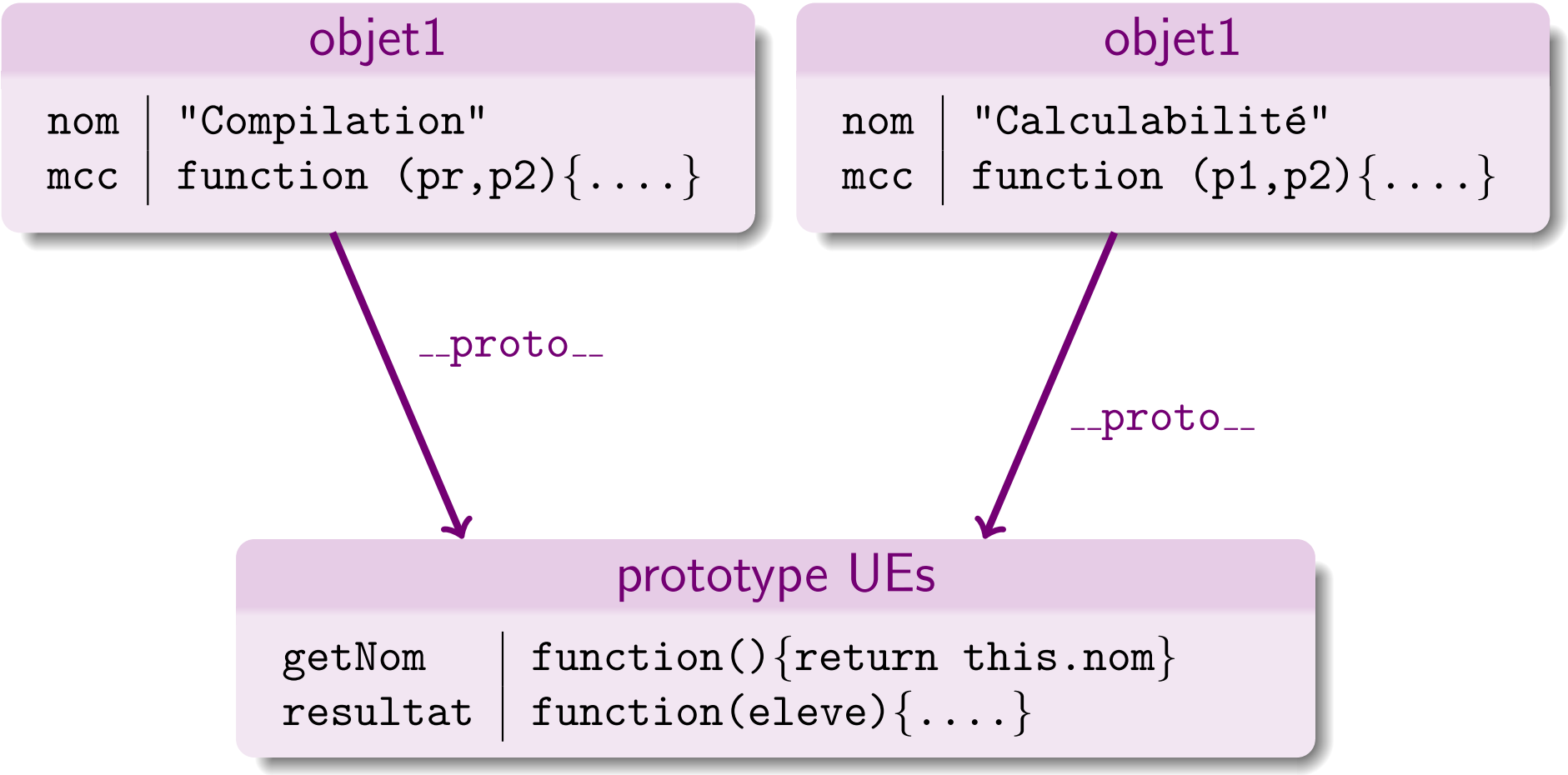
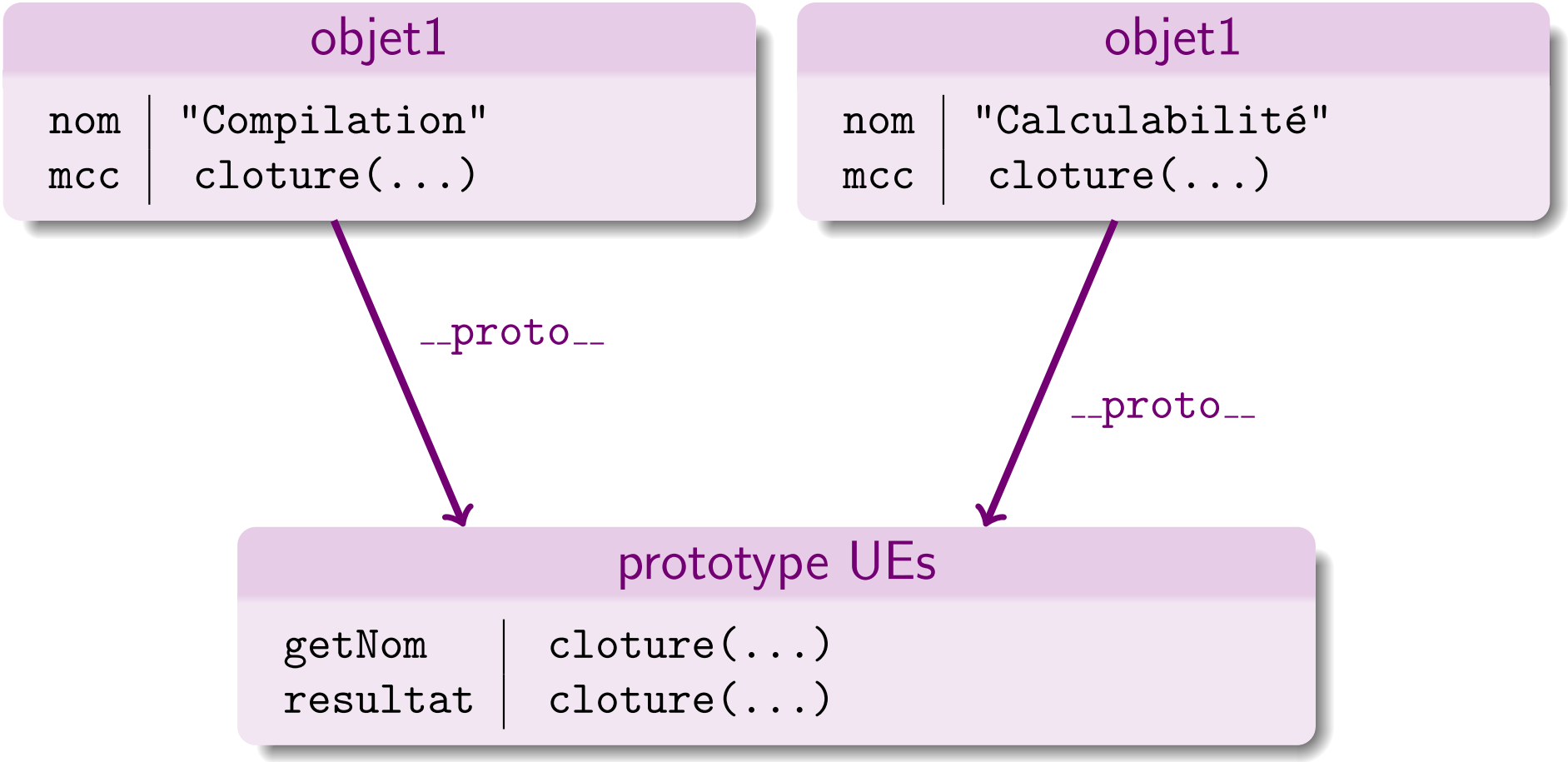
Objet JS : attributs + prototype (⊇méthodes)



Objet JS : attributs + prototype (⊇méthodes)



Fonction attribut et attribut statique

On peut avoir une fonction comme attribut

Impossible en Java, où on doit utiliser des sous-classes anonymes.

Et les attributs statiques ?

Le mot clé "static" déclare l'attribut dans la classe et non dans l'objet, ni même dans le prototype.
`(new MaClasse).attrStatic` est indéfini, il faut utiliser `MaClasse.attrStatic`.

Remarque : normalement le prototype ne contient que des fonctions, mais rien de vous empèche de le modifier... ce qui serait très sale...

Wraper objet autours des classes/fonctions

En JS : une classe est un objet

- un attribut `nom` (non spécifié, pas forcément le nom de a classe...)
- un attribut `prototype` (notion définie précédement)
- potentielement d'autres champs

une fonction aussi...

- un attribut `nom` (pas forcément le nom dans votre environement...)
- une clôture invisible
- potentielement d'autres champs

avec un lien vers le prototype des fonctions.

... de manière à ce qu'une classe ressemble une fonction

Le prototype d'une classe est le même que celui de la fonction.

Le but est uniquement la rétrocompatibilité (et la non-exposition des clôtres).

Wraper objet autours des classes/fonctions

Notion en constante évolution

- "typeof" sur une classe et une fonction rend "function".
- la console firefox indique "class" et "function" correctement.

Dans le projet

- Une classe doit être un objet avec un champ `prototype`.
- Les fonctions restent des clôtures.

L'attribut `nom` n'est pas obligatoire mais utile pour débugguer.
De même que l'attribut `length` qui contient l'arrité du constructeur.

Avec 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;
}
```

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(objet{__proto__ := Pop})	o1:pile	o2:pile

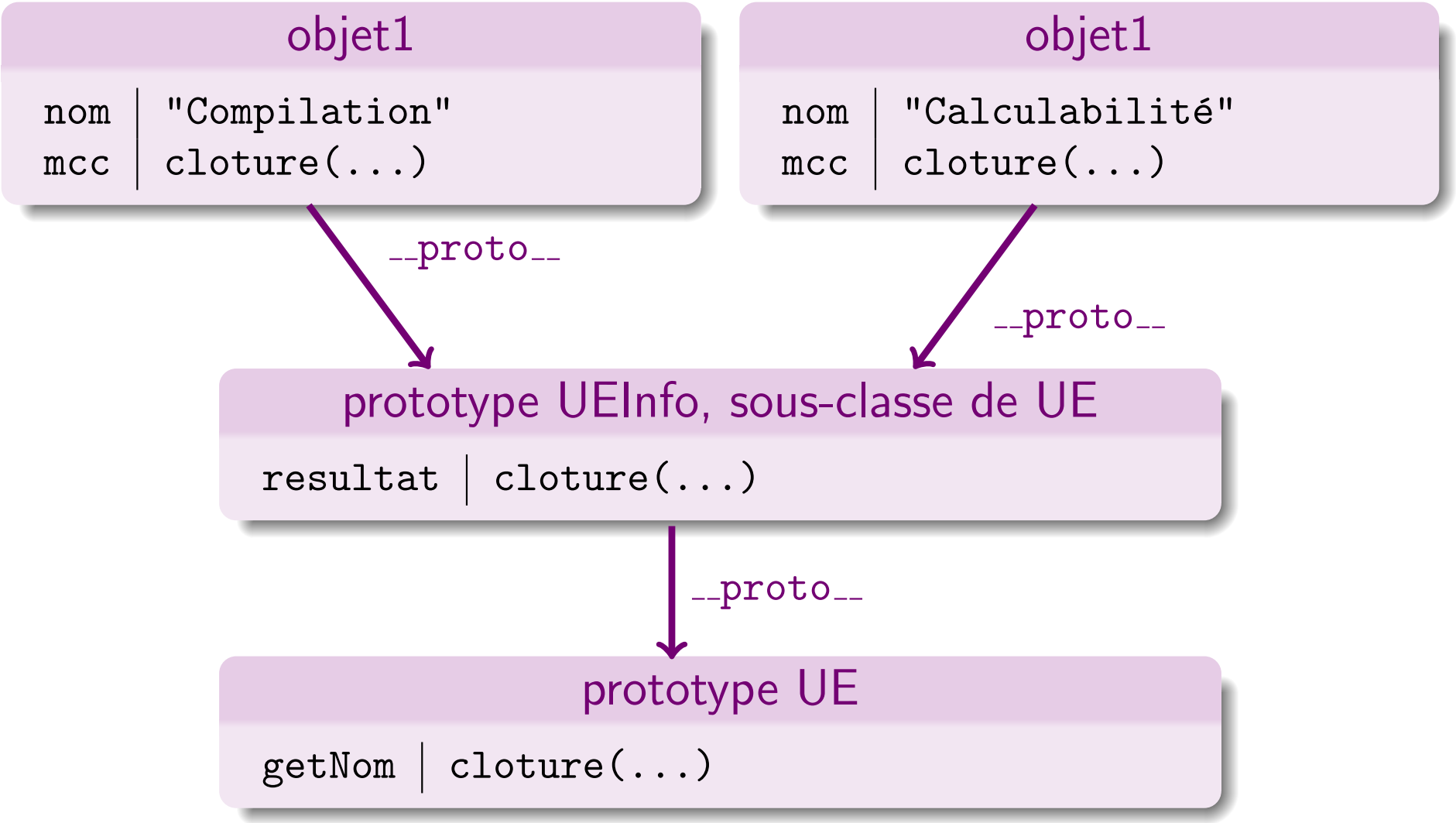
Le problème du this dans un constructeur

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

```
#MaClasse
NewObj
NewObj
NewClo 4
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
```

Héritage : Imbrication des Prototypes



Héritage : Imbrication des Prototypes

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

$\simeq$

```
Foo = {
  name : ``Foo'';
  prototype : {
    constructor : function () {
      return {
        a : 42,
        __proto__ : Foo.prototype
      };
    },
    __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.

Comportement par défaut des "get" iddentiques

GetVar x

- cherche la variable `x` dans l'environement locale
- si non trouvée remonte au niveau + global
- si non trouvé à la racine rend undefined.

GetObj x

- cherche l'attribut/methode localement
- si non trouvé remonte le proto
- si non trouvé à la racine rend undefined.

Comportement par défaut des "set" différents

SetVar x

- cherche la variable `x` dans l'environement locale
- si non trouvée remonte au niveau + global,
- si non trouvé insert à la racine,
- si trouvé modifie la version trouvée.

SetObj x

- cherche l'attribut/methode localement,
- si trouvé modifie la version trouvée,
- si non trouvé insert à l'origine,
- ne remonte jamais les niveaux de prototypes

Peut-on connaître le id du contexte/objet statiquement ?

L'analyse sémentique de portée : connaître le contexte

Les variables déclarées (càd les entrés du contextes) peuvent être connues statiquement.

les champs d'un objects sont dynamique

On ne peut pas connaître statiquement les champs d'un objet, mais on peut on deviner un sous ensemble.

C'est leprincipe de type en Java.

Implémentations possibles des dictionnaires et Choix en pratique

Discution selon le temps

Disclaimer :

Ces slides (et ce cours) n'ont pas pour but de présenter la sémentique officiel de JS, mais une présentation pédagogique de consceptes transversaux en s'appuiant sur JS.

Pour connaitre la sémentique officielle de JS, voir la norme ECMA : https://262.ecma-international.org/

Mensonge 1 : l'opérateur && rend un résultat non casté

En JS, 0&&3 rends 0 et non false.

Le vrais code de e1&&e2 serait plutôt

```
    [[e1]]
    Copy
    TypeOf
    Case
    Jump 1
    NbToBo
    CondJmp end
    Drop
    [[e2]]
end:...
```

Mensonge 2 : this vs context

Le contexte global est un objet

Plus précisément c'est une partie de l'objet `Window`, votre fenêtre.

Le "this" est l'objet courant

Par défaut c'est cet objet `Window`...

Mensonge 3 : classe vs constructeur

Dans le projet `new`≃`FrmPrt`

On lit `[new MaClasse]()`, où `new MaClasse` est un objet vide obtenu en appliquant `FrmPrt` à `MaClasse`

En JS (`new MaClasse()`)≃`MaClasse()`

On lit `new [MaClasse()]`, où `new` ne fait rien et la classe `MaClasse` est confondu avec son constructeur.
En particulier, on a l'égalité :

```
MaClass.prototype.constructor  ===  MaClass
```