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

Dorothée Huynh

Haifa Ben Hmida El Abri

Programmation Comparée

05 Février 2021

I. Introduction

Les langages Java, Javascript et Ocaml ont été créés dans le milieu des années 1990, Ocaml en 1996, Java et JavaScript en 1995. Malgré cette proximité temporelle, nous verrons leurs similitudes et différences sur des notions communes sur les objets et les classes. Chacun de ces langages ont été définis dans différents systèmes de typage : le typage structurel, le typage nominal et les prototypes. Le typage structurel sert à deux objets d'avoir des types compatibles dans le cas où les types ont une structure identique. En effet, le typage structurel est plus flexible que le typage nominal car il permet la création de types et d'interfaces. Le typage nominal permet de vérifier la compatibilité de type des deux variables si et seulement si leurs déclarations nomment le même type. Les systèmes de types nominaux contrastent avec les systèmes structurels, où les comparaisons sont basées sur la structure des types en question et ne nécessitent pas de déclarations explicites alors que le typage nominal est utile pour empêcher l'équivalence de type accidentelle. La programmation orientée prototypes présente de nombreux avantages comme l'héritage dynamique, l'utilisation d'une mémoire différente de celle d'un objet à classe, et surtout une grande liberté dans la programmation qui permet au développeur de se focaliser d'abord sur l'exécution de quelques objets avant de classifier ces objets plus classiquement.

II. Comparaison des langages sur des notions communes

1. Les objets

Contrairement à Java où les objets sont des instances de classe et où une classe est un ensemble d'attributs, aussi appelés variables, et d'actions, implémentées sous la forme de méthodes, les objets en Ocaml sont séparés du système de classe. Ils peuvent être définis tels quels. Le type d'un objet est le type de toutes les méthodes qu'il contient, sans prendre en compte ses variables.

Nous pouvons créer des objets mutables et non mutables.

En Ocaml, un objet mutable a un état interne modifiable car au moins une de ses variables est mutable. On définit une variable mutable avec le mot-clé mutable. Il est nécessaire d'expliciter la mutabilité d'une variable en Ocaml car la caractéristique par défaut d'une variable est d'être immutable. Exemple d'un objet mutable dans le code source 15 : objet représentant un point en une dimension (abscisse x), doté d'un accesseur de x et d'une méthode pour augmenter x.

Quand on utilise un objet non mutable, il n'y a pas de modification de l'objet mais création d'un nouvel objet dont l'état interne équivaut à l'interne de p modifié. Exemple d'une pile immutable dans le code source 16. Les éléments de la pile sont stockés dans une liste (c'est une constrainte liée au pattern matching présent dans les méthodes pop et push) nommée v. L'objet possède un accesseur de v, une méthode pop qui renvoie la tête de liste et un objet pile immutable contenant la queue, une méthode push qui renvoie un nouvel objet pile immutable contenant la liste v concaténée avec le paramètre hd. À noter la syntaxe spéciale alternative pour créer un objet inline : l'objet est délimité par des accolades et les valeurs entre chevrons.

En Java, il est également possible de créer des classes à état interne non mutable ou mutable. Cependant, pour qu'une variable soit immutable il faut le spécifier explicitement à l'aide du mot-clé final. C'est l'inverse d'Ocaml où c'est la mutabilité qu'il faut spécifier explicitement.

Les langages orientés objet basés sur des classes, comme Java, se fondent sur deux entités principales distinctes qui sont les classes et les instances. Alors qu'un langage basé sur des prototypes, comme JavaScript, n'utilise pas cette distinction puisqu'Il ne possède que des objets. On peut avoir des objets prototypiques qui sont des objets agissant comme un modèle sur lequel on pourrait obtenir des propriétés initiales pour un nouvel objet. Tout objet peut définir ses propres propriétés, que ce soit à l'écriture ou pendant l'exécution. De plus, chaque objet peut être associé comme le prototype d'un autre objet, auquel cas le second objet partage les propriétés du premier. Le prototype d'un objet est utilisé pour fournir de façon dynamique des propriétés aux objets qui héritent du prototype.

2. Les classes

Les classes Java permettent de décrire une catégorie d'objets qui diffèrent par la valeur de leurs attributs. Les objets sont les instances de ces classes. On prend l'exemple d'une classe personne. Une personne a les attributs **age** et **nom** et les méthodes **getNom()** et **setNom(String nom).** Une implémentation en java : (code source 1)

Dans le langage Java, il existe une classe principale nommée Object. Toutes les classes qu'elles soient, prédéfinies ou créées par l'utilisateur du langage, héritent tous de la classe Object. Quand on déclare une classe sans la faire hériter explicitement de Object, le compilateur se charge de la rajouter. Donc en général une classe hérite toujours implicitement de Object. (code source 4) Cette déclaration est équivalente à la déclaration précédente : (code source 5).

Dans le langage Ocaml, la définition d'une classe se fait via un objet et on préfixe la définition de l'objet par le mot-clé class suivi du nom de la classe. Le type de la classe est le nom et type de chaque attribut et le nom et type de chaque méthode. À la différence de l'objet, le nom et le type des attributs apparaît dans le type de la classe. Une classe est vue comme un alias pour le type de l'objet avec lequel elle est définie. On appelle les méthodes d'une classe de la même manière que celles de l'objet qui sert à sa définition. Voir l'exemple d'une classe en Ocaml dans le code source 17. Tout comme en Java, on instancie une classe avec le mot-clé new suivi du nom de la classe.

Les classes en JavaScript sont du sucre syntaxique par-dessus les prototypes.

3. La construction d'une instance de classe

En Java, les constructeurs sont des méthodes qui ont la particularité de posséder le même nom que la classe à laquelle ils appartiennent. On reprend notre exemple ici : nous avons construit notre constructeur de la classe Personne. (code source 2)

En Ocaml, l'initialisation des attributs d'une classe ou d'un objet se fait lors de la déclaration de l'attribut. On peut aussi mettre des paramètres à la classe pour initialiser les attributs avec ces paramètres. Il n'y a pas de méthode spécifiquement dédiée à l'initialisation de la classe, comme c'est le cas avec les constructeurs en Java. Il y a en Ocaml la notion d'initialisateur. Un initialisateur est une

méthode cachée et anonyme qui est appelée immédiatement après la création de l'objet. Elle peut accéder au self et aux variables d'instance. On définit un initialisateur grâce au mot-clé initializer. Voir l'exemple de code source 19 qui est une classe de point affichable contenant un initialisateur qui permet d'afficher une string "new point at" et qui appelle la méthode print de la classe.

En JavaScript, les mot-clés constructeur et class sont du sucre syntaxique. Dans l'exemple suivant on a créé une classe personne qui possède un constructeur (codes source 10), ce qui est équivalent à la fonction suivante. (code source 11)

4. L'instance courante de la classe

Dans le langage Java, le mot clé **This** permet de désigner l'objet dans lequel on se trouve, c'est-à-dire que lorsque l'on désire faire référence dans une fonction membre à l'objet dans lequel elle se trouve. Lorsque l'on désire accéder à une donnée membre d'un objet à partir d'une fonction membre du même objet, il suffit de faire précéder le nom de la donnée membre par **This.** Dans le cas aussi où l'on désire une fonction membre qui retourne un pointeur vers l'objet dans lequel elle se trouve, la variable This est indispensable comme dans l'exemple de code source 6.

Dans le langage Ocaml, on peut faire une liaison à l'objet courant mais il faut le faire de manière explicite. Contrairement à Java et JavaScript qui ont le mot-clé this pour représenter l'objet courant, Ocaml n'en possède pas. On peut donner n'importe quel identificateur valide pour le nom désignant l'objet courant, mais self est le plus souvent utilisé. Voir l'exemple de code source 18 où la déclaration du self à côté du mot-clé object suffit pour pouvoir l'utiliser dans les méthodes de la classe printable_point.

Dans le langage Javascript il faut se rappeler qu'il n'y a que du code et des objets, tous les autres concepts de la programmation orientée objets ne sont que du sucre syntaxique. Quand on essaie de comprendre du code qui utilise le mot clé **this**, le bon réflexe est donc de regarder le contexte où le code qui utilise **this** a été appelé. On utilise les fonctions call et apply pour définir directement la valeur de **this** à l'exécution, comme l'exemple de code source 12 où il affiche l'objet passé en paramètre. this peut aussi être utilisé dans du code qui n'est pas dans une fonction : dans l'exemple de code source 13, this vaut undefined tandis que dans le code source 14, this n'est pas l'objet hello camille mais global, window ou undefined selon le mode d'exécution de ce code.

5. L'héritage

On procède au concept d'héritage qui est parmi les concepts les plus importants de la programmation orienté objet. On peut définir l'héritage comme un mécanisme permettant de créer une nouvelle classe à partir d'une classe existante en lui définissant ses propriétés et ses méthodes. Voyons dans cet exemple en langage Java que la classe femme herite de la classe personne, et lorsqu'on instancie la classe femme on pourra utiliser les methodes de la classe personne comme changer le nom à camille. (code source 7)

Dans le langage Java, une classe ne peut hériter que d'une seule autre classe et une classe ou une interface peut hériter de plusieurs autres interfaces. Pour spécifier l'héritage, on utilise le mot-clé extends pour qu'une classe hérite d'une autre classe ou qu'une interface hérite d'une autre interface. Dans le langage Ocaml, une classe peut hériter de plusieurs autres classes. Pour qu'une classe hérite

d'une autre classe, il faut utiliser le mot-clé inherit comme dans le code source 22. Afin d'hériter d'autres classes, il suffit d'écrire une ligne de inherit pour chaque classe mère.

L'objet de la classe mère est désigné par le mot-clé super en Java alors qu'en Ocaml c'est un identificateur dont le nom est laissé au libre choix du développeur. En Ocaml, il faut utiliser le mot-clé as pour créer un alias désignant l'objet de la classe mère (exemple dans le code source 22). Attention, cet alias n'est pas un vrai objet. Comme le mot-clé super en Java, il peut être seulement utilisé pour appeler des méthodes de la classe mère.

Dans les langages Java et Ocaml, il n'est pas nécessaire de redéfinir dans la classe fille une méthode de la classe mère. Les méthodes non privées sont héritées depuis la classe mère et sont donc accessibles dans la classe fille. Dans le langage Ocaml, l'ordre de définition de l'héritage par rapport aux méthodes de la classe fille est important. En effet, si on redéfinit une méthode (même nom qu'une méthode de la classe mère) dans la classe fille puis qu'on définit l'héritage avec le mot-clé inherit, c'est la méthode de la classe mère qui redéfinit celle de la classe fille alors que c'est l'inverse si on spécifie d'abord l'héritage avec inherit puis qu'on redéfinit la méthode dans la classe fille. Si une méthode est définie comme étant privée, on peut la rendre publique dans une classe fille en la redéfinissant sans le mot-clé private.

Dans le langage JavaScript, les mots-clés class, extends, super et constructor sont du sucre syntaxique. On peut voir JavaScript comme étant un langage basé sur les prototypes, chaque objet pouvant avoir un prototype objet d'où il hérite des méthodes et des attributs. Un prototype peut lui aussi avoir son prototype objet duquel il héritera des méthodes et des attributs et ainsi de suite. Cela constitue ce qu'on appelle une chaîne de prototypes. Cela permet d'expliquer pourquoi différents objets possèdent des attributs et des méthodes définis à partir d'autres objets. En réalité, les méthodes et attributs sont définis dans l'attribut prototype (la fonction constructrice de l'objet et non pas dans les instances des objets elles-mêmes). Voir l'exemple de code source 25 où il y a une classe mère Personne et une classe fille Professeur écrites sans le sucre syntaxique de la programmation objet orientée classe. Pour initialiser les attributs de la classe mère on appelle la fonction call, avec this et les paramètres du constructeur de la classe mère. Pour hériter des méthodes de la classe mère, on définit le prototype de la classe fille comme ayant la valeur d'un nouvel objet (appel à la méthode create() de la classe Object) qui a le prototype de la classe mère. De cette manière, le constructeur de la classe mère est également hérité et il est nécessaire de redéfinir le constructeur de la classe fille avec le nom de la classe fille. Pour appeler une méthode de la classe mère dans une méthode de la classe fille, on applique la fonction call à la méthode de la classe mère.

6. L'abstraction

L'abstraction sert à gérer la complexité en masquant les détails inutiles à l'utilisateur. Cela permet à l'utilisateur d'implémenter une logique plus complexe sans comprendre ni même penser à toute la complexité cachée.

Dans le langage Java, le mot clé **abstract** est utilisé avec les classes ou les méthodes. Dans cet exemple on a créé une classe Personne abstraite avec une méthode abstraite aussi chanter et on a voulu l'instancier mais ça produit une erreur car cette classe ne peut pas être utilisée pour créer des objets (code source 8). Les interfaces sont vues comme des classes totalement abstraites et elles ne

contiennent aucune méthode concrète comme le cas des méthodes void chanter ou parler dans le code source 9. Une interface définit un contrat (ensemble de méthodes) qui est matérialisé lorsqu'une classe implémente cette interface.

Dans le langage Ocaml, le mot-clé virtual est utilisé pour déclarer une méthode sans l'implémenter. L'implémentation est faite dans les sous classes. Une classe contenant au moins une méthode virtuelle doit être virtuelle elle aussi et donc être définie avec le mot-clé virtual entre le mot-clé class et le nom de la classe. Une classe virtuelle ne peut pas être instanciée, c'est l'équivalent de la classe abstraite en Java. Voir le code source 20 sur une classe abstraite abstract_point qui a comme sous-classe point.

Dans le langage JavaScript, il n'y a pas cette notion d'abstraction.

7. La visibilité

Dans le langage Java, il y a plusieurs niveaux de visibilité des variables, méthodes et classes : privé (accès seulement dans la classe courante), protégé (accès dans la classe courante et les classes filles), package (accès dans le même package), public (accès dans toutes les classes).

Dans le langage Ocaml, c'est soit privé soit public (visibilité publique implicite). Une méthode privée est déclarée avec le mot-clé private. Elle n'est pas visible dans l'interface d'une classe (c'est-à-dire non accessible depuis l'extérieur) et peut être invoquée seulement par les autres méthodes du même objet. Cette visibilité peut être modifiée par redéfinition de la méthode dans une classe fille.

En JavaScript, tout a une visibilité publique. Il y a par ailleurs les objets Window (cas général) et WorkerGlobalScope (cas des workers, c'est-à-dire des threads en arrière-plan) qui ont une portée globale qui englobe les objets dans le cas de Window ou les workers dans le cas de WorkerGlobalScope.

8. Comparatif des langages Java, Ocaml, JavaScript

	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
Instance courante	Mot-clé this.	Pas de mot-clé. Identificateur au choix. Doit être explicitement lié à l'objet.	Mot-clé this. Valeur qui 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.

III. Spécificités liées au langage

1. Static en Java

Le mot-clé static sur un attribut d'une classe indique que cet attribut est une variable de classe et non une variable d'instance. Cela signifie que sa valeur est accessible entre toutes les instances de la clases. (code source 3)

2. Les méthodes binaires en Ocaml

Une méthode binaire est une méthode qui prend en argument un objet de type self. Un exemple courant est la méthode de comparaison (égalité, infériorité, supériorité…). Voir l'exemple de code source 24 de la classe Square qui représente un carré. Cette classe possède une méthode equals qui prend en param un autre carré et qui renvoie true si cet autre carré a la même largeur que le carré courant, self.

Il est nécessaire d'avoir un objet du même type. Ici, dans le cas de la classe Square, on ne peut écrire des méthodes binaires seulement sur des objets de type Square. Dans d'autres langages, des mécanismes de vérification dynamique de type ou surcharge de méthode permettent de s'affranchir de cette contrainte et d'avoir du polymorphisme, mais Ocaml n'a rien de tout cela. Une bonne solution est de définir un type somme qui contient tous les sous-types. Par exemple shape_repr : Square of int | Circle of int. Il faut écrire explicitement tous les sous-types. Puis on définit la méthode binaire dans le super type et les sous-types doivent implémenter cette méthode binaire et une autre méthode repr (représentation interne des objets) qui indique quel self est utilisé. Pour cacher la fonction repr, on utilise le système de modules : on fait un module Shape, contenant Square et le type share_repr.

3. Le hoisting en JavaScript

Le Hoisting (ou remontée des déclarations en français) rend les variables et les fonctions disponibles pour une utilisation avant que la variable ne reçoive une valeur ou que la fonction ne soit définie. En effet, il place les déclarations de variables, de fonctions et de classes en haut de leur portée avant l'exécution.

En réalité, JavaScript ne déplace ni n'ajoute de code aux déclarations de hoisting. Ces déclarations sont mises en mémoire pendant la phase de compilation de l'interpréteur, les rendant disponibles avant l'exécution du code.

IV. Conclusion

Java est un langage orienté objet alors que javascript est considéré comme orienté objet et procédural. Le code d'une application en Java est interprété initialement, mais la JVM surveille les séquences de bytecode fréquemment exécutées et les traduit en code machine pour une exécution directe sur le matériel, alors qu'en javascript le code est exécuté par un navigateur web. Par rapport au processus de compilation, le code java est compilé explicitement en bytecode alors qu'en Javascript pas de compilation. En Java il y a de nombreux types de base et c'est le contraire en Javascript peu de types de base. Le typage du langage en Java est fort et statique alors qu'en Javascript il est faible et dynamique.

En Ocaml on crée explicitement les objets à l'aide d'un enregistrement de fonctions d'ordre supérieur et d'un état masqué qui est un état des variables d'instance. On a aussi une certaine flexibilité par la composition où les objets ne peuvent implémenter qu'une seule interface mais qui peut être composée des interfaces des classes mères. En java, on crée des objets avec les classes , les méthodes et les constructeurs. Ce langage est flexible grâce à l'extension où le sous-typage permet aux objets associés de partager une interface commune.

VII. Références

[1] https://www.tutorialspoint.com/java/java_object_classes.htm

[2] https://docs.oracle.com/javase/tutorial/java/concepts/object.html

[3] https://caml.inria.fr/pub/docs/manual-ocaml/objectexamples.html

[4] https://dev.realworldocaml.org/objects.html

[5] https://dev.realworldocaml.org/classes.html

[6] https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference

[7] https://developer.mozilla.org/en-US/docs/Glossary/Global_object

[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

[9] https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes

[10] https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/Heritage

Annexes

Code source 1

```
1  public class Personne {
2      int age;
3      String nom;
4      public getNom(){
5          return nom;
6      }
7      public setNom(String nom){
8          this.nom = nom;
9      }
10 }
11 public class CreatePersonne {
12     public static void main (String[] args){
13     Personne p  = new Personne();
14     p.setNom("Martin");
15     }
16 }
17
```

Code source 2

```
1 public class Personne {
2     int age;
3     String nom;
4     public Personne(int age, String nom){
5         this.age =age;
6         this.nom =nom;
7     }
8 }
9
```

Code source 3

```
1 public class Femme extends Personne {
2     static float taille = 70.9;
3     public Femme(){}
4 }
5
```

Code source 4

```
1  public class Personne extends Object {
2      int age;
3      String nom;
4      public getNom(){
5      return nom;
6      }
7      public setNom(String nom){
8      this.nom = nom;
9      }
10 }
11
```

Code source 5

```
1  public class Personne {
2      int age;
3      String nom;
4      public getNom(){
5      return nom;
6      }
7      public setNom(String nom){
8      this.nom = nom;
9      }
10 }
11
```

Code source 6

```
1  public class Personne {
2      int age;
3      public AgePersonne(int age){
4          this.age=age;
5      }
6      Personne * RetournePersonne(){
7          return this;
8      }
9  }
10
```

Code source 7

```
1  public class Femme extends Personne {
2      public String chanter(){
3          return "la la la….";
4      }
5  }
6
7  public class CreatePersonne {
8      public static void main (String[] args){
9          Femme f = new Femme();
10         f.setNom("Camille");
11         System.out.println(f.chanter());
12     }
13 }
14
```

Code source 8

```
1  abstract class Personne{
2      public abstract void chanter();
3  }
4  Personne p = new Personne();    // Erreur
5
```

Code source 9

```
1  interface Personne_Interf {
2      void chanter();
3  }
4
5  interface Personne_Interf extends another_interf{
6      void parler();
7  }
8
9  class Personne implements another_interf{
10     void chanter(){
11         System.out.println("la la la ..");
12     }
13 }
```

Code source 10

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

Code source 11

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

Code source 12

```
function personne ()
    {   console.log(this) ;
    }
personne.call({hello: 'camille'});
```

Code source 13

```
function personne ()
    {   console.log(this) ;
    }
personne();
```

Code source 14

```
function personne ()
{   function individu ()
    {
        console.log(this) ;
    }
    personne();
}
personne.call({hello: 'camille'});
```

Code source 15

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

Code source 16

```
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;;
```

Code source 17

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

Code source 18

```
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;;
```

Code source 19

```
 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;;
```

Code source 20

```
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 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;;
```

Code source 21

```
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;;
```

Code source 22

```
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;;
```

Code source 23

```
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
```

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

end;;	

Code source 24

```
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;;
```

Code source 25

```
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);
}
```