

Programmation Objet : Concepts Avancés

Cours 6 : Étude comparée de C#, Python, Scala et O'Caml

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

7 novembre 2008

Comment choisit-on un langage de programmation ?

- ▶ Il faut se donner des critères de comparaison :
 - ▶ l'expressivité ;
 - ▶ les garanties apportées par le système de type ;
 - ▶ le ou les paradigmes de programmation qu'ils favorisent ;
 - ▶ degré de formalisation, de standardisation ;
 - ▶ la pérennité ;
 - ▶ l'existence de compilateurs efficaces ;
 - ▶ l'existence de bibliothèques ;
 - ▶ ...
- ▶ Savoir répondre à ces questions permet de déterminer le langage le plus pertinent vis-à-vis de vos besoins.



Objectif de ce cours

- ▶ Dans ce cours, nous allons présenter les **spécificités** de 4 langages à objets.
-
- ▶ C# (3.0), un langage :
 - ▶ impératif et fonctionnel,
 - ▶ statiquement typé,
 - ▶ à base de classes,
 - ▶ à sous-typage nominal,
 - ▶ avec inférence des types (faible).
 - ▶ Python, un langage :
 - ▶ impératif et fonctionnel,
 - ▶ dynamiquement typé,
 - ▶ à base de classes.
 - ▶ Scala, un langage :
 - ▶ impératif et fonctionnel,
 - ▶ statiquement typé,
 - ▶ à base de classes,
 - ▶ à sous-typage nominal,
 - ▶ avec inférence des types.
 - ▶ OCaml, un langage :
 - ▶ impératif et fonctionnel,
 - ▶ statiquement typé,
 - ▶ à base de classes et de modules,
 - ▶ à sous-typage structurel,
 - ▶ avec inférence des types.
-

Pour plus tard ...



- ▶ Nous mettons de côté, pour le moment :
 - ▶ la **reflexivité** proposée par C#, Python et Scala
 - ⇒ Ce sera l'objet d'un prochain cours.
 - ▶ le **polymorphisme paramétrique** proposée par C#, O'Caml et Scala
 - ⇒ Ce sera l'objet d'un prochain cours.

C#

En deux mots ...

- ▶ C# est un langage de programmation objet développé par Microsoft.
- ▶ Il est compilé vers le *Common Language Runtime* (CLR), la plateforme d'exécution de l'environnement .Net. (nous le reverrons lors du cours sur l'interopérabilité).
- ▶ Il existe une implémentation libre de C#, appelée Mono.
- ▶ C# existe en 3 versions :
 - ▶ 1.0 – normalisé par l'ECMA (ECMA-334) en décembre 2001 et par l'ISO/CEI (ISO/CEI 23270) en 2003.
 - ▶ 2.0 – normalisé par l'ECMA (ECMA-334) en juin 2006 et par l'ISO/CEI (ISO/IEC 23270 :2006) en septembre 2006.
 - ▶ 3.0 – sortie en 2007.
- ▶ Si la syntaxe de C# est proche du C++, les caractéristiques du langage le rapproche plus de Java ou Pascal Objet.

Les types et les valeurs de C#

- ▶ C# propose un système de type unifié : comme en Java, toutes les valeurs sont des objets (des sous-classes d'une class Object).
- ▶ Il y a deux grandes catégories de valeurs :
 - ▶ les valeurs de base de type `int`, `bool`, ...
 - ▶ les valeurs références (stockent des références aux données).
- ▶ Ces valeurs sont allouées sur la pile.
- ▶ Une instance d'objet est toujours allouée sur le tas.
- ▶ Un ramasse-miettes s'occupe de la gestion mémoire.
- ▶ Il existe un mode `unsafe` qui permet de manipuler les adresses des objets sous forme de pointeurs (réservé à une programmation de bas niveau).

Initialisation des variables

- ▶ La non initialisation des variables est une source de bugs inépuisable en programmation impérative.
- ▶ En C#, on ne peut pas utiliser une variable tant qu'elle n'est pas initialisé.

```
using System ;  
public class Foo {  
  
    public static void Main () {  
        int i ;  
        Console.WriteLine (i) ;  
    }  
}
```

```
/tmp % smcs notinit.cs  
notinit.cs(6,24) : error CS0165: Use of unassigned local variable 'i'  
Compilation failed: 1 error(s), 1 warnings
```

- ▶ Si on désire vraiment déclarer une variable non initialisée, il faut lui donner un type annulable (*nullable*). Pour cela, on utilise le type d'un point d'interrogation.
- ▶ Exemple : `int? i = null ;`

Comment partager un simple entier ?

- ▶ On utilise souvent des références sur des valeurs de base pour partager des résultats entre fonctions en travaillant en espace constant.
- ▶ C# réintroduit le mot-clé de Pascal **ref** qui permet de passer un argument (initialisé) par référence :

```
using System ;  
public class Foo {  
    public static void dincr (ref int i) { i += 2 ; }  
    public static void Main () {  
        int i = 0 ;  
        dincr (ref i) ; /* On doit préciser qu'on passe l'argument par référence. */  
        Console.WriteLine (i) ; /* i vaut 2 ici. */  
    }  
}
```

- ▶ Il introduit aussi un mot-clé **out** permettant de passer un argument non initialisé par référence.

Les structures, des enregistrements classiques

- ▶ Contrairement à C++, le mot-clé `struct` de C# n'est pas (quasi-)équivalent au mot-clé `class`.
- ▶ Les structures de C# sont traitées comme des valeurs non référencées. Elles sont allouées sur la pile et on ne peut pas définir de relation d'héritage entre elles.
- ▶ Ces valeurs sont des implémentations de n-uplets plus légères que les objets.

Les types anonymes, des enregistrements définis à la volée

- ▶ Prédéfinir les types de n-uplets qui vont être utilisés peut parfois être fastidieux (par exemple, dans une couche de liaison avec une base de données).
- ▶ Le compilateur C# peut automatiquement insérer ces définitions à partir des utilisations de n-uplets faites dans le programme.
- ▶ Il s'agit d'une forme (faible) d'**inférence des types**.
- ▶ Ainsi, il est possible d'écrire :

/ "var" signifie que le compilateur doit déterminer le type de la variable. */*

/ Ce type n'est utilisable que pour les variables locales. */*

```
var i1 = new { x = 0, y = "bar" }
```

```
var i2 = new { x = 42, y = "foo" }
```

sans même qu'une classe contenant un attribut 'x' et un attribut 'y' n'est été prédéfinie.

- ▶ Les variables 'i1' et 'i2' ont le même type.

Les fonctions sont de première classe en C#

- ▶ Comme nous l'avons vu dans le cours sur les *design patterns*, manipuler des fonctions comme des **objets de première classe** est essentiel pour programmer des comportements.
- ▶ On peut très bien utiliser des objets pour cela car ils encapsulent des méthodes. Néanmoins :
 - ▶ le processus est très lourd syntaxiquement car il faut déclarer une classe spécifique pour chaque fonction,
 - ▶ le mécanisme de **capture de l'environnement** doit être fait manuellement.

Comparez !

```
class fresh_names {
private: int _i;
public:
    class fresh_name_generator {
private: fresh_names& _f;
public:
    fresh_name_generator (fresh_names& f) :
        _f (f) {}
    std::string operator() () {
        std::stringstream s;
        _f._i++;
        s << "name_" << _f._i << std::ends;
        return s.str ();
    }
};

int main (int argc, char** argv) {
    fresh_names pool;
    fresh_names::fresh_name_generator
        generator (pool);
    std::cout << generator () << std::endl;
    std::cout << generator () << std::endl;
}
```

```
let fresh_name_generator_pool () =
    let x = ref 0 in
        fun () → incr x; "name_" ^ string_of_int !x

let fresh_name_generator =
    fresh_name_generator_pool ()

let _ =
    output_string (fresh_name_generator ()) ;
    output_string (fresh_name_generator ())
```

Les délégués, les fonctions de première classe de C#

```
class Program {  
    public delegate string Generator ();  
  
    public static Generator pool () {  
        int x = 0;  
        return () => { x++; return "name_" + x; };  
    }  
  
    static public void Main () {  
        Generator g = pool ();  
        System.Console.WriteLine (g ());  
        System.Console.WriteLine (g ());  
    }  
}
```

Des fonctions de première classe sans inférence des types ?

- ▶ Il n'est pas nécessaire de proposer un mécanisme d'inférence des types quand on fournit des fonctions de première classe . . . mais ça peut éviter au programmeur d'écrire de très gros types !

Les membres de classe en C#

- ▶ Comme en C++, les membres d'une classe peuvent être des attributs, des méthodes, des constantes, des types, des classes, des constructeurs et des destructeurs.
- ▶ On trouve en plus des **indiceurs**, des **événements** et des **propriétés**.
- ▶ Les modificateurs d'accès sont aussi importés de C++ : **private**, **public**, **protected**.
- ▶ Est rajouté le modificateur **internal** qui permet de ne donner accès à un composant seulement à l'intérieur de l'unité de compilation.

Les événements

- ▶ Les événements sont une implémentation du patron sujet-observateur directement dans le langage.
- ▶ On peut publier un événement dans une classe.
- ▶ Des instances peuvent ensuite s'inscrire pour être informées lorsque l'événement se concrétise.
- ▶ Nous reviendrons sur cette notion dans le chapitre sur l'interopérabilité.

Les propriétés

- ▶ Le mécanisme le plus simple pour assurer un premier niveau d'encapsulation consiste à ne jamais donner un accès direct aux attributs d'un objet.
- ▶ La pratique courante lorsqu'un attribut "logique" existe naturellement est de fournir des **accesseurs** en lecture et en écriture au client.
- ▶ Exemple : un vecteur est défini par une direction et une norme.
- ▶ Le code du client ressemble alors à ceci :

```
e.setNorm (f.getNorm () - g.getNorm ())
```

- ▶ Les propriétés permettent de définir des accesseurs qui s'utilisent syntaxiquement comme des attributs classiques :

```
e.Norm = f.Norm - g.Norm
```

ce qui allège la syntaxe.

Les propriétés : exemple

```
class Program {  
  
    class Vector {  
        private float norm ;  
        public float Norm {  
            get { return norm ; }  
            set { System.Diagnostics.Debug.Assert (value  $\geq$  0) ; norm = value ; }  
        }  
    }  
  
    public static void Main () {  
        Vector x = new Vector () ;  
        x.Norm = 42 ;  
        x.Norm = x.Norm + 1 ;  
    }  
  
}
```

Les propriétés en lecture seule

- ▶ Il suffit d'omettre l'accessor en écriture pour ne donner au client qu'une interface en lecture seule sur l'attribut.
- ▶ Il existe un modificateur `readonly` qui force l'immutabilité d'un attribut une fois qu'il a été initialisé (au moment de sa déclaration ou dans le constructeur de la classe).
- ▶ On trouve toujours le mot-clé `const` qui est similaire à `readonly` mais un peu plus restrictif : l'initialisation doit être faite nécessairement au sein de la déclaration.

Les indicateurs

```
class SampleIntCollection
{
    private int[] arr = new int[100];
    public int this[int i] {
        get { return arr[i]; } /* Is used to evaluate this[x]. */
        set { arr[i] = value; } /* Is used to evaluate this[x] = e; */
    }
}

class Program
{
    static void Main(string[] args)
    {
        SampleIntCollection intCollection = new SampleIntCollection();
        stringCollection[0] = 42;
        System.Console.WriteLine(intCollection[0]);
    }
}
```

(source : <http://msdn.microsoft.com/en-us/library/6x16t2tx.aspx>)

Déclaration de classes en C#

- Les déclarations de classe en C# se font ainsi :

```
public class (abstract|sealed) A : B, I1, ..., IN {  
    public A () {} /* Un constructeur par défaut. */  
    public A (T x) {} /* Un constructeur à un argument de type T. */  
}
```

- Le mot-clé `sealed` signifie qu'on ne peut pas hériter de la classe.
- L'héritage est `simple` (dans notre exemple, B est une classe).
- On peut implémenter de multiples interfaces (I1, ..., IN ici).

Les déclarations de méthodes en C#

- ▶ La syntaxe des déclarations de méthode a été conçu pour qu'un client puisse comprendre la résolution des méthodes déclarées dans une classe A en regardant seulement le code de la classe A (contrairement à C#).
- ▶ Voici cette syntaxe :

method_declaration = modifier T method_id (arguments) ;

modifier = new | public | protected | internal | private | static
| virtual | sealed | override | abstract | extern

- ▶ On retrouve beaucoup de mot-clé de C++ , ils ont ici un sens à peu près équivalent.
- ▶ Le mécanisme de résolution statique/dynamique a été importé avec une légère amélioration.

Liaison tardive en C#

```
class Program {  
    class A { public virtual void foo () { System.Console.WriteLine ("A.foo"); } }  
    class B : A { public virtual void foo () { System.Console.WriteLine ("B.foo"); } }  
    class C : B { public virtual void foo () { System.Console.WriteLine ("C.foo"); } }  
  
    public static void Main () {  
        C c = new C ();  
        A a = c;  
        a.foo ();  
    }  
}
```

- Ce programme ne compile pas en C# :

lateBinding.cs(8) warning CS0114 'Program.B.foo()' hides inherited member 'Program.A.foo()'. To make the current member **override** that implementation, add the **override** keyword. Otherwise add the **new** keyword

Liaison tardive en C#

```
class Program {  
    class A { public virtual void foo () { System.Console.WriteLine ("A.foo"); } }  
    class B : A { public override void foo () { System.Console.WriteLine ("B.foo"); } }  
    class C : B { public override void foo () { System.Console.WriteLine ("C.foo"); } }  
  
    public static void Main () {  
        C c = new C ();  
        A a = c;  
        a.foo ();  
    }  
}
```

- Le comportement est celui attendu : C.foo est appelée.

```
Program { A { public virtual void foo () { System.Console.WriteLine ("A.foo"); } } B public virtual new void foo ()
```

Liaison tardive restreinte en C#

```
class Program {  
    class A { public virtual void foo () { System.Console.WriteLine ("A.foo"); } }  
    class B : A { public override sealed void foo () { System.Console.WriteLine ("B.foo"); } }  
    class C : B { public override void foo () { System.Console.WriteLine ("C.foo"); } }  
  
    public static void Main () {  
        C c = new C ();  
        A a = c;  
        a.foo ();  
    }  
}
```

- Le mot-clé `sealed` interdit la redéfinition :

lateBinding.cs(12) error CS0239 'Program.C.foo()' cannot override inherited member 'Program.B.foo()' because it is sealed

Références

- ▶ <http://msdn.microsoft.com>
- ▶ <http://www.cs.vu.nl/~vadi m/grammars/CSharp/grammar.html>
- ▶ http://www.mono-project.com/CSharp_Compiler

Python

Présentation

- ▶ Python est langage de programmation orienté-objet, fonctionnel et impératif, développé par Guido van Rossum (Google). La première version de Python a été publiée en 1991.
- ▶ Contrairement aux langages vus précédemment, Python est un langage à **typage dynamique**, interprété dans un environnement d'exécution muni d'un ramasse-miettes.
- ▶ Il est utilisé par Google, la NASA, YouTube, ...

*Retour sur
le typage dynamique
et
le typage statique*

Une question qui revient sans cesse ...

Que fait ce programme ?

Une question qui revient sans cesse ...

Que fait ce programme ?

- ▶ Est-ce qu'il s'exécute sans erreur ?

Une question qui revient sans cesse ...

Que fait ce programme ?

- ▶ Est-ce qu'il s'exécute sans erreur ?
- ▶ Est-ce qu'il fait ce qu'il doit faire ?

Une question qui revient sans cesse ...

Que fait ce programme ?

- ▶ Est-ce qu'il s'exécute sans erreur ?
 - ▶ Est-ce qu'il fait ce qu'il doit faire ?
- ⇒ J'aimerais le vérifier automatiquement ...

Posons la question à Alan Turing ...



« Il est impossible d'écrire un programme qui décide si un programme arbitraire termine. »

Une limite infranchissable . . . mais que pouvons-nous faire ?

- ▶ On peut vérifier qu'une fonction qui attend deux arguments est toujours appelée avec deux arguments.
 - ▶ On peut vérifier qu'une fonction qui attend un entier est toujours appelée avec une valeur entière.
 - ▶ On peut vérifier que dans un contexte utilisant un objet de type A, un objet de type B sera compatible car B est un sous-type de A.
 - ▶ On peut vérifier qu'une preuve P, donnée par le programmeur, certifie qu'un programme trie correctement une liste (cf. le cours sur la certification) ou que ce programme termine toujours quelque soit la liste.
- ⇒ Ces programmes fournissent des **garanties** sur toutes les exécutions possibles du programme en observant seulement son code source.
- ⇒ Ce sont des **analyses statiques**.
- ⇒ Obtenir des analyses statiques toujours plus fines est un domaine de recherche très actif !

Comment faire face à l'inconnu ?

- ▶ Dans un langage comme Pascal , la vérification des types contraint le programmeur à connaître précisément tous les types des données que son programme va utiliser.
 - ▶ Dans les systèmes de type avec polymorphisme, on peut manipuler des données en ayant une connaissance partielle de leur type :
 - ▶ soit leur type est opaque (polymorphisme paramétrique, comme en O'Caml) ;
 - ▶ soit leur type est un sous-type d'une interface connue (sous-typage).
- ⇒ Malheureusement, cette connaissance partielle est parfois une approximation trop imprécise de l'inconnu ...

Concéder un test dynamique : l'exemple de instanceof

- ▶ Lorsque l'analyse statique n'est pas assez précise, on permet au programmeur d'effectuer un test à l'exécution permettant de recouvrer statiquement une information sur la nature des données :

```
public Shape is_shape (Object o) {  
    if (o instanceof Shape)  
        // Si le test a réussi, on a la garantie statique de le type de o est Shape.  
        return (Shape) o ;  
    else  
        // Erreur !  
}
```

- ▶ La garantie était :
Si le programme est bien typé, il s'exécute sans erreur.
- ▶ Elle est maintenant **plus faible** :
Si le programme est bien typé et que instanceof n'échoue jamais, il s'exécute sans erreur.
mais on a gagné en expressivité.

Pour aller encore plus loin . . .

- ▶ La recherche actuelle en typage statique tend à :
 - ▶ augmenter l'expressivité des types pour que l'analyse statique effectuée par le vérificateur de type soit plus précise ;
 - ▶ intégrer des tests dynamiques dont l'issue positive ou négative peut être traitée comme une nouvelle connaissance statique (typage hybride, types algébriques généralisés, typage multi-niveaux).

Mais en attendant ...

- ▶ Certaines fonctionnalités utiles ne seront pas intégrées dans les langages de l'industrie avant quelques dizaines d'années.
- ▶ En particulier, la **générativité**, c'est-à-dire la capacité à produire automatiquement du code, est nécessaire dès à présent pour :
 - ▶ Faire interagir des composants hétérogènes (cf. le cours sur l'interopérabilité) ;
 - ▶ Permettre la programmabilité des applications par l'utilisateur.
- ▶ Comment embarquer des objets répondant à des messages inconnus dans une application existante ?

Le choix de la vérification dynamique

- ▶ Python et Objective-C font le choix de ne pas vérifier les programmes avant leurs exécutions.
- ▶ Les valeurs du langage ainsi que la description de leurs types ont une existence durant l'évaluation du programme.
- ▶ Juste avant d'évaluer une expression, on vérifie que les valeurs ont la bonne structure. [DEMONSTRATION]
- ▶ Ce sont donc des langages **non sûrs**.
- ▶ En contrepartie, ils offrent plus de souplesses que les langages à typage statique.
- ▶ En particulier, ils sont parfaitement adaptés pour implémenter les fonctionnalités du transparent précédent.

Structure d'un programme en Python

- ▶ Python est un langage **interprété**.
- ▶ On peut l'utiliser de façon dans une boucle d'interaction ou par le biais de fichiers de **commandes** dont l'extension est .py (mode *batch*).

Commandes en Python

- Il existe plusieurs classes de commandes, en voici la grammaire BNF :

```
compound_stmt  ::=  if_stmt
                  |  while_stmt
                  |  for_stmt
                  |  try_stmt
                  |  with_stmt
                  |  funcdef
                  |  classdef
suite           ::=  stmt_list NEWLINE
                  |  NEWLINE INDENT statement+ DEDENT
statement      ::=  stmt_list NEWLINE
                  |  compound_stmt
stmt_list      ::=  simple_stmt (";" simple_stmt)* [";"]
```



Il y a quelque chose de bizarre dans cette grammaire, le trouverez-vous?

Des blocs par indentation

- ▶ En Python, les blocs de commandes sont définis grâce à l'indentation.

```
if x > 0
    print 'Ici, x est positif.'
    y = -x
else
    print 'Ici, y est négatif.'
```

Les valeurs de Python

- ▶ Toute donnée est un objet en Python.
- ▶ Un objet est composé d'une identité, d'un type et d'une valeur.
- ▶ Les types de valeurs prédéfinis de Python sont :
 - ▶ les nombres :
 - ▶ les entiers (machine, à précision arbitraire, booléens) ;
 - ▶ les nombres complexes ;
 - ▶ les flottants ;
 - ▶ les séquences :
 - ▶ non modifiables : chaîne de caractères, chaîne UNICODE, n-uplet ;
 - ▶ modifiables : listes, tableaux ;
 - ▶ les ensembles modifiables ou non ;
 - ▶ les structures associatives : dictionnaires, tables de base de données ;
 - ▶ les données exécutable (les fonctions, les méthodes, les classes, ...) ;
 - ▶ les modules, les fichiers, ...

[DEMONSTRATION]

Définition de hiérarchie de classes en Python

```
# ! /usr/bin/env python
```

```
import sys
```

```
import os
```

```
import os.path
```

```
class FileSystemNode
```

```
    def __init__ (self, name)  
        self.__name = name
```

```
    def name (self)    return self.__name
```

[DEMONSTRATION]

Définition de hiérarchie de classes en Python (suite)

```
class File (FileSystemNode):
    def __init__ (self, name) :
        FileSystemNode.__init__ (self, name)
        try:
            self.__fd = open (name)
            self.__mode = True
        except (IOError, OSError):
            self.__mode = False
            print 'Error during file loading ', name

    def isleaf (self) : return True
    def children (self) : return []
    def file (self): return self.__fd
    def mode (self): return self.__mode

    def close (self):
        if self.mode ():
            self.__fd.close ()
            self.__mode = False
```

[DEMONSTRATION]

Définition de hiérarchie de classes en Python (suite)

```
class Directory (FileSystemNode):
    def __init__ (self, name) :
        FileSystemNode.__init__ (self, name)
        try:
            self._members = map (self.load_member, os.listdir (name))
        except (IOError, OSError):
            print 'Error while loading directory ', name

    def load_member (self, name):
        full_name = os.path.join (self.name (), name)
        if os.path.isdir (full_name):
            return Directory (full_name)
        else:
            f = File (full_name)
            f.close ()
            return f
```

[DEMONSTRATION]

Des fonctions très polymorphes grâce au typage dynamique

- L'absence de typage fort permet d'écrire des fonctions très générales :

```
def show_with_parenthesis (x)  
    print '(', x.show (), ')'
```

[DEMONSTRATION]

- ... mais encore une fois, sans aucune garantie.

Programmation dirigée par les types dynamiques

- Comme les types sont des données, il est possible d'écrire des fonctions qui observent le type de leur argument et agissent différemment en fonction de ce type.

```
class magic_array (dict):
    def __init__ (self):
        dict.__init__ (self)
        self._named_position = {
            'first' : lambda (self): self[0],
            'second' : lambda (self): self[1],
            'third' : lambda (self): self[2]
        }

    def __getitem__ (self, x):
        if type(x) == int:
            return dict.__getitem__ (self, x)
        elif type(x) == str:
            try: return self._named_position[x] (self)
            except: raise TypeError
        else:
            raise TypeError

x = magic_array ()
x[0] = 'foobar'
print x['first']
print x['fourth']
print x[True]
```

Liste définie par compréhension en Python

- On peut définir une liste en décrivant les éléments qui la composent à l'aide d'un générateur.

```
def cartesian_product (l1, l2)    return [ (x, y) for x in l1 for y in l2 ]  
                                # { x,y | x ∈ l1 ∧ y ∈ l2 }
```

```
def diagonal (p)    return [ x for (x, y) in p if x == y ]  
                    # {x | x,y ∈ p ∧ x == y}
```

Conclusion temporaire sur Python

- ▶ Python est un langage orienté objet qui offre des mécanismes très simples et une vue unifiée de la programmation par objets.
- ▶ Le typage dit dynamique n'a rien à voir avec le typage statique.
- ▶ Il s'agit d'une **décoration des données**, des **méta-données**, permettant à l'interprète de détecter les erreurs . . .
- ▶ . . . mais il est déjà trop tard ! Les erreurs se sont déjà produites !
- ▶ Par contre, nous verrons dans le cours sur l'interopérabilité dans quelle mesure ces méta-données facilitent l'adaptation et l'extensibilité automatique des composants.

Une unique référence pour patienter

- ▶ <http://www.python.org>

Scala

Présentation

- ▶ `Scala` est un langage fonctionnel orienté objet statiquement typé à base de classes et de `traits`.
- ▶ Il est développé par Martin Odersky de l'École Polytechnique Fédérale de Lausanne.
- ▶ La version 1.0 a été publiée en 2003, la version 2.0 a été publiée en 2006.
- ▶ `Scala` se compile vers `DotNet` et vers la `JVM`. Il interagit très bien avec les bibliothèques existantes sur ces deux plateformes.
- ▶ On peut utiliser `Scala` dans une boucle d'interaction.

Valeurs et types du langage

- ▶ Toute valeur est un objet.
- ▶ En tant que langage fonctionnel, Scala propose des fonctions de première classe, qui peuvent être vues comme des objets.

Définition de classes

```
class A (arg_1  U_1, ..., arg_k  U_k) extends B {  
  val x = ...  
  def my_method (x_1  T_1, ..., x_n  T_n)  T_r = ...  
  type T = ...  
  class A = ...  
}
```

(Les U et les T sont des types.)

- ▶ L'héritage est **simple**.
- ▶ Le langage intègre une **inférence des types partielle**. On peut omettre certains types lorsqu'ils sont déductibles du contexte.

Exemple

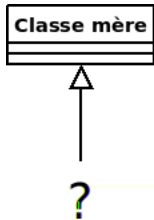
```
class File {  
  def read    int = ...  
  def write (x int)    unit = ...  
}  
  
object the_program {  
  def main (args    Array[String]) = {  
    val x = new File;  
    x.write (42);  
    x write 42; // Syntaxe équivalente.  
  }  
}
```

- Le mot-clé **object** permet d'implémenter le patron de conception **singleton**. La classe est instanciée lors de sa première utilisation et c'est ensuite toujours la même instance qui est utilisée.

Les **traits**, un remplacement sain de l'héritage multiple

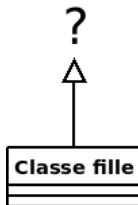
- ▶ L'héritage multiple est utile pour joindre des hiérarchies de classes découplées.
- ▶ Cependant, il est souvent difficile à implémenter pour le concepteur de langage et il est parfois trop peu expressif pour certaines compositions.
- ▶ Scala importe (dans un cadre de type statique) la **composition mixin** du langage SmallTalk pour généraliser la notion même d'héritage.

L'asymétrie de la composition par héritage



- En programmation objet traditionnelle, on développe une classe en pensant à ses possibles extensions sous la forme de sous-classes.

L'asymétrie de la composition par héritage



- Pourquoi ne pas développer une extension **sans savoir ce que l'on est en train d'étendre** ?

Exemple en C++

```
struct File {  
    virtual void write (int x) { ... }  
};
```

```
struct SyncFile : public File {  
    void acquireLock () {}  
    void releaseLock () {}  
    virtual void write (int x) { acquireLock () ; File::write (x) ; releaseLock () ; }  
};
```

```
struct Stream {  
    virtual void write (int x) { ... }  
};
```

```
struct SyncStream : public Stream {  
    void acquireLock () {}  
    void releaseLock () {}  
    virtual void write (int x) { acquireLock () ; Stream::write (x) ; releaseLock () ; }  
};
```



Comment factoriser le code de synchronisation? ^a

^aExemple tiré de « Adding traits to (statically typed) languages »
Oscar Nierstrasz, Stéphane Ducasse, Stefan Reichhart and Nathanael Schärli

Une tentative en C++

```
struct File {  
    virtual void write (int x) { }  
};  
  
struct Stream {  
    virtual void write (int x) { }  
};  
  
struct Sync {  
    void acquireLock () {}  
    void releaseLock () {}  
    virtual void directWrite (int x) = 0;  
    virtual void write (int x) { acquireLock (); this→directWrite (x); releaseLock (); }  
};  
  
struct SyncFile : public File, Sync {  
    virtual void write (int x) { this→Sync::write (x); }  
    virtual void directWrite (int x) { this→File::write (x); }  
};  
  
struct SyncStream : public Stream, Sync {  
    virtual void write (int x) { this→Sync::write (x); }  
    virtual void directWrite (int x) { this→Stream::write (x); }  
};
```


Une solution grâce aux templates de C++

```
struct File {  
    virtual void write (int x) { }  
};
```

```
struct Stream {  
    virtual void write (int x) { }  
};
```

```
template <class Super>  
struct Sync : public Super {  
    void acquireLock () {}  
    void releaseLock () {}  
    virtual void write (int x) { acquireLock (); this→Super::write (x); releaseLock (); }  
};
```

```
struct SyncFile : public Sync<File> {};
```

```
struct SyncStream : public Sync<Stream> {};
```

Mais ...

- ▶ Les classes templates et l'héritage multiple n'ont pas été prévus pour répondre aux problèmes de la composition mixin.
- ▶ Certaines irrégularités du langage rendent périlleux le passage à l'échelle de cette méthode.
- ▶ Par exemple, il n'y a aucune vérification que le paramètre Super de la classe Sync possède bien une méthode write.
- ▶ C'est seulement lorsque la classe Sync est utilisée que la vérification du code de Sync est effectuée.

Les traits en Scala

- ▶ Le mécanisme central du système de types de Scala est la composition mixin.
- ▶ Les traits sont des composants logiciels ne contenant que des méthodes (pas d'attributs). Certaines méthodes sont définies tandis que d'autres sont seulement déclarées.
- ▶ La classe mère d'un trait est indéterminée.
- ▶ Le mot-clé `with` est utilisée pour insérer un trait dans une hiérarchie de classes. C'est à cette occasion que la classe mère du trait est instanciée dans le code des méthodes.

Solution en Scala

```
class File {  
  def write (x: Int) : Unit = println ("File.write")  
}  
  
class Stream {  
  def write (x: Int) : Unit = println ("Stream.write")  
}  
  
trait Writable {  
  def write (x: Int) : Unit  
}
```

// "extends" ne fixe pas une classe pour `super`, il contraint
// juste la future classe à posséder une méthode write.

```
trait Sync extends Writable {  
  def acquire_lock : Unit = println ("Acquire")  
  def release_lock : Unit = println ("Release")  
  
  abstract override def write (x : Int) : Unit = {  
    this.acquire_lock;  
    super.write (x);  
    this.release_lock  
  }  
}
```

```
class SyncA extends A with Sync {}  
class SyncB extends B with Sync {}
```

Le traitement des listes en Scala

- Comme en Python ou Haskell, on peut construire des listes par **compréhension** :

```
def cartesian_product (l1 List[int], l2 List[int]) List[(int, int)] =  
  for (val x ← l1 ; val y ← l2) yield (x, y)
```

```
def main (args Array[String]) {  
  val l1 = List(1, 2, 3)  
  val l2 = List(1, 2, 3)  
  for (x ← cartesian_product (l1, l2))  
    println (x)  
}
```

- « **val** $x \leftarrow l1$ » se lit « $x \in l1$ ».

Le pattern matching en Scala

- ▶ Définir une fonction par cas est une pratique très courante en informatique.
- ▶ Les langages de la famille ML facilitent cela à l'aide des types algébriques.
- ▶ Typiquement, on définit des fonctions par induction sur la structure d'un arbre ainsi :

```
type color = Red | Black
```

```
type tree = Empty | Node of color × tree list
```

```
let rec is_red_node = function
```

```
| Empty | Node (Black, _) → false
```

```
| Node (Red, children) → List.for_all is_black_node children
```

```
and is_black_node = function
```

```
| Node (Red, children) → false
```

```
| Empty → true
```

```
| Node (Black, children) → List.for_all is_red_node children
```

```
let is_redblack_tree = is_black_node
```

Être un arbre rouge-noir en Scala

```
abstract class Color
case class Red () extends Color
case class Black () extends Color
```

```
abstract class Tree
case class Empty () extends Tree
case class Node (val c: Color, val children: List[Tree]) extends Tree
```

```
object RedBlackTree {
  def is_red_node (x: Tree) : boolean =
    x match {
      case Empty () => false
      case Node (Red (), children) => false
      case Node (Black (), children) => children forall this.is_black_node
    }
  def is_black_node (x: Tree) : boolean =
    x match {
      case Empty () => true
      case Node (Red (), children) => children forall this.is_red_node
      case Node (Black (), children) => false
    }
  def is_redblack_tree : Tree => boolean = this.is_black_node
}
```


Conclusion préliminaire sur Scala

- ▶ Scala introduit une composition plus générale que l'héritage, la composition **mixin**
- ▶ Cette composition permet de **découpler** le développement des classes.
- ▶ La sémantique de Scala est très claire et formalisée.
- ▶ Ce langage a importé des mécanismes provenant du monde fonctionnel et du monde objet.
- ▶ Nous verrons que le système de types de ce langage est particulièrement adapté à la programmation générique.

O'Caml

Présentation

- ▶ O'Caml est un langage fonctionnel proposant un système d'objets et un système de modules pour organiser les composants.
- ▶ Il est développé par l'équipe Gallium de l'INRIA.
- ▶ Il s'exécute dans un environnement intégrant un ramasse-miette.
- ▶ Une boucle d'interaction est fournie.

Les valeurs et les types du langage

- ▶ Les valeurs du langage sont formées par les valeurs de base (entiers, flottants, ...), les valeurs de type algébrique (liste, arbre, ...), les fonctions, les enregistrements, les objets ...
[DEMONSTRATION]
- ▶ Les types sont stratifiés en deux couches : les types monomorphes et les schémas de types.
- ▶ Le langage intègre une inférence de type capable de calculer le type le plus général d'un programme sans annotations.

Un objet est un enregistrement

- ▶ En première approximation, un objet est vue comme un enregistrement défini récursivement.
- ▶ La syntaxe pour construire un objet est :

object

val x = 42

val y = 37

method x = x

method y = y

method show = "I am a point !"

end

- < show string; x int; y int > = <obj>

- ▶ Les appels de méthode sont de la forme : obj#method e_1 ... e_n
- ▶ Exemple : point#show

Exemple de création d'objets

```
# let minmax x y =  
  if x < y then object  
    method min = x  
    method max = y  
  end else object  
    method max = x  
    method min = y  
  end ;;  
val minmax :  $\alpha \rightarrow \alpha \rightarrow < \text{max} : \alpha ; \text{min} : \alpha > = < \text{fun} >$   
  
# let last_memo f = object (self)  
  val mutable last = []  
  method apply x =  
    let y = f x in  
    last  $\leftarrow$  y :: last ;  
    y  
  method last = last  
  method reset = last  $\leftarrow$  [] ; self  
end ;;  
val last_memo :  
  ( $\alpha \rightarrow 'b$ )  $\rightarrow$  (< apply :  $\alpha \rightarrow 'b$  ; last : 'b list ; reset : 'c > as 'c) =  
  <fun>
```

Le polymorphisme sur la partie inconnue des objets

- ▶ Le sous-typage des objets en O'Caml est **structurel**.
- ▶ Il est implémenté grâce au **polymorphisme de rangées**, qui est une **quantification sur la partie inconnue** de l'objet.

```
# let display p = Printf.printf "(%d, %d) [%s]" p#x p#y p#show;;  
val display    < show    string; x    int; y    int; .. > → unit = <fun>
```

- ▶ La fonction display est compatible avec tous les objets qui possèdent les méthodes x, y et show.
- ▶ La notation .. signifie qu'il existe une partie inconnue de l'objet par rapport à laquelle on est polymorphe.

Une classe est un générateur d'objets dont on peut hériter

- ▶ Les classes sont utilisées pour produire des objets et réutiliser le code d'autre classe par le biais de l'héritage.

```
class point x y = object
  val x = x
  val y = y
  method x = x
  method y = y
  method show = "Hi ! I am a point !"
end

class color_point x y (c : ['Red | 'Black]) = object
  inherit (point x y)
  val color = c
  method color = color
  method show = "Hi ! I am a color point !"
end
```

- ▶ L'héritage peut être **multiple**.
- ▶ On retrouve les notions habituelles de classe et de méthodes abstraites.

Les types de classe

- ▶ Parfois, l'inférence de type infère un type trop général.
- ▶ Il est donc nécessaire de le contraindre à l'aide d'annotation de type ou à l'aide de **type de classe**

```
class type point_type = object
  val x    int
  val y    int
  method x  int
  method y  int
  method show  string
end
```

```
class point x y    point_type = object
  val x = x
  val y = y
  method x = x
  method y = y
  method show = "Hi ! I am a point !"
end
```

Les coercions explicites

- ▶ Le sous-typage en présence de polymorphisme paramétrique rend complexe l'inférence des types.
- ▶ Pour simplifier ce problème, O'Caml ne choisit pas implicitement un type plus général lorsqu'il a le choix entre plusieurs types.

```
# let p = new point 0 0;; let pc = new color_point 0 0 'Red;;  
val p : point = <obj>  
val pc : color_point = <obj>  
# [p; pc];;  
Characters 4-6:  
  [p; pc];;  
    ^
```

This expression has **type** color_point but is here used **with type** point
The second **object type** has no **method** color

- ▶ Dans ces cas, la conversion de type doit être fait explicitement par le programmeur en utilisant une **coercion** explicite :

```
# [p; (pc : > point)];;  
- : point list = [<obj>; <obj>]
```

Les objets fonctionnels

- ▶ O'CamL simplifie la définition d'objets **fonctionnels**.
- ▶ L'état (les attributs) de l'objet ne sont pas modifiables.
- ▶ Un message est alors vu comme une fonction produisant un objet à partir d'un autre.

```
#class functional_point y =  
  object  
    val x = y  
    method get_x = x  
    method move d = {< x = x + d >}  
  end;;  
class functional_point :  
  int →  
    object (α) val x : int method get_x : int method move : int → α end  
#let p = new functional_point 7;;  
val p : functional_point = <obj>  
#p#get_x;;  
- : int = 7  
#(p#move 3)#get_x;;  
- : int = 10  
#p#get_x;;  
- : int = 7
```

(source : <http://caml.inria.fr/pub/docs/manual-ocaml/manual005.html>)