Université de Paris

Objets : structurel (OCaml) vs. nominal (Java) vs. prototypes (JS)

Réalisé par :
Dorothée HUYNH
Haifa BEN HMIDA EL ABRI

Introduction

- **Typage structurel :**
  - La flexibilité.
- **Typage nominal :**
  - Une meilleure sécurité de typage.
- **Prototypage :**
  - La précision et la rapidité.

Les objets en Java

- Les classes et les instances
- Les constructeurs
- Les membres statiques
- La classe principale Object
- Le mot-clé this
- L'héritage
- L'abstraction

Les classes et les instances

```
public class Personne {
        int age;
        String nom;
        public getNom(){
                return nom;
        }
        public changeNom(String nom){
                this.nom = nom;
        }
}
```

```
public class CreatePersonne {
        public static void main (String[] args){
                Personne p  = new Personne();
                p.changeNom(“Martin”);
        }
}
```

Les constructeurs

```
public class Personne {
        int age;
        String nom;
        public Personne(int age, String nom){
                this.age =age;
                this.nom =nom;
        }
}
```

Les membres statiques

```
public class Femme extends Personne {

        static float taille = 70.9;

        public Femme(){}

}
```

La classe principale Object

```
public class Personne extends Object {
        int age;
        String nom;
        public getNom(){
                return nom;
        }
        public setNom(String nom){
                this.nom = nom;
        }
}
```

Le mot clé “this”

```
public class Personne {
        int age;
        public AgePersonne(int age){
                this.age=age;
        }
        Personne * RetournePersonne(){
                return this;
        }
}
```

L'héritage

```
public class Femme extends Personne {
        public String chanter(){
                return “la la la….”;
        }
}
```

```
public class CreatePersonne {
        public static void main (String[] args){
                Femme f = new Femme();
                f.changeNom(“Camille”);
                System.out.println(f.chanter());
        }
}
```

L'abstraction

```
abstract class Personne{

        public abstract void chanter();

}

Personne p = new Personne();   // Erreur
```

```
interface Personne_Interf {
        void chanter();
}

interface Personne_Interf extends another_interf{
        void parler();
}

class Personne implements another_interf{
        void chanter(){
                System.out.println("la la la ..");
        };
}
```

Les objets en Ocaml

- Les objets
- Les classes
- Self
- Les initialisateurs
- Les méthodes virtuelles
- Les méthodes privées
- L'héritage
- Les méthodes binaires et les amis

Les objets

Mutable

```
# let p =
    object
      val mutable x = 0
      method get_x = x
      method move d = x <- x + d
    end;;
```

```
val p : < get_x : int; move : int -> unit > = <obj>
```

```
# p#get_x;;
- : int = 0
# p#move 3;;
- : unit = ()
# p#get_x;;
- : int = 3
```

Les objets

Immutable

```
# let imm_stack init =
    object
      val v = init

      method pop =
        match v with
        | hd :: tl -> Some (hd, {< v = tl >})
        | [] -> None

      method push hd = {< v = hd :: v >}
      method get = v
    end;;
```

```
val imm_stack :
  'a list -> (< pop : ('a * 'b) option; push : 'a -> 'b > as 'b) = <fun>
```

```
# let is = imm_stack[5;7;9];;
val is : < get : int list; pop : (int * 'a) option;
push : int -> 'a > as 'a =
  <obj>
# is#pop;;
- : (int * (< get : int list; pop : 'a; push : int ->
'b > as 'b)) option
    as 'a
= Some (5, <obj>)
# match is#pop with | Some(v, im) ->
Some(v, im#get) | None -> None;;
- : (int * int list) option = Some (5, [7; 9])
# (is#push 3)#get;;
- : int list = [3; 5; 7; 9]
# is#get;;
- : int list = [5; 7; 9]
```

Les classes

```
# class point =
   object
     val mutable x = 0
          method get_x = x
     method move d = x <- x + d
   end;;
```

```
class point :
  object val mutable x : int method get_x : int method move : int -> unit end
```

```
# let p = new point;;
val p : point = <obj>
# p#get_x;;
- : int = 0
# p#move 3;;
- : unit = ()
# p#get_x;;
- : int = 3
```

Self

```
# class printable_point x_init =
   object (self)
        val mutable x = x_init
        method get_x = x
        method move d = x <- x + d
        method print =    print_int self#get_x
   end;;
```

```
class printable_point :
  int ->
  object
    val mutable x : int
    method get_x : int
    method move : int -> unit
    method print : unit
  end
```

```
# let p = new printable_point 7;;
val p : printable_point = <obj>

# p#print;;
7- : unit = ()
```

Les initialisateurs

```
#class printable_point x_init =
        object (self)
            val mutable x = x_init
            method get_x = x
            method move d = x <- x + d
            method print = print_int self#get_x; print_newline ()
            initializer print_string "new point at "; self#print
        end;;
```

```
class printable_point :
 int ->
 object
  val mutable x : int
  method get_x : int
  method move : int -> unit
  method print : unit
 end
```

```
# let p2 = new printable_point 17;;
new point at 17
val p2 : printable_point = <obj>
# p2#move 5;;
- : unit = ()
# p2#print;;
22
- : unit = ()
# p2#get_x;;
- : int = 22
```

Les méthodes virtuelles

```
# class virtual abstract_point x_init =
  object (self)
    val mutable virtual x : int
    method virtual get_x : int
    method get_offset = self#get_x - x_init
    method virtual move : int -> unit
  end;;
```

```
class virtual abstract_point :
  int ->
  object
    val mutable virtual x : int
    method get_offset : int
    method virtual get_x : int
    method virtual move : int -> unit
  end
```

```
# class point x_init =
  object
    inherit abstract_point x_init
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
  end;;
```

```
class point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

```
# let abspoint = new abstract_point 5;;
Error: Cannot instantiate the virtual
class abstract_point
# let p = new point 5;;
val p : point = <obj>
# p#get;;
Error: This expression has type point
       It has no method get
# p#get_x;;
- : int = 5
# p#move 7;;
- : unit = ()
# p#get_x;;
- : int = 12
# p#get_offset;;
- : int = 7
```

Les méthodes privées

```
# class restricted_point x_init =
  object (self)
    val mutable x = x_init
    method get = x
    method private pmove d = x <- x + d
    method move d = self#pmove d
  end;;
```

```
class restricted_point :
  int ->
  object
    val mutable x : int
    method get_x : int
    method move : int -> unit
    method private pmove : int -> unit
  end
```

```
# let p = new restricted_point 0;;
val p : restricted_point = <obj>

# p#pmove 5;;
Error: This expression has type restricted_point
       It has no method pmove
Hint: Did you mean move?

# p#move 5;;
- : unit = ()

# p#get_x;;
- : int = 5
```

L'héritage

```
class ['a] stack init = object
 val mutable v : 'a list = init

 method get = v

 method pop =
  match v with
  | hd :: tl ->
    v <- tl;
    Some hd
  | [] -> None

 method push hd =
  v <- hd :: v
end;;
```

```
class double_stack init = object
 inherit [int]  stack init as super

 method push hd =
  super#push (hd * 2)
end;;
```

```
class ['a] stack :
 'a list ->
 object
  val mutable v : 'a list
  method get : 'a list
  method pop : 'a option
  method push : 'a -> unit
 end

class double_stack :
 int list ->
 object
  val mutable v : int list
  method get : int list
  method pop : int option
  method push : int -> unit
 end
```

L'héritage

```
class ['a] stack init = object
 val mutable v : 'a list = init

 method get = v

 method pop =
  match v with
  | hd :: tl ->
    v <- tl;
    Some hd
  | [] -> None

 method push hd =
  v <- hd :: v
end;;
```

```
class double_stack init = object
 inherit [int]  stack init as super

 method push hd =
  super#push (hd * 2)
end;;
```

```
# let st = new stack[3;5;7];;
val st : int stack = <obj>
# st#get;;
- : int list = [3; 5; 7]

# let st2 = new stack [(1,1);(2,3);(5,0)];;
val st2 : (int * int) stack = <obj>
# st2#get;;
- : (int * int) list = [(1, 1); (2, 3); (5, 0)]

# let ds = new double_stack [2;3];;
val ds : double_stack = <obj>
# ds#get;;
- : int list = [2; 3]
# ds#push 4;;
- : unit = ()
# ds#get;;
- : int list = [8; 2; 3]
# ds#pop;;
- : int option = Some 8
# ds#get;;
- : int list = [2; 3]
```

L'héritage et les méthodes privées

```
# class restricted_point x_init =
  object (self)
   val mutable x = x_init
   method get = x
   method private pmove d = x <- x + d
   method move d = self#pmove d
  end;;
```

```
class restricted_point :
 int ->
 object
  val mutable x : int
  method get_x : int
  method move : int -> unit
  method private pmove : int -> unit
 end
```

```
# class point_again x =
  object (self)
   inherit restricted_point x
   method virtual pmove : _
  end;;
```

```
class point_again :
 int ->
 object
  val mutable x : int
  method get : int
  method move : int -> unit
  method pmove : int -> unit
 end
```

```
# let p = new point_again 8;;
val p : point_again = <obj>

# p#pmove;;
- : int -> unit = <fun>

# p#get;;
- : int = 8

# p#pmove 6;;
- : unit = ()

# p#get;;
- : int = 14
```

Les méthodes binaires

```
# class square w = object(self : 'self)
  method width = w
  method area = Float.of_int (self#width *   self#width)
  method equals (other : 'self) = other#width = self#width
end;;
```

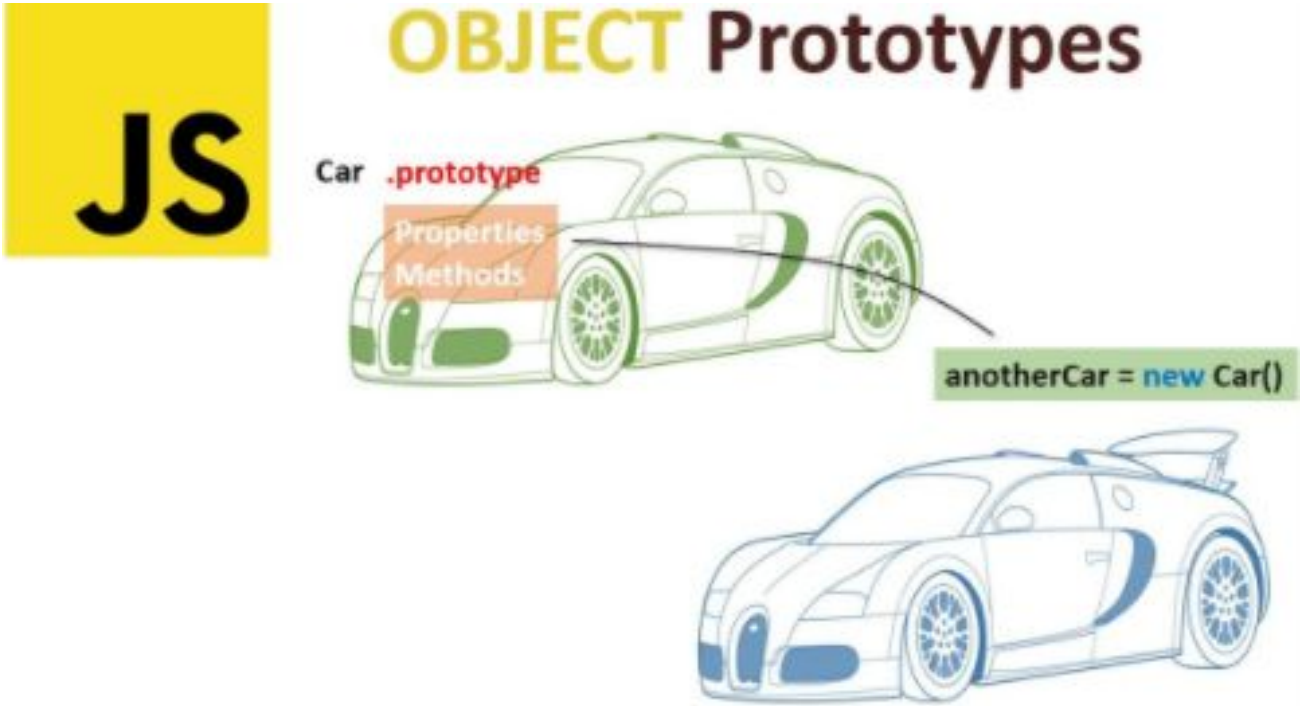
```
class square :
  int ->
  object ('a)
    method area : float
    method equals : 'a -> bool
    method width : int
  end
```

```
# let sq = new square 6;;
val sq : square = <obj>
# sq#area;;
- : float = 36.
# sq#width;;
- : int = 6
# let sq2 = new square 7;;
val sq2 : square = <obj>
# sq#equals sq2;;
- : bool = false
# let sq3 = new square 6;;
val sq3 : square = <obj>
# sq#equals sq3;;
- : bool = true
# (new square 5)#equals (new square 5);;
- : bool = true
```

Les objets en JavaScript

- Les prototypes objets
- Les “classes” en JavaScript
- Le mot-clé this
- L’héritage
- Les objets de portée globale (Global objects)

Les prototypes Objets



Les “classes” en Javascript

```
class Personne {
   constructor(firstname, lastname) {
      this.firstname = firstname;
      this.lastname= lastname;
   }
}
```

```
function Personne {
      this.firstname = firstname;
      this.lastname= lastname;
}
```

Le mot clé “this”

```
var personne = {
        firstName: "Camille",
        lastName : "Albane",

        fullName : function() {
          return this.firstName + " " + this.lastName;
        }
};
```

L'héritage

```
function Personne(prenom, nom, age) {
 this.nom = {prenom, nom};
 this.age = age;
 Personne.prototype.presenter = function() {
  console.log('Salut! Je suis ' + this.nom.prenom + ' ' +
this.nom.nom + ' et j'ai ' + this.age + ' ans.');
};
};
function Professeur(prenom, nom, age, matiere) {
 Personne.call(this, prenom, nom, age);
 this.matiere = matiere;
 Professeur.prototype = Object.create(Personne.prototype);
 Professeur.prototype.constructor = Professeur;
}
Professeur.prototype.presenter = function() {
  Personne.prototype.presenter.call(this);
  console.log('J'enseigne la matière ' + this.matiere);}
```

```
> let nina = new Personne("Nina", "Dupont", 23);
< undefined
> nina.presenter();
[Log] Salut! Je suis Nina Dupont et j'ai 23 ans.
< undefined
> mary = new Professeur("Mary", "Wilson", 42,
"anglais");
< Professeur {nom: {prenom: "Mary", nom:
"Wilson"}, age: 42, matiere: "anglais", presenter:
function}
> mary.presenter();
[Log] Salut! Je suis Mary Wilson et j'ai 42 ans.
[Log] J'enseigne la matière anglais
< undefined
```

Les objets de portée globale

Window

```
> var foo = "foobar"; foo === window.foo;
< true
> function greeting() {console.log("Hi!");}
< undefined
> window.greeting();
[Log] Hi!
< undefined
> greeting();
[Log] Hi!
< undefined
```

Java ou Javascript ?

- Interpréteur de bytecode.
- Compilation explicite en bytecode.
- Nombreux types de base.
- Statique.
- Orienté objet.
- Rapidité d'exécution des programmes.

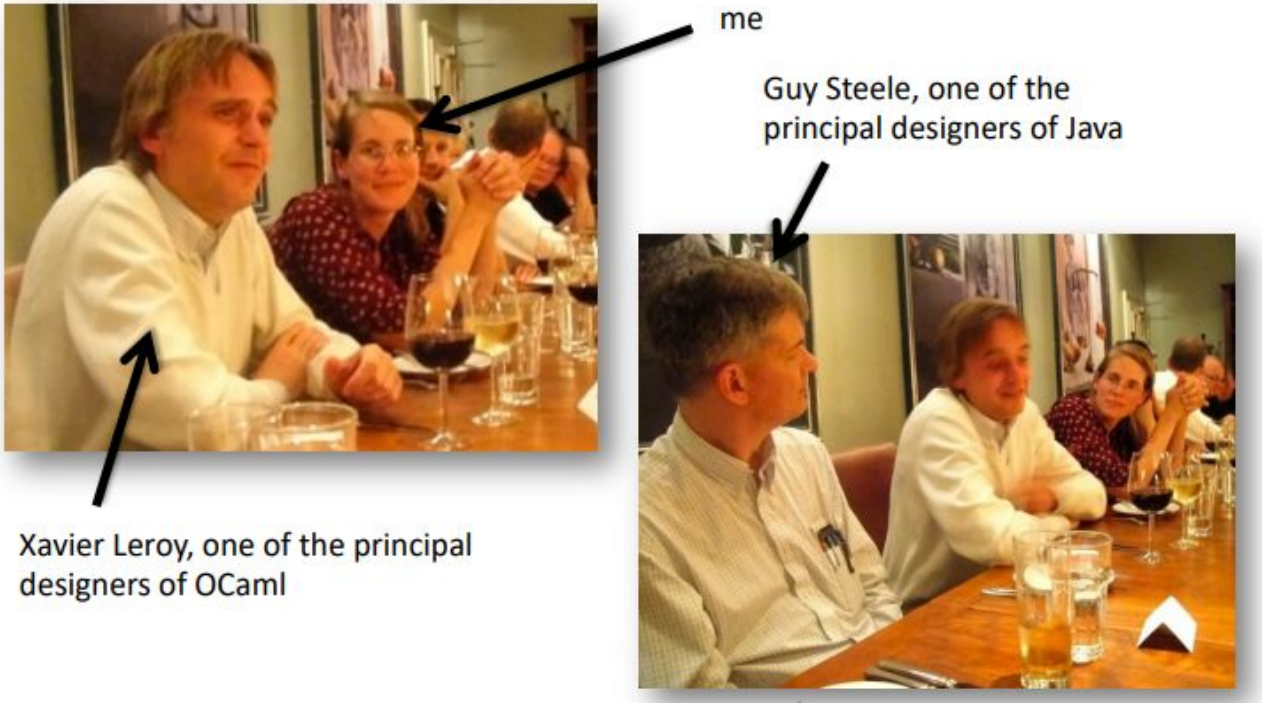
- Exécuté par un navigateur web.
- Pas d'étape de compilation.
- Peu de types de base.
- Dynamique.
- Multiple.
- Lents.

Ocaml ou Java ?

- Créer explicitement des objets à l'aide d'un enregistrement de fonctions d'ordre supérieur et d'un état masqué.
- Flexibilité par la composition: les objets ne peuvent implémenter qu'une seule interface.

- Notion primitive de création d'objets (classes, avec champs, méthodes et constructeurs)
- Flexibilité grâce à l'extension: le sous-typage permet aux objets associés de partager une interface commune.

Ocaml ou Java ?



Récapitulatif

	Java	Ocaml	JavaScript
Objet et classe	Objet = Instance d'une classe. Classe d'objets	Séparés l'un de l'autre	La classe est un sucre syntaxique
Héritage	Une classe peut hériter d'une seule autre classe, mais de plusieurs interfaces.	Héritage multiple entre classes	Héritage simple entre objets. Chaîne de prototypes
this	Mot-clé. Instance courante.	Identificateur au choix. Doit être explicitement lié à l'objet.	Mot-clé. Dépend du contexte (global, fonction). Choix de lier this à un contexte.
Abstraction	Oui	Oui	Non
Privé/Public	Oui	Privé modifiable par sous-classe	Pas de privé.

Merci pour votre attention.
Avez-vous des questions ?

Références

1. Java
- https://www.tutorialspoint.com/java/java_object_classes.htm
- https://docs.oracle.com/javase/tutorial/java/concepts/object.html
2. Ocaml
- https://caml.inria.fr/pub/docs/manual-ocaml/objectexamples.html
- https://dev.realworldocaml.org/objects.html
- https://dev.realworldocaml.org/classes.html
3. JavaScript
- https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference
- https://developer.mozilla.org/en-US/docs/Glossary/Global_object
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects
- https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes
- https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/Heritage