Compilation course 10 : Classes and Prototypes

Flavien BREUVART

April 2, 2026

History of JS classes: Starts as syntactic sugar hiding a constructor

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){return 128;}
}
new Maclasse();
```

$\simeq$

```
function new_MaClasse (){
  return {
    attr1 : 42,
    attr2 : undefined,
    meth1 : function (){
              return 128;
    }       }
}
new_MaClasse();
```

History of JS classes: Starts as syntactic sugar hiding a constructor

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){return 128;}
  constructor(x){
    this.attr2 = x;
    }
}
new MaClasse(64);
```

$\simeq$

```
function new_MaClasse (x){
  rez = {
    attr1 : 42,
    attr2 : undefined,
    meth1 : function (){
              return 128;
  }         }
  rez.attr2 = x;
  return rez;
}
new_MaClasse(64);
```

Consequence: environement binded to the point where the class is created

```
function f(x) {
  var y=42
  class MaClass{
    x = y;
    g() {return y++;}
  }
  return MaClass;
}
maclasse = f(42);
ob1 = new maclasse();
ob1.g();
ob2 = new maclasse();
ob1.x;
ob2.x;
```

Consequence: environement binded to the point where the class is created

```
function f(x) {
  var y=42
  class MaClass{
    x = y;
    g() {return y++;}
  }
  return MaClass;
}
maclasse = f(42);
ob1 = new maclasse();
ob1.g();
ob2 = new maclasse();
ob1.x;
ob2.x;
```

$\longrightarrow 42$

$\longrightarrow 43$

History of JS classes: Need for factorisation in classes whith many methodes

Slow initialisation

For each execution of the constructor, the closures are reinitialised.

Inneficient space management

Methodes closures are integrates in each objects.

Example: For chained lists, each node would contains more than 70 closures which have to be initialised at each insertion !

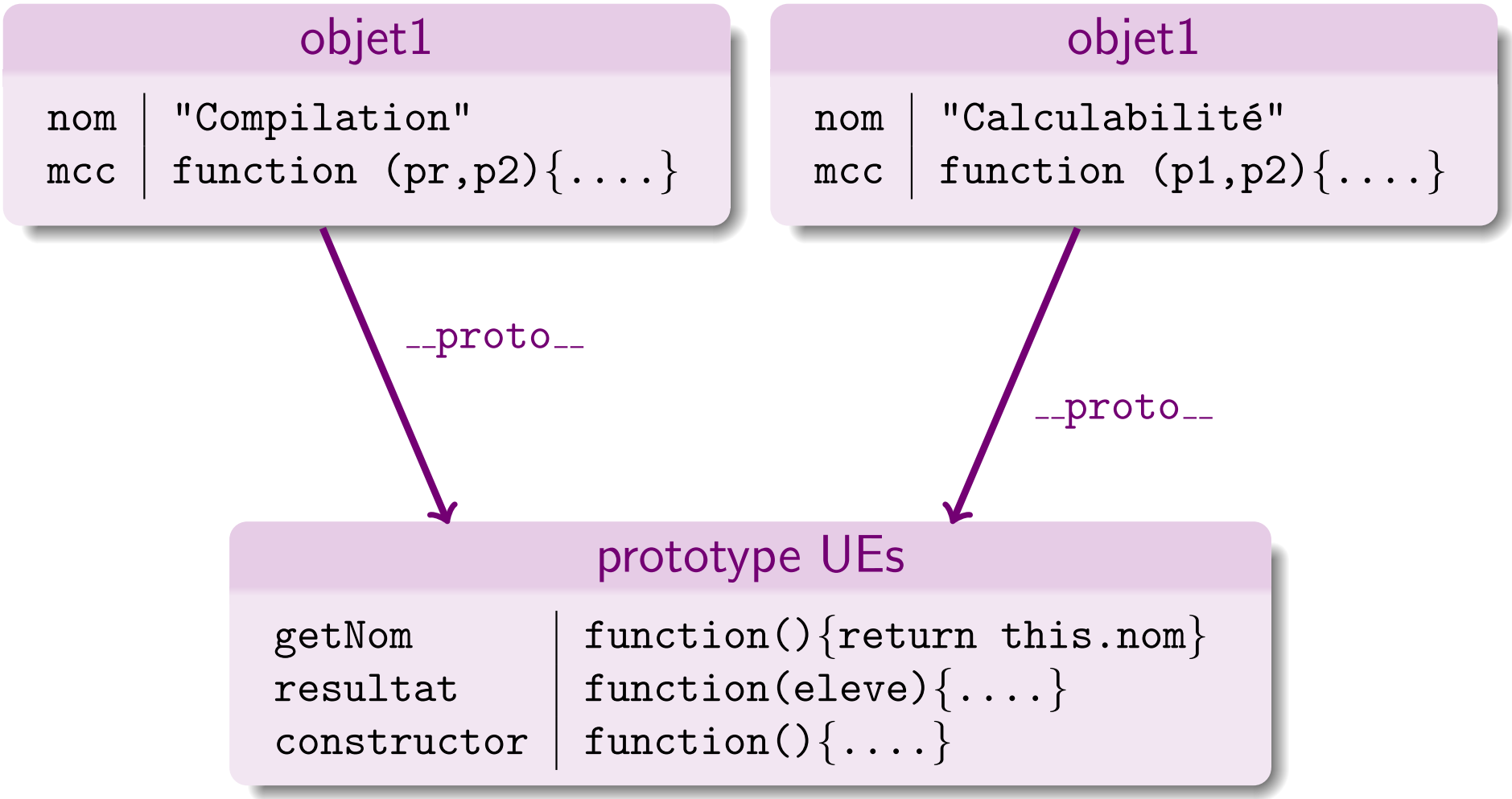
History of JS classes: Prototype containing a constructor

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){
    return 128;
  }
  meth2(){
    this.attr1++;
  }
}
new Maclasse();
```

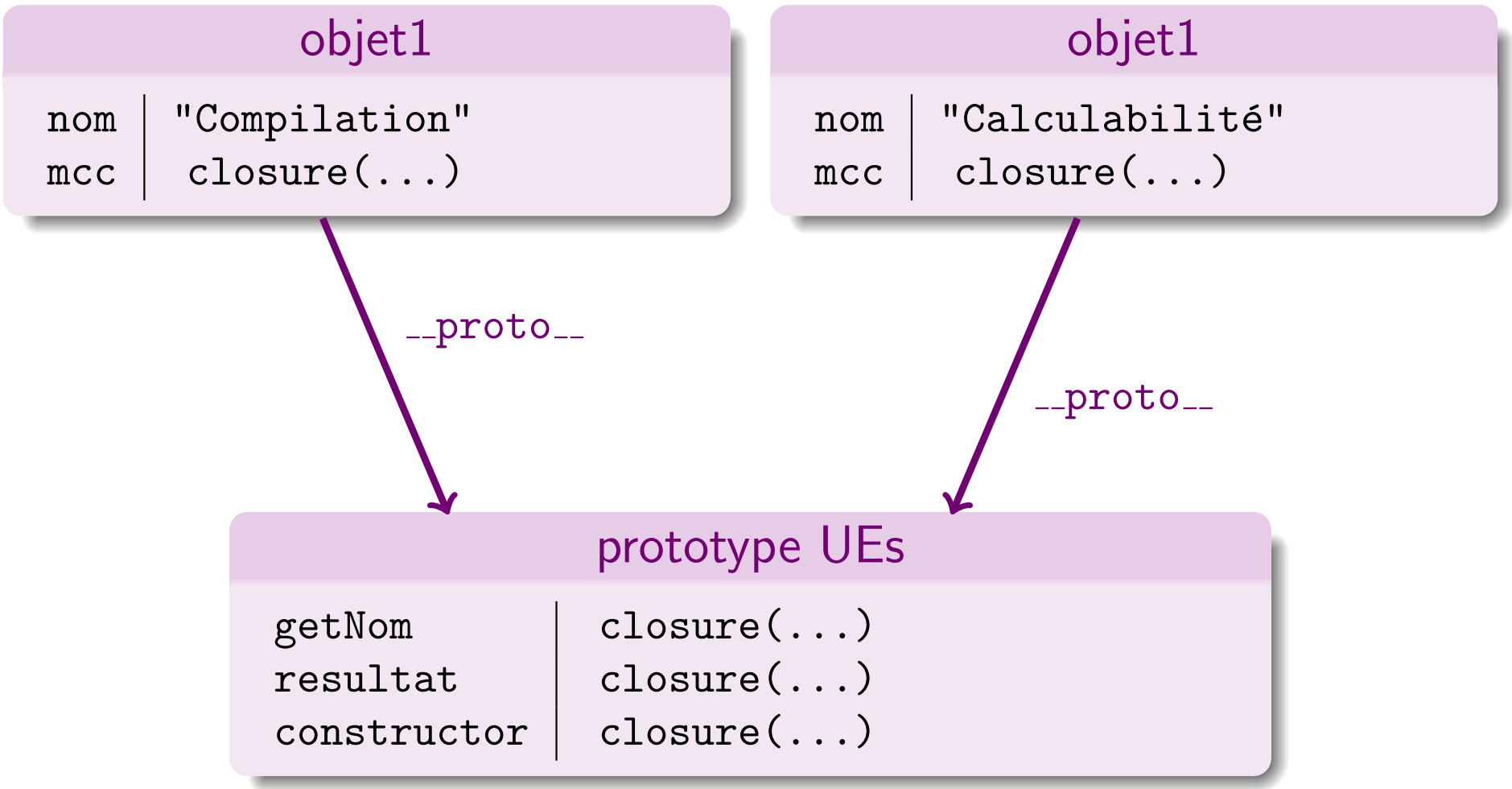
$\simeq$

```
MaClasse = {
  meth1 : function (){return 128;}
  meth2 : function (){this.attr1++;}
  constructor : function () {
    return {
      __proto__ : MaClasse,
      attr1 : 42,
      attr2 : undefined
    };
  }
}
MaClass.constructor();
```

JS object : attributs + prototype (⊇methodes)



JS object : attributs + prototype (⊇methodes)



Functional attributs and "static" attributs

JS allows functions as attribut

(In Java: objects of anonymus sub-class of that of function...)

JS (which is interpreted) allow for static atributs ?

The "static" keyword is an atribut of the class, not of the object (not evec the prototype),.
`(new MyClass).staticAttr` do not exist, you need to use `MyClass.staticAttr`.

Remark : prototypes only contain functions, but you can add by had a pseudo-static attribut there that are shared by all objecs but not accessible from the class... pretymuch useless...

Object wraper around classes (and functions)

In JS : a class est an objet

- an attribut `name` (not specified, not necessarily the name of the class...)
- un attribut `prototype`
- static attributs
- length (see functions)

so are function...

- an attribut `name` (not necessarily the name in the environement...)
- an invisible closure
- other secondary attributs
- length (arrity)

which prototype is that of the class of all functions.

... so that a class looks like a function

The prototype of a class is, in fact, that of functions...

The only objective is that of retrocompatibility (class = constructor)

Object Wraper arround classes (and functions)

Constant evolutions with each version of JS

- “typeof” on a classe and function givec “function”.
- howeover, firefox console diferenciate “class” and “function” types.

In the project (most advanced fragment)

- A class has to be ab object with a `prototype` attribut.
- Functions can remain closures (feel free to wrap them though).

With Wraper

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){
    return 128;
  }
  meth2(){
    this.attr1++;
  }
}
```

$\simeq$

```
MaClasse = {
  name : "MaClasse";
  prototype : {
    meth1 : function (){return 128;}
    meth2 : function (){this.attr1++;}
    constructor : function () {
      return {
        __proto__ : MaClasse.prototype,
        attr1 : 42,
        attr2 : undefined
      };
    }
  }
  __proto__ : prototype_des_fonctions;
}
```

With Wraper (and using this)

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){
    return 128;
  }
  meth2(){
    this.attr1++;
  }
}
```

$\simeq$

```
MaClasse = {
  name : "MaClasse";
  prototype : {
    meth1 : function (){return 128;}
    meth2 : function (){this.attr1++;}
    constructor : function () {
      return {
        __proto__ : this,
        attr1 : 42,
        attr2 : undefined
      };
    }
  }
  __proto__ : prototype_des_fonctions;
}
```

mini-JSMachine

```
class MaClasse {
  attr1 = 42;
  attr2;
  meth1(){
    return 128;
  }
  meth2(){
    this.attr1++;
  }
}
new MaClasse();
```

```
NewObj
NewObj
NewClo 14
SetObj meth1
NewClo 14
SetObj meth2
NewClo 19
SetObj constructor
SetObj prototype
SetVar MaClasse
GetVar MaClasse
GetObj prototype
FrmPrt
GetObj constructor
StCall
Call
Halt
```

```
#meth1
CsteNb 128
Return
#meth2
GetVal this
Copy
GetObj attr1
CsteNb 1
AddiNb
SetObj attr1
Return
# constructeur
GetVar this
CsteNb 42
SetObj attr1
CsteUn
SetObj attr2
Return
```

FrmPrt	Push(newObj{__proto__ := Pop})	o1:pile	o2:pile

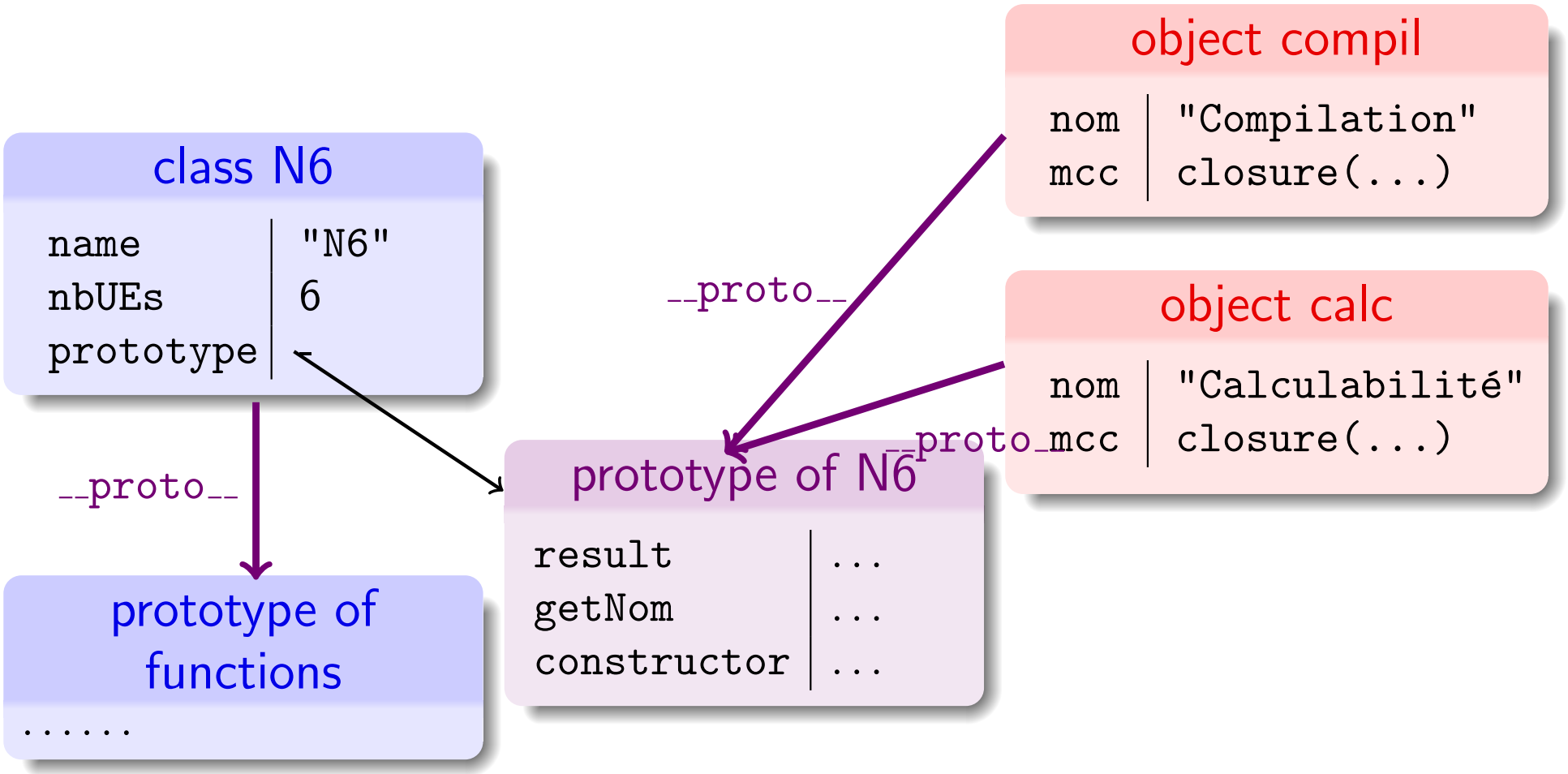
(optional) "this" in a constructor

```
class MaClasse {
  a;
  constructor(x){
    this.a=x;
  }
}
new MaClasse();
```

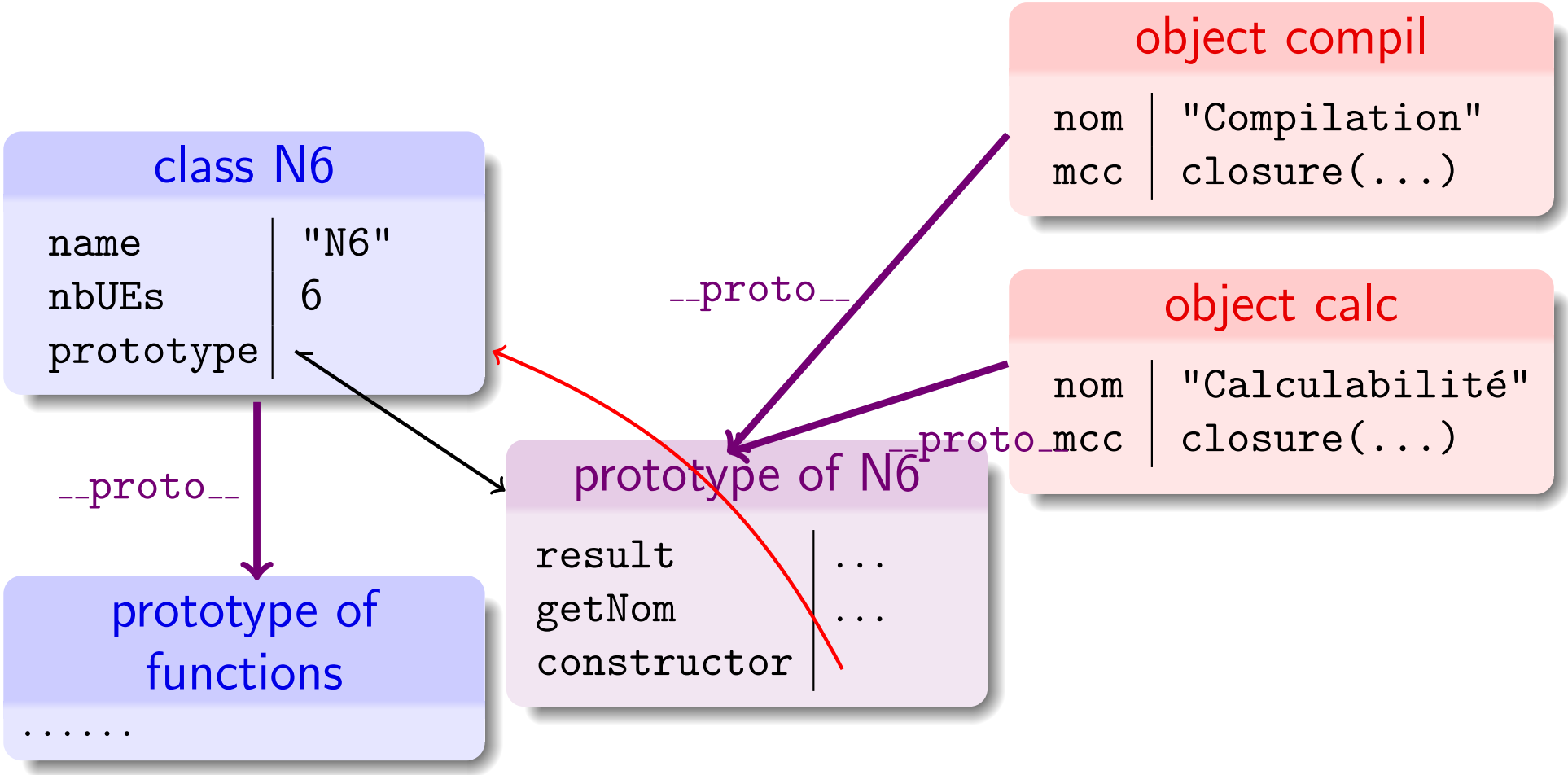
```
#MaClasse
NewObj
NewObj
NewClo 13
SetObj constructor
SetObj prototype
SetVar MaClasse
#Main
GetVar MaClasse
GetObj prototype
FrmPrt
Copy
GetObj constructor
StCall
Swap
SetIn this
Call
Halt
```

```
# constructeur
GetVar this
CsteUn
SetObj a
SetVaD this
GetVat this
GetVar x
SetObj a
Return
```

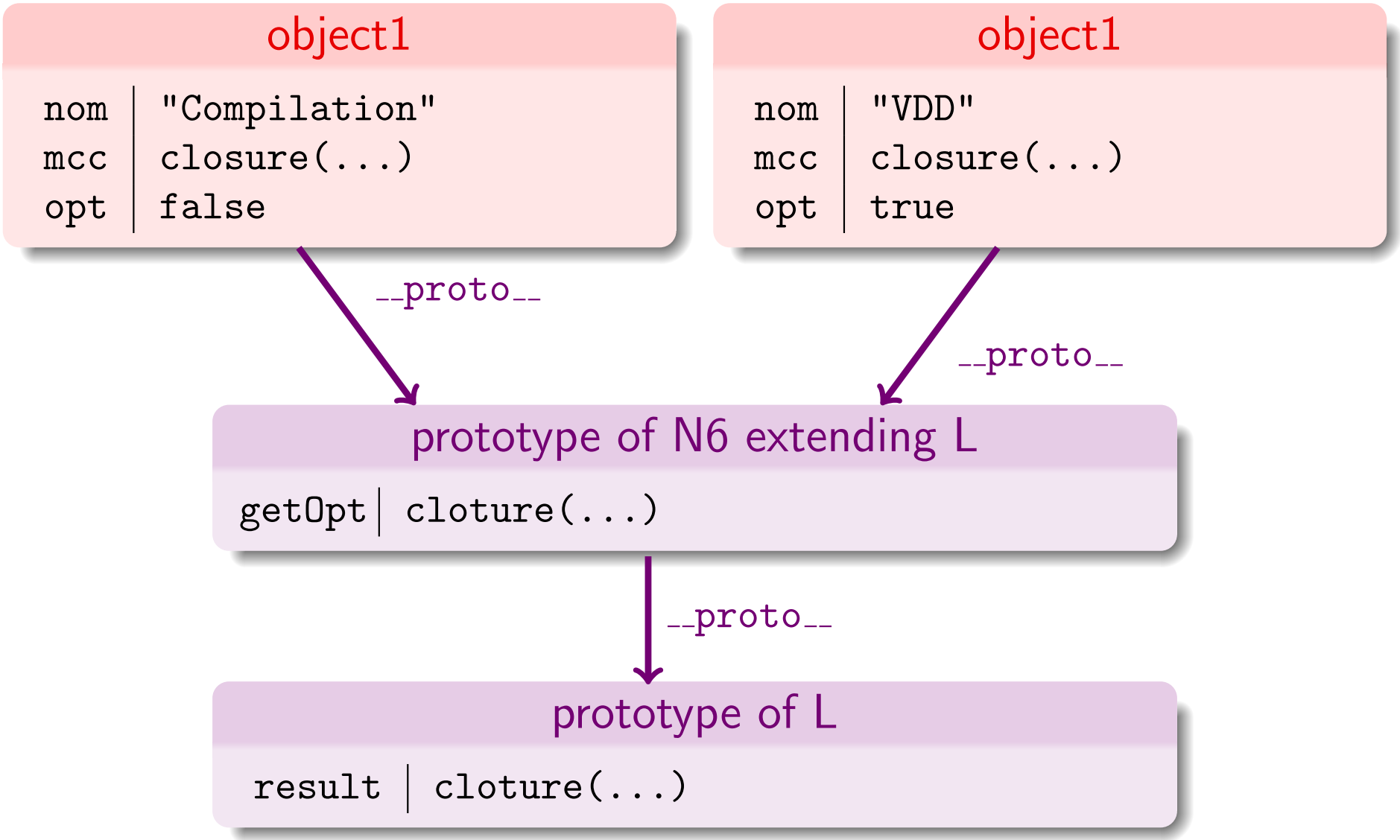
In picture


object compil
nom | "Compilation"
mcc | closure(...)
class N6
name | "N6"
nbUEs | 6
prototype
__proto__
object calc
nom | "Calculabilité"
mcc | closure(...)
__proto__
__proto__
prototype of N6
result | ...
getNom | ...
constructor | ...
prototype of functions
.....

In picture (reality)



Inheritance : Imbrication of prototypes



Inheritance : Imbrication of prototypes

```
class Foo extends Bar {
  a;
  constructor(x){
    this.a=x;
  }
}
```

$\simeq$

```
Foo = {
  name : "Foo";
  prototype : {
    constructor : function () {
      __rez = this.__proto__.constructo
      __rez.a = 42;
      return __rez;
    },
    __proto__ : Bar.prototype
  }
  __proto__ : prototype_des_fonctions;
}
```

Inheritance : (using super)
Imbrication of prototypes

```
class Foo extends Bar {
  a;
  constructor(x){
    this.a=x;
  }
}
```

$\simeq$

```
Foo = {
  name : "Foo";
  prototype : {
    constructor : function () {
      __rez = super.constructor();
      __rez.a = 42;
      return __rez;
    },
    __proto__ : Bar.prototype
  }
  __proto__ : prototype_des_fonctions;
}
```

JSMachine

```
class Foo extends Bar {
  a;
  constructor(x){
    this.a=x;
  }
}
```

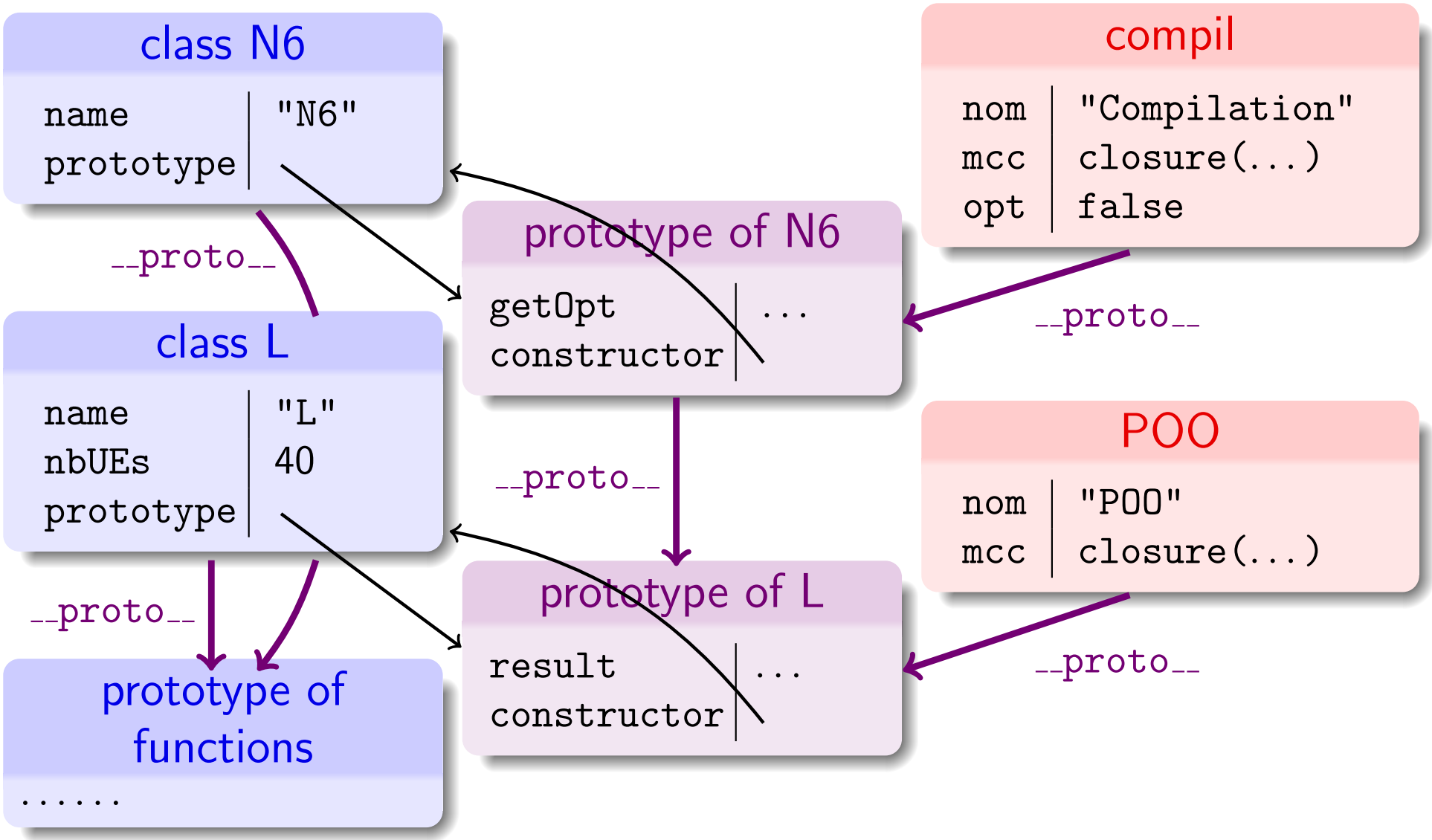
```
#class Foo
NewObj
GetVar Bar
GetObj prototype
FrmPrt
NewClo 4
SetObj constructor
SetObj prototype
SetVar Foo
Halt
```

```
# constructeur
GetVar this
GetPrt
GetPrt
GetObj constructor
StCall
GetVar this
SetIn  this
Call
GetVar x
SetObj a
Return
```

Attention au constructeur

Le constructeur commence par un appel au constructeur de la super-classe.

In picture


class N6
name
"N6"
prototype
__proto__
class L
name
"L"
nbUEs
40
prototype
__proto__
prototype of functions
.....
prototype of N6
getOpt
...
constructor
__proto__
prototype of L
result
...
constructor
compil
nom
"Compilation"
mcc
closure(...)
opt
false
__proto__
POO
nom
"POO"
mcc
closure(...)
__proto__

"GetVar" and "GetObj" are similar

GetVar x

- looks for variable `x` in CC (local env.)
- if unfound goes up to the more global one
- if unfound at the root, returns undefined.

GetObj x

- looks for variable `x` in the attributs,
- if unfound goes up to the prototype
- if unfound at the root, returns undefined.

"GetVar" and "GetObj" are slightly different

SetVar x

- looks for variable `x` in CC (local env.)
- if found modify it,
- if unfound goes up to the more global one,
- if unfound at the root, insert it there.

SetObj x

- looks for variable `x` in the attributs,
- if found modify it otherwise insert it.
- But never goes up the prototypes.

Can we statically know the position of a value in a contexte/object ?

The "get" of a dictionarry is less efficient that vector access.

Scope (sementic) analysis : knowing the context

Declared variables, and their order, can be known statically.

Objects attributs and methodes are of dynamics nature

We can't statically know all the fields and methodes of an object, but we can know a subset of them and maybe their position (mono inheritence).

This is the idea of types in Java.

Possible Implementations of dictionnairies and practical choice

Discution depending on the time

Disclaimer :

These slides (and this cours) do not have the objective of presenting the formal sementics of JS, this is just an analysis of transversal compilation-related conscepts based on JS examples.

For the official JS sementics, see ECMA norm:
https://262.ecma-international.org/