

JavaScript

Séance 3

Ceci n'est pas un objet

Contenu

- ✖ this
- ✖ new
- ✖ prototype
- ✖ __proto__
- ✖ constructor
- ✖ lookup
- ✖ héritage
- ✖ __defineGetter/Setter__

Contenu

- ✖ Variable statique
- ✖ `__lookup__`/hasOwnProperty
- ✖ Apply vs Call
- ✖ Programmation générative
- ✖ eval
- ✖ new Function
- ✖ bind
- ✖ package ?

this et new

```
> function F1 () { a = this; }  
undefined  
> F1()  
undefined  
> a  
Window
```

```
> new F1()  
F1  
> a  
F1  
> a()  
TypeError: object is not a function  
> a.constructor  
function F1() { a = this; }
```

Le mot-clé New

- ✖ Il s'applique sur une fonction
- ✖ Il crée un objet qui sera accessible par `this` dans la fonction new-isée
- ✖ L'objet créé aura pour constructeur (`constructor`) la fonction new-isée

Quel intérêt ?

prototype = un template

```
> F1.prototype.x=2
2
> a.x
2
> b=new F1()
F1
> b.x
2
```

prototype

- ✖ Chaque objet est un dictionnaire de propriétés
- ✖ Un dictionnaire secondaire lui est associé à sa création
`<fonction>.prototype`
- ✖ Par la suite, on peut y accéder via
`__proto__`

__proto__ ≠ <function>.prototype

```
> function F1() {}  
undefined  
> F1.prototype.x=3;  
3  
> a=new F1()  
F1  
> F1.prototype = { y : 2 }  
Object  
> b = new F1()  
F1  
> a.y  
undefined  
> a.x  
3  
> b.x  
undefined  
> b.y  
2
```

```
> a.__proto__==F1.prototype  
false  
> b.__proto__==F1.prototype  
true  
> F1.prototype.isPrototypeOf(b)  
true
```

L'objet se base sur le prototype associée à la fonction au moment de sa création

constructor

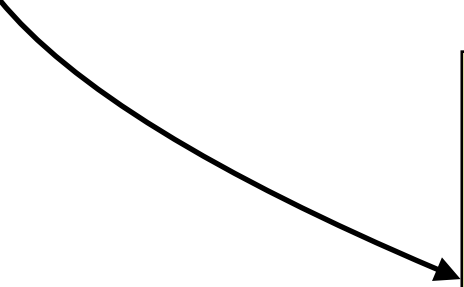
- ✱ `a.constructor` renvoie le constructeur du `__proto__` de `a`
- >> la fonction qui a été new-isée pour créer le prototype utilisé par `a` comme template

Pas simple

hein ?

Exemple

```
> function F1() {}  
> F1.prototype.x=3  
// F1.prototype est une instance de F1  
  
> function F2() {}  
> F2.prototype = { x : 3 }  
// F2.prototype est une instance de Object
```



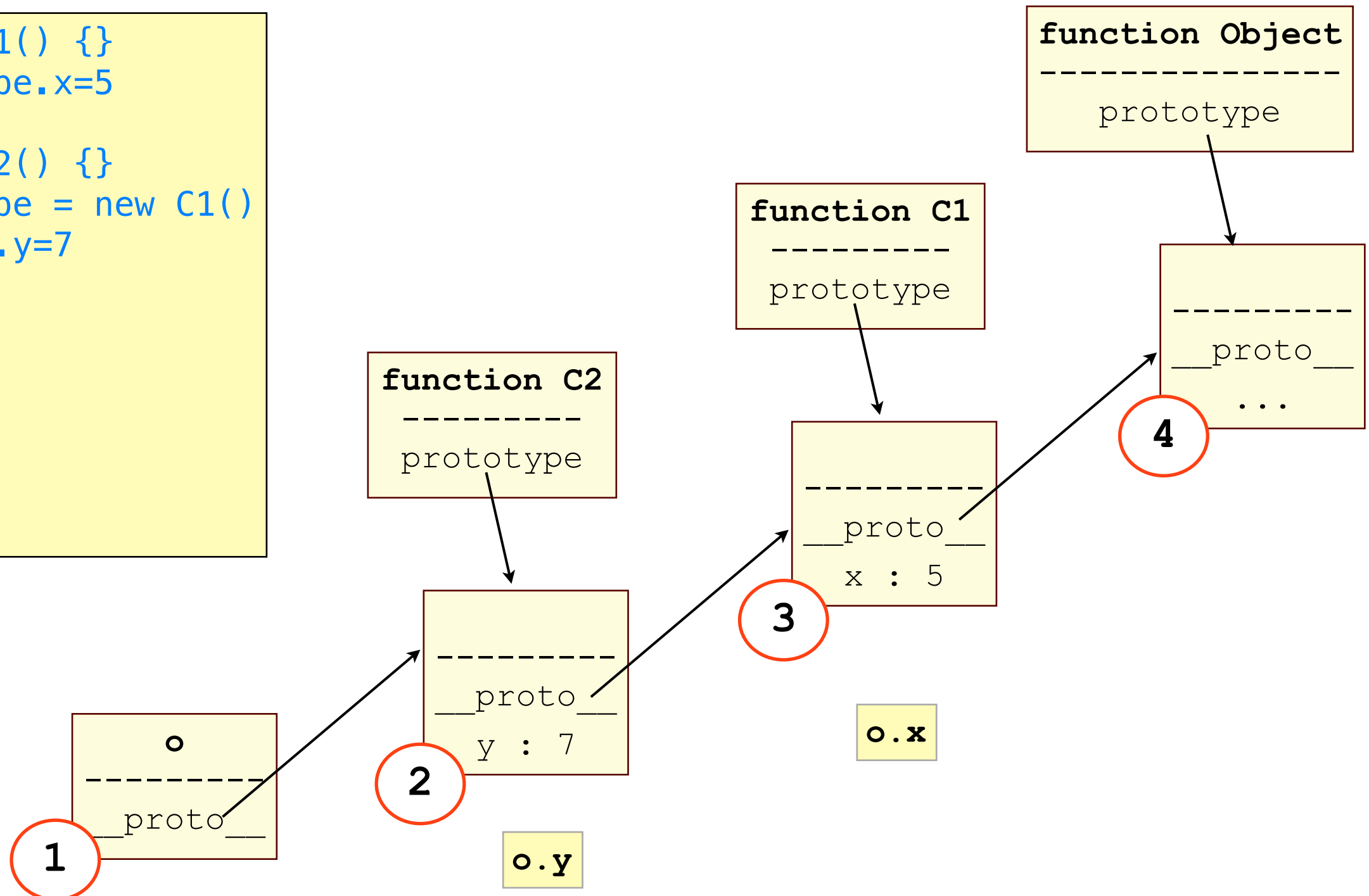
```
> a=new F1()  
F1  
> b=new F2()  
F2  
> a.constructor  
function F1() {}  
> b.constructor  
function Object() { [native code] }  
> a.__proto__.constructor  
> function F1() {}  
> b.__proto__.constructor  
function Object() { [native code] }
```

Lookup

Mais où javascript cherche-t-il la valeur d'une propriété sur un objet ?

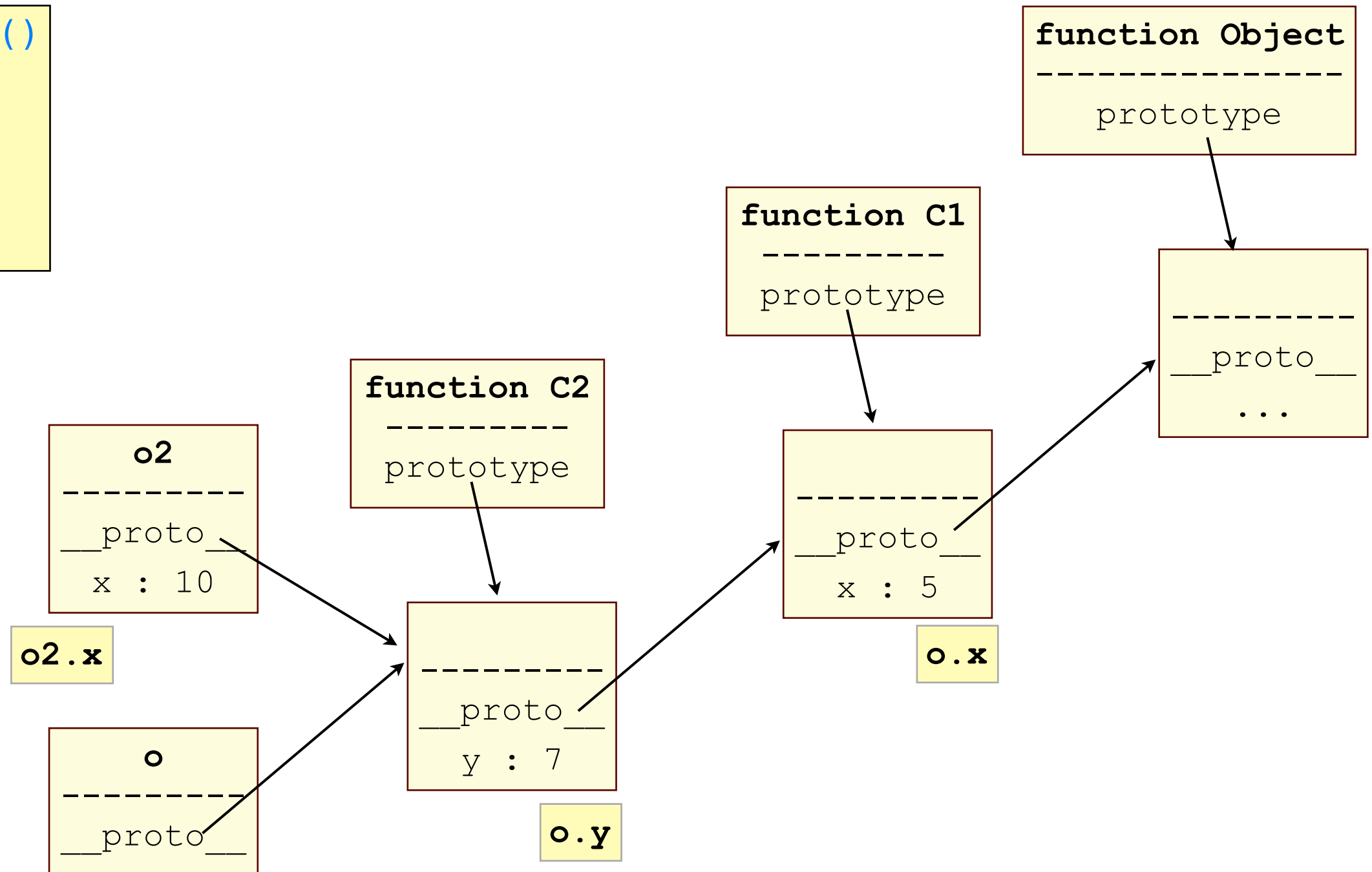
Sens du lookup

```
> function C1() {}  
> C1.prototype.x=5  
  
> function C2() {}  
> C2.prototype = new C1()  
C2.prototype.y=7  
  
> o=new C2()  
> o.x  
5  
> o.y  
7
```



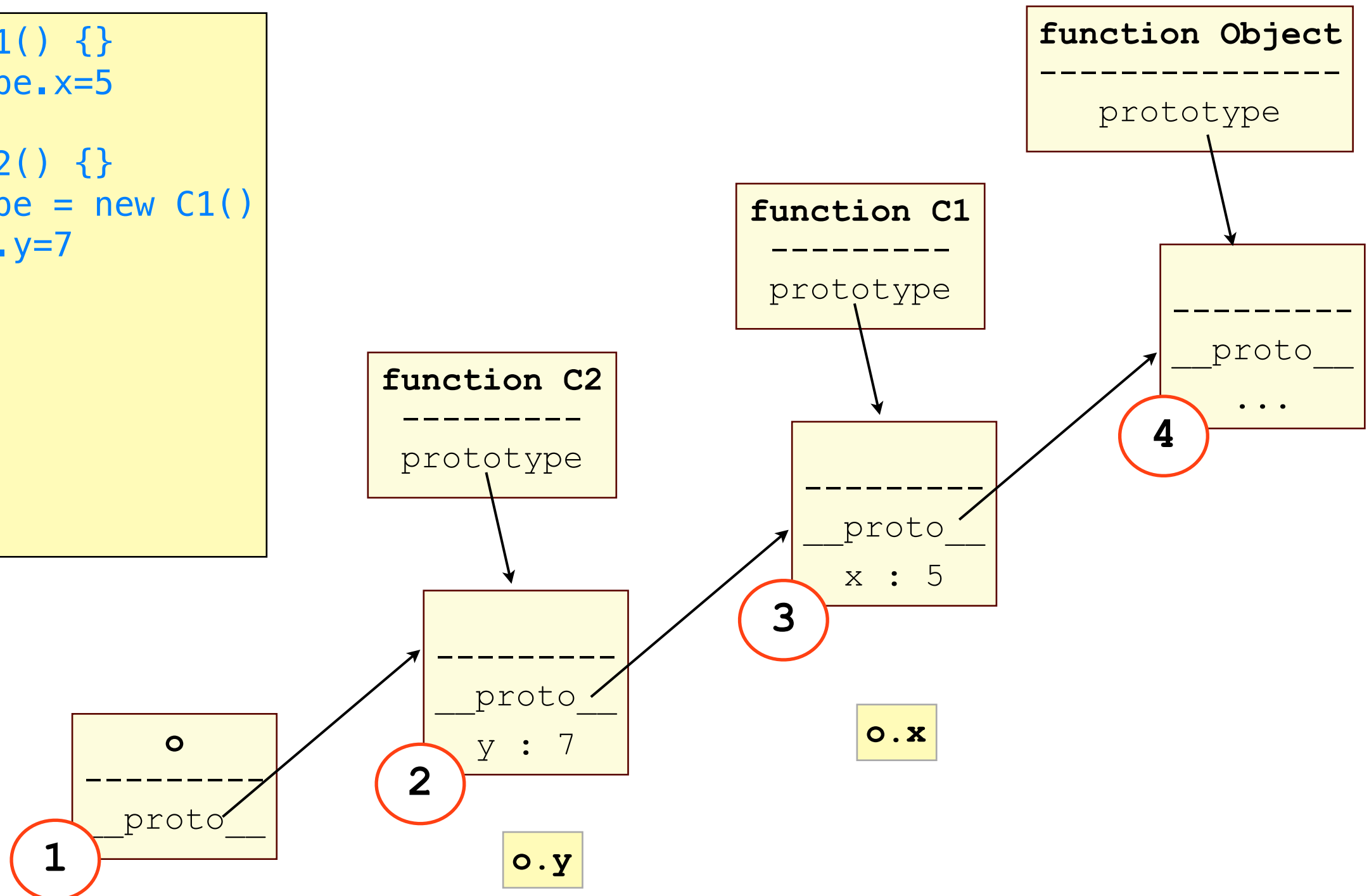
Et donc ...

```
> o2=new C2()  
> o2.x=10  
> o.x  
5  
> o2.x  
10
```



Oui : c'est bien de l'héritage made in javascript !

```
> function C1() {}  
> C1.prototype.x=5  
  
> function C2() {}  
> C2.prototype = new C1()  
C2.prototype.y=7  
  
> o=new C2()  
> o.x  
5  
> o.y  
7
```



Protéger les propriétés

```
> C2.prototype._x=3
> C2.prototype.__defineGetter__("x", function () { return this._x; })
> C2.prototype.__defineSetter__("x", function (value) { this._x=value; })
// accéder à la propriété à x se fera désormais par les 2 précédentes
// fonctions
> o.x
3
> o.x=7
7
> o._x
7
> o3=new C2()
> o3.x
3
// o a son propre _x d'où une valeur différente de o3 qui pointe
// sur le _x du prototype
```

Mais on peut voir le
_x !?!

Y a une astuce

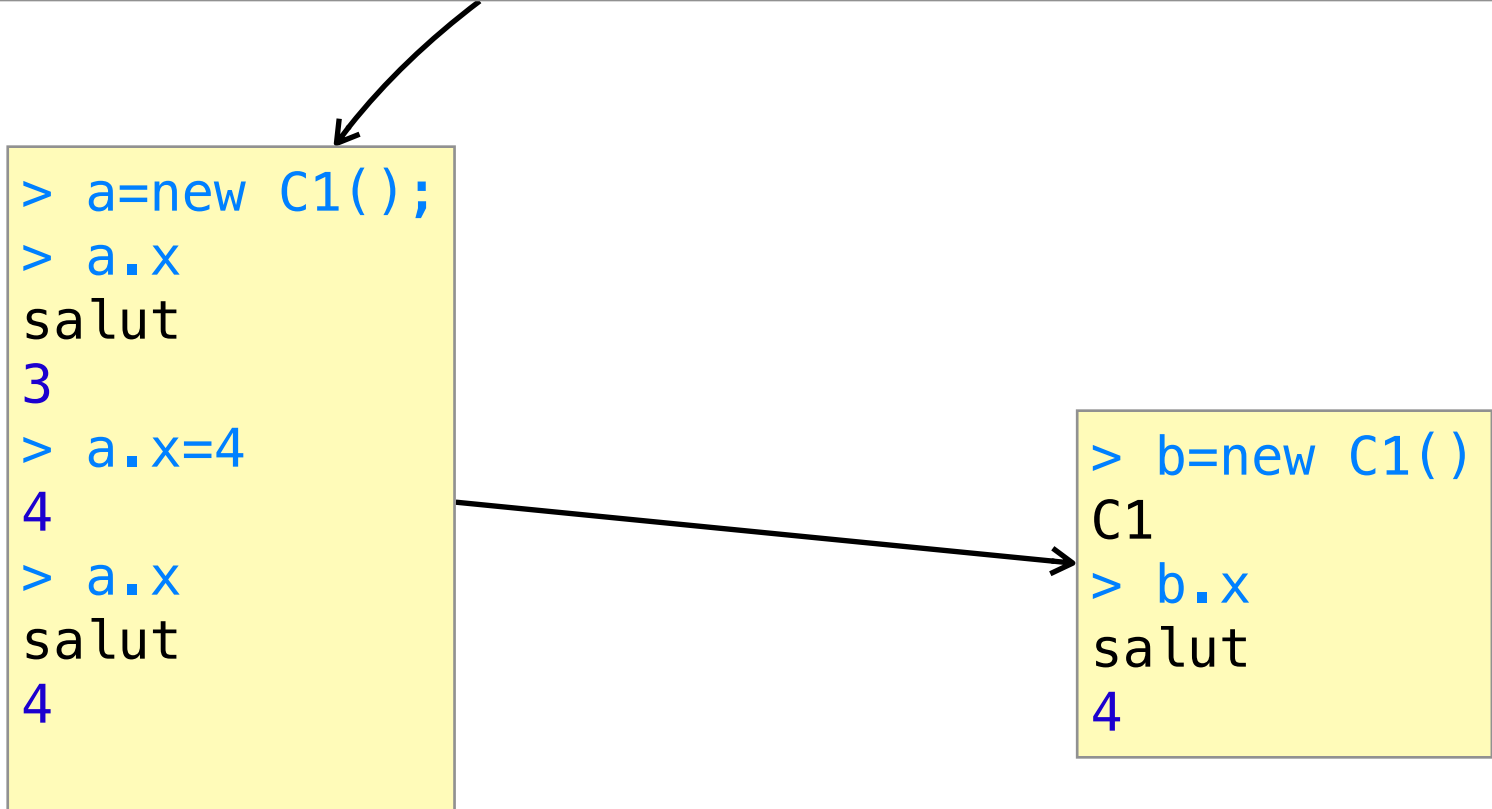
variable temporaire

```
> function C3 () {  
  var x = 3;  
  this.__defineGetter__("x", function () { console.log("salut");return x; });  
  this.__defineSetter__("x", function (value) { x=value; });  
}  
> a=new C3()  
C3  
> a.x  
salut  
3
```

Et pour des variables
communes comme celles du
prototype ?

Variable statique

```
> function C1() {}  
> C1.prototype= new function () {  
  var x = 3;  
  this.__defineGetter__("x", function () { console.log("salut");return x; });  
  this.__defineSetter__("x", function (value) { x=value; });  
}
```



```
> a=new C1();  
> a.x  
salut  
3  
> a.x=4  
4  
> a.x  
salut  
4
```

```
> b=new C1()  
C1  
> b.x  
salut  
4
```

__lookup__ et
hasOwnProperty

__lookup__

✖ Sur le précédent exemple

```
> a.__lookupGetter__("x")  
function () { console.log("salut");return x; }  
> a.__lookupSetter__("x")  
function (value) { x=value; }
```

Une sorte de get des getters/
setters pour un nom donné

hasOwnProperty

✱ Sur le précédent exemple

```
> a.hasOwnProperty("x")  
false  
> a.y=5  
5  
> a.hasOwnProperty("y")  
true
```

for (var key in object)

✱ Sur l'exemple précédent

```
> for (var i in a) { console.log(i);}
y
x
```

Un mix entre `hasOwnProperty` et `lookup`, moins les propriétés de base d'un objet.

Apply vs Call

Don't call me baby

Différents mais pareils

```
> function ajouteX(valeur) {this.x=valeur;}  
> a={};b={}  
> ajouteX.apply(a,[4]);  
> a.x  
4  
  
> ajouteX.call(b,6);  
> b.x  
6
```

Apply et call fixe le scope de la fonction avec le premier argument

Programmation générationnelle

Vive eval

Au lieu de

```
> C1.prototype= {  
  nom : "",  
  setNom : function (value) {  
    if (typeof(value)!="string")  
      throw "value must be a string";  
    this.nom=value;  
  },  
  prenom : "",  
  setPrenom : function (value) {  
    if (typeof(value)!="string")  
      throw "value must be a string";  
    this.prenom=value;  
  },  
  adresse : "",  
  setAdresse : function (value) {  
    if (typeof(value)!="string")  
      throw "value must be a string";  
    this.adresse=value;  
  }  
}
```

```
> a=new C1()  
C1  
> a.nom  
""  
> a.nom=3  
3  
> a.setNom(3)  
"value must be a string"  
> a.setNom("Le Pallec")  
undefined
```

On peut créer

```
> function addStringSetter(pty) {  
  var code = "var __f__=function (value) {\n";  
  code+="  if (typeof(value)!='string')\n";  
  code+="    throw 'value must be a string';\n";  
  code+="  this.\"+pty+\"=value;\n";  
  code+="}\n";  
  eval(code);  
  return __f__;  
}
```

Et donc

```
> C1.prototype= {  
  nom : "",  
  setNom : addStringSetter("nom"),  
  prenom : "",  
  setPrenom : addStringSetter("prenom"),  
  adresse : "",  
  setAdresse : addStringSetter("adresse")  
}  
  
> b=new C1()  
C1  
  
> b.setPrenom  
function (value) {  
  if (typeof(value)!='string')  
    throw 'value must be a string';  
  this.prenom=value;  
}
```

Mais eval c'est bof...

new Function

```
> function addStringSetter(pty) {  
  var code="  if (typeof(value)!='string')\n";  
  code+="    throw 'value must be a string';\n";  
  code+="  this.\"+pty+\"=value;\n";  
  return new Function("value",code);  
}
```

```
> function C2() {}  
> C2.prototype= {  
  nom : "",  
  setNom : addStringSetter("nom"),  
  prenom : "",  
  setPrenom : addStringSetter("prenom"),  
  adresse : "",  
  setAdresse : addStringSetter("adresse")  
}
```

```
> d=new C2()  
> d.setAdresse  
function anonymous(value) {  
  if (typeof(value)!='string')  
    throw 'value must be a string';  
  this.adresse=value;  
}
```

Imaginons

```
> function C1() {}  
> C1.prototype= new function () {  
  var nom;  
  this.__defineGetter__("nom", function () { return nom; });  
  this.__defineSetter__("nom", function (value) { nom=value; });  
  var prenom;  
  this.__defineGetter__("prenom", function () { return prenom; });  
  this.__defineSetter__("prenom", function (value) { prenom=value; });  
  var adresse;  
  this.__defineGetter__("adresse", function () { return adresse; });  
  this.__defineSetter__("adresse", function (value) { adresse=value; });  
  ...  
}
```

Comment on fait là ?

Un peu de génération

```
> function codeForVariable(pty) {  
  code="var "+pty+"\n";  
  code+="this.__defineGetter__ ('"+pty+"', function () { return "+pty+"; });\n";  
  code+="this.__defineSetter__ ('"+pty+"', function (value) { "+pty+"=value; });\n";  
  return code;  
}  
undefined  
> function C1 () {  
  eval(codeForVariable("nom"));  
  eval(codeForVariable("prenom"));  
  eval(codeForVariable("adresse"));  
}  
undefined  
> a=new C1()  
C1  
> a.__lookupGetter__("nom")  
function () { return nom; }
```

C'est mieux... mais on aura
préfééré une methode
addVariable

Une fonction qui englobe tout

```
> function addVariable (pty) {  
  eval (codeForVariable(pty));  
}  
  
> function C1 () {  
  addVariable("nom");  
  addVariable("prenom");  
  addVariable("adresse");  
}  
  
> a=new C1()  
C1  
> a.__lookupGetter__("nom")  
undefined  
  
> window.__lookupGetter__("adresse")  
function () { return adresse; }
```

Ça ne marche pas car le code est exécuté dans `addVariable` où le `this` vaut `window`

Avec un petit call, ça va mieux

```
> function C1 () {  
  addVariable.call(this,"nom");  
  addVariable.call(this,"prenom");  
  addVariable.call(this,"adresse");  
}  
  
> a=new C1()  
C1  
> a.__lookupGetter__("nom")  
function () { return nom; }
```

On fixe le scope d'addVariable
avec la méthode call

Et si on mettait le call dans
addVariable ?

Bof

```
> function addVariable (object,pty) {  
  eval.call(object,codeForVariable(pty));  
}  
  
> function C1 () {  
  addVariable(this,"nom");  
  addVariable(this,"prenom");  
  addVariable(this,"adresse");  
}  
  
> e=new C1()  
C1  
> e.__lookupGetter__("nom")  
undefined
```

Ça ne marche pas : `this` vaut
toujours `window`

Bind - ons !

```
> function addVariable (object,pty) {  
  eval(codeForVariable(pty));  
}  
  
> function C1 () {  
  var addVariable2=addVariable.bind(this);  
  addVariable2(this,"nom");  
  addVariable2(this,"prenom");  
  addVariable2(this,"adresse");  
}  
  
> f=new C1()  
  
> f.__lookupGetter__("adresse")  
function () { return adresse; }
```

addVariable2 aura toujours
pour scope le `this` passé en
paramètre de `bind` !

Comment fait-on un package ?

trop dur

Houlala

```
> p1 = {}  
Object  
  
> p1.C1 = function () {}  
function () {}  
  
> a=new p1.C1()  
p1.C1
```

Des questions ?