

How to define a class?

- What are the repeated parts in HTML?
- Make a proper class name
- What are the parameters for the constructor?
- What will be the properties for the class?

To define a class for the table...



```
<caption> Ric's Score </caption>
<thead>
    <tr>
        <th>Subject</th>
        <th>Score</th>
    </tr>
</thead>
<tbody>
    <tr>
        <td>Math</td>
        <td>100</td>
    </tr>
    <tr>
        <td>Chinese</td>
        <td>87</td>
    </tr>
</tbody>
```

Some useful functions...

- document.createElement(tagName);
- someNode.textContent = textString;
- someNode.appendChild(childNode);

In-Class Practice!!

【How do "class" help you?】

Any better way to write the code and make it cleaner and more extensible (maybe more object-oriented)?

We will introduce the React.JS framework starting from next lecture

感謝聆聽！

Ric Huang / NTUEE

(EE 3035) Web Programming