Objets Structurels (OCaml) vs. Nominaux (Java) vs. Prototypes (JS)

LENOIR Quentin - HADJAB Lynda

Année Universitaire 2021-2022

Introduction

Il existe en programmation orientée objets trois grands courants pour définir des objets : le typage structurel, nominal et prototypal. Les langages sont souvent basés sur un seul d'entre eux, même si certains langages essayent d'en utiliser plusieurs en même temps (donner exemple). Nous allons présenter les caractéristiques de ces trois typages, et comparer leurs caractéristiques.

Exposé

I - Typage Structurel

Le typage structurel est l'un des systèmes de types existant en programmation informatique, qui est basé sur la propriété, tel que deux types d'objets sont dits équivalents si et seulement s'ils ont la même structure. En d'autres termes, cette vérification de l'équivalence ne se vérifie pas par rapport à la déclaration explicite des types mais par rapport à la structure de chaque type c'est-à-dire s'ils ont les mêmes champs publics et des méthodes de type et signature compatibles. De même, un type T1 est considéré comme un sous-type d'un type T2 si et seulement si T1 a tous les membres publics (de type et signature compatible) que T2, mais T1 peut en avoir plus c'est-à-dire que T1 peut avoir plus de méthodes que T2 contrairement à T2 pour pouvoir dire que T1 est un sous type de T2 .

L'un des langages de programmations majeur qui utilise ce système de types est le langage Ocaml, car lorsque le typeur d'Ocaml compare deux types d'objets, il compare leurs structures et pas leurs noms. Par exemple si on considère les deux classes suivantes en Ocaml :

```
class person _name () =
object
        val mutable name = _name
        method getname = name
        method setname _name = name <- _name
end;;

class product _name _price =
object
        val mutable name = _name
        val mutable price = _price
        method getname = name
        method setname _name = name <- _name
        method getprice = price
        method setprice _price = price <- price + _price
end;;
```

Les interfaces de Person et Product ont la même structure, ils ont les mêmes noms et types de méthodes. En Ocaml les deux types (Person et Product) sont considérés équivalents, compatibles et égaux.

De même le type ou la classe Product est considéré comme un sous type de Person parce que tous les membres publics de Person sont de types compatibles, ce qui veut dire la classe Product contient tous les membres public de Person mais l'inverse n'est pas vrai, Person n'est pas un sous type de Product car Person ne contient pas tous les membres public de Product. Par exemple, on ne trouve pas la fonction getprice dans l'objet Person.

II - Typage Nominal

Le typage nominal est l'un des systèmes de types les plus utilisés en langages de programmation, et est donc employé dans des langages comme Java, TypeScript & Scala. Avec ce système de typage, l'équivalence et la compatibilité des types est vérifiée par rapport au nom des types, c'est-à-dire s'ils n'ont pas le même nom de type alors ils seront considérés comme non compatibles ou bien si ce n'est pas déclaré explicitement que les deux objets ont le même type comme via une relation d'héritage alors les types ne seront pas considérés égaux et donc seront incompatibles.

Le typage nominal est un sous-ensemble du typage structurel étendu avec une relation supplémentaire déclarant explicitement la relation de sous-type (comme la clause extends). De même, le typage nominal ne permet pas de créer de nouveaux super-types sans modification des sous-types existants, ce qui réduit la flexibilité.

On considère les deux classes suivantes en Java :

```
class Person {
  public String name;
  public String getName() { return this.name;}
  public void setName(String _name) { this.name = _name;}
 }

class Product {
  public String name;
  public String getName() { return this.name;}
  public void setName(String _name) { this.name = _name;}
 }

Product p = new Person(); //Erreur Product n'est pas de type ou sous-type Person
```

En essayant de déclarer un objet de type Product et l'initialiser avec le type Person on va avoir une erreur de compilation en Java, contrairement au langage Ocaml car avec Java l'équivalence de type est déterminée par le nom explicite de type et non par la structure de type comme en Ocaml. On peut remarquer que les deux classes ont la même structure ce qui fait qu'en avec Ocaml il n'y aurait pas eu d'erreurs de compilation et les types auraient été compatibles. Le typage nominal ne permet également pas de réaliser de l'héritage multiple, ce que le typage structurel permet, ainsi le typage nominal offre une meilleure sécurité des types, au coût d'une flexibilité réduite.

Le typage nominal permet donc une meilleure sécurité de type et ainsi de prévenir des équivalences de types accidentelles contrairement au typage structurel dans laquelle le nom des types n'est pas pris en considération lors de la vérification de l'égalité des types. Cependant, avec le typage nominal, il est impossible de créer des types et des interfaces ad-hoc , dans lesquelles une valeur (ou une interface) peut avoir plus d'un type, contrairement au sous-typage structurel. De ce fait, on peut dire que le sous-typage structurel est plus flexible que le sous-typage nominal. De même, on peut créer un nouveau type à partir d'un sous type déjà existant Y sans modifier le sous type Y pour l'adapter à ce nouveau type, c'est tout le contraire de sous typage nominal qui fait l'inverse.

III - Typage Prototypal

Le typage prototypal est le plus permissif des trois typages, car il permet une grande modularité et flexibilité sur les objets (prototypes) que l'on manipule. Ainsi un prototype est un objet, au même titre que les autres, il a donc une existence et une présence physique en mémoire, contrairement aux objets à classes où les classes sont des abstractions et les objets des instances des classes.

Le code d'une classe à objet est statique, il ne peut être modifié après déclaration, or, avec les prototypes, le code est dynamique, il peut être appelé, modifié, il doit être nommé cependant. Ainsi, toute modification est possible à tout moment de l'exécution. Ceci permet donc de toucher sur du code qui est déjà en production par exemple, si une modification doit être apportée à tous les objets d'un même type, en modifiant le prototype parent, ou commun de ce type, tous les enfants seront mis à jour dynamiquement. Ceci est à double tranchant : on peut propager du bon code rapidement et efficacement, en revanche l'inverse est tout aussi vrai, il est possible d'introduire des bugs majeurs très rapidement. Considérons le code JavaScript suivant, qui utilise un typage prototypal :

```
let Personne = function(nom,prenom,age){ //Constructeur de notre objet
   this.nom = nom;
   this.prenom = prenom;
   this.age = age;
}

let Quentin = new Personne("Lenoir","Quentin",”22”);
// état de l'objet Quentin : Personne {nom: 'Lenoir', prenom: 'Quentin', age: 22}

Personne.prototype.nomComplet = function(){
   alert(this.nom + " " + this.prenom) //this fonctionne car nous sommes dans le prototype de Personne
}

Quentin.nomComplet() //fonctionne et affiche bien “Lenoir Quentin” dans une popup, malgré
l'instanciation avant la déclaration

let Eleve = function(nom,prenom,age,moyenne){
    Personne.call(this,nom,prenom,age);
    this.moyenne = moyenne;
}

Eleve.prototype = Object.create(Personne.prototype); //Héritage de l'objet Personne
Eleve.prototype.constructor = Eleve; //Sinon on perd le constructeur de Eleve

let Lynda = new Eleve(“Hadjab”,”Lynda”,”22”,15);

Lynda.nomComplet() //fonctionne car Eleve hérite des méthodes de Personne grâce au prototypage !

Personne.prototype.biographie = function(){alert('Je m'appelle '+this.prenom+' '+this.nom+' et j'ai
'+this.age+' !')};
```

```
Lynda.biographie(); //fonctionne car la chaîne de prototypage est mise à jour dynamiquement !
```

Cet exemple montre donc la flexibilité qu'un typage par prototype offre, et la facilité avec laquelle il est mis en place, mais cependant il montre aussi que c'est un outil qu'il faut maîtriser, et dont il faut s'assurer la rigueur de comportement. Le typage prototypal permet aussi de redéfinir les prototypes fournis par défaut par Javascript, ce qui est déconseillé, pour plusieurs raisons. L'une d'entre elles étant que certaines librairies qu'on pourrait vouloir utiliser en Javascript pourrait elle-même déjà les redéfinir, et ainsi causer des conflits dans l'exécution. C'est aussi possible que la mise à jour du prototype ne soit pas effective sur l'entièreté d'un projet, et que si deux développeurs implémentent différemment un même prototype et que l'un d'eux a un souci, il pourrait être difficile de comprendre d'où vient le problème.

L'héritage par prototype ne permet pas d'héritage multiples, car il fonctionne par chaîne et on ne peut pas appartenir à deux chaînes en même temps. De même, la notion d'interface n'existe pas, car une méthode peut être implémentée à tout moment à n'importe quel prototype. On peut donc reproduire le comportement d'interfaces via héritage.

Pour résumer, la programmation par prototype offre une grande flexibilité, une programmation dynamique et simplifiée, qui implique aussi de savoir ce que l'on fait avec.

Conclusion

Pour conclure, ces trois grands typages ont chacun leurs caractéristiques, leurs avantages et leurs inconvénients, et ainsi, leur choix peut dépendre de la modularité requise pour le programmeur, ou bien de la facilité syntaxique. En effet, pour une flexibilité et une facilité plus grande, le typage prototypal sera plus propice, mais en revanche pour un code moins flexible mais qui offre une meilleure sécurité des types qu'un typage structurel, le typage nominal sera plus avantageux.

Bibliographie

https://fr.wikipedia.org/wiki/Syst%C3%A8me_structurel_de_types

https://wiki.haskell.org/Polymorphism

https://fr.wikipedia.org/wiki/Syst%C3%A8me_nominatif_de_types

https://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_prototype

https://france2.wiki/wiki/Prototype-based_programming

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/Object_prototypes

https://developer.mozilla.org/fr/docs/Web/JavaScript/Inheritance_and_the_prototype_chain