

Introduction  
à la programmation OO (Orientée Objet)  
et au langage CRYSTAL 1.18.2  
(Paradigmes de Programmation – Ensimag 1A)

Avec hyperliens (soulignés) à suivre  
surtout ceux vers les docs CRYSTAL comme ici  
ou vers les exemples du cours : [08class\\_inheritance.cr](#)

Sylvain Boulmé

## Plan du cours

CRYSTAL dans le bestiaire des langages OO

Le typage statique de CRYSTAL

Introspection du demi-treillis des types de CRYSTAL

Classes et héritage (en CRYSTAL)

Illustrations avec la classe générique `Array(T)` prédéfinie

Conclusion

## Définitions *informelles* de la prog. OO

- ▶ Programmes constitués d'*objets* communiquant par appels de *méthodes*;
- ▶ un tel *objet* encapsule des données dans des *attributs* et porte des *méthodes* pour accéder ou modifier ces données;
- ▶ les *méthodes* sont des procédures (ou fonctions) accessibles uniquement via un *objet*.

### Exemple de syntaxe usuelle (e.g. en PYTHON ou JAVA)

- ▶ `o.x = 3` affecte 3 à l'*attribut* `x` de l'*objet* `o`.
- ▶ `o1.foo(o2)` appelle la *méthode* `foo` de l'*objet* `o1` avec `o2` comme argument.

↪ ressemble à des enregistrements (*record/struct*)  
 sauf que sélection du `foo` souvent basée sur *typage dynamique*.

## Une brève histoire des langages OO

Paradigme très dominant depuis années 1990, avec notamment  
C++ et EIFFEL puis JAVA  
mais initié dans années 1970 par SIMULA, SMALLTALK et CLU.

Depuis SMALLTALK-80, les langages OO sont très dynamiques,  
permettant aux programmes de *s'introspecter* voire de  
*s'auto-modifier* (cf. réflexion).  
Ça nécessite du **typage dynamique**.

## Langages OO modernes ont typage dynamique **et** statique

Typage de RUBY et PYTHON essentiellement **dynamique**  
même si *typage statique graduel*  
pour détecter plus de bugs statiquement (avant les tests)

Voir motivations de RUBY pour typage statique  
et celles de PYTHON pour se limiter à du typage graduel.

JAVA impose un **typage statique fort** qui garantit que l'erreur  
`'o1' object has no method 'foo'`  
n'arrive jamais à l'exécution (car *toujours* détectée statiquement).

JAVA permet des “contournements” explicites du typage statique,  
via conversions ponctuelles de types disant au compilateur :

*fais moi confiance sur cette conversion ici, déclenche une  
erreur à l'exécution si je me trompe*

## CRYSTAL (apparu en 2014) versus JAVA et RUBY

Un langage OO pur (i.e. *tout est objet*) inspiré de RUBY mais avec **typage statique fort** qui permet à son compilateur, basé sur LLVM, de produire du *code très efficace*.

Typage statique plus expressif que celui de JAVA :  
e.g. type statique de variables qui change sous certaines branches de if-then-else ou après certaines affectations.

Contrairement à JAVA, typage statique *non compositionnel* :  
garanties uniquement sur programme **complet**.

Typage méthodes  $\neq$  contrat (vérifié sur la définition)

↳ juste une restriction sur les contextes d'appels autorisés.

↪ typage explicite **optionnel** (ie "duck typing statique").

## Premiers pas en CRYSTAL

Deux lignes de commande pour compiler `prog.cr` puis l'exécuter

```
crystal build prog.cr # compile prog.cr en ./prog
./prog                # exécution du binaire
```

Ou une seule ligne (sans créer "prog" dans répertoire courant)

```
crystal prog.cr
```

Consultez le Crystal Book  
et en particulier manuel de référence du langage

Ce cours fournit des exemples  
[essayez les](#), avec les modifications suggérées pour en voir l'effet !

# Plan du cours

CRYSTAL dans le bestiaire des langages OO

**Le typage statique de CRYSTAL**

Introspection du demi-treillis des types de CRYSTAL

Classes et héritage (en CRYSTAL)

Illustrations avec la classe générique `Array(T)` prédéfinie

Conclusion

## Typage non compositionnel (très spécifique à CRYSTAL)

### Exemple 01static\_typing.cr

```

1 def pay(x : Int32) : Nil
2   if x < 0
3     bug(x) # nom pas défini, devrait échouer !
4   end
5 end
6 puts("Hello!")
7 puts("Hey, type hello please :)")
8 answer : String | Nil = gets
9 if answer == "hello please :)"
10  puts("Congrat, you win 1000$...")
11  # pay(1000) # <- commenter cette ligne provoque
12             # une erreur de typage statique
13             # même si pas d'erreur dynamique pour x >= 0
14 elsif answer == nil
15  puts("Please, enter something !")
16 else
17  puts("Sorry, \"#{answer}\" is a bad answer")
18 end

```

NB : fonctions *implicitement* mutuellement récursives,  
cf 02mutualrecursion.cr.

## Typage explicite (généralement) optionnel mais recommandé pour messages d'erreurs plus clairs !

### Exemple 03optional\_typing.cr

```

1  def foo(x)
2    puts("15 est-il divisible par #{x} ?", 15 % x == 0)
3  end
4  u = 5
5  foo(u)
6  u /= 2 # équivalent à u = u / 2
7  foo(u)

```

Message d'erreur dans la **déclaration** de foo :

```
2 | puts("15 est-il divisible par #{x} ?", 15 % x == 0)
```

Error: expected argument #1 to 'Int32#%' to be Int, not Float64

Typage explicite (généralement) optionnel  
mais recommandé pour messages d'erreurs plus clairs !

**Restriction** de type (uniquement) sur argument de foo

```

1 def foo(x : Int32)
2   puts("15 est-il divisible par #{x} ?", 15 % x == 0)
3 end
4 u = 5
5 foo(u)
6 u /= 2  # équivalent à u = u / 2
7 foo(u)

```

Message d'erreur sur le **deuxième** appel à foo :

```
7 | foo(u)
   ^
```

Error: expected argument #1 to 'foo' to be Int32, not Float64

## Typage explicite (généralement) optionnel mais recommandé pour messages d'erreurs plus clairs !

### Restriction de type (uniquement) sur **déclaration** de u

```

1 def foo(x)
2   puts("15 est-il divisible par #{x} ?", 15 % x == 0)
3 end
4 u : Int32 = 5
5 foo(u)
6 u /= 2 # équivalent à u = u / 2
7 foo(u)

```

NB : **déclaration** de u = première affectation !

Message d'erreur sur la **deuxième** affectation à u :

```

6 | u /= 2 # équivalent à u = u / 2
  ~

```

Error: type must be Int32, not (Float64 | Int32)

# Plan du cours

CRYSTAL dans le bestiaire des langages OO

Le typage statique de CRYSTAL

**Introspection du demi-treillis des types de CRYSTAL**

Classes et héritage (en CRYSTAL)

Illustrations avec la classe générique `Array(T)` prédéfinie

Conclusion

## Objets et types en CRYSTAL

Toute donnée est un *objet* qui a (au moins) un *type*.

Un tel *type* représente un ensemble d'*objets*.

### Exemples de *type de base*

- ▶ Int32 type des entiers signés 32 bits comme 42 ;
- ▶ UInt8 type des entiers signés 8 bits comme 42u8 ;
- ▶ Float64 type des flottants 64 bits comme 3.14 ;
- ▶ String type des chaînes de caractère comme "Hello" ;
- ▶ Nil type qui a comme unique élément `nil` (une valeur similaire au `None` de PYTHON) ;
- ▶ NoReturn type vide pour expliciter qu'une fonction comme `raise` ne termine pas normalement, cf :

```
def internal_error (name : String) : NoReturn
  raise "Unexpected error #{name}, please report!"
end
```

- ▶ `class` le type des types, comme `Int32`, `String` et `class` lui-même !

## Unions binaires de deux types (quelconques)

Opérateur infixe `|`, retournant union binaire de 2 types qqc, associatif, commutatif, idempotent, et avec `NoReturn` comme élément neutre. Exemples (déjà vus) :

- ▶ `Float64 | Int32` type inféré par le compilateur dans le 3<sup>e</sup> message d'erreur à la diapo 10.
- ▶ `String | Nil` type vu à la diapo 9 qui correspond en fait au type de retour de la fonction `gets` où :
  - ▶ valeur `nil` (de type `Nil`) retournée si l'utilisateur·ice tape les touches `Ctrl+D` (terminant ainsi l'entrée standard).
  - ▶ à distinguer de chaîne vide `""` (de type `String`) retournée si l'utilisateur·ice tape un retour à la ligne sans rien taper d'autre.

NB : on peut abrégé ce type en `String?`

ou plus généralement `A?` pour `A | Nil`

(aka un type *option* comme en `HASKELL`, `OCAML` ou `RUST`).

## Opérateur de sous-typage

Opérateur infix `<` qui teste l'inclusion (stricte) entre types

- ▶ `Int32` et `Float64` sous-types de `Float64 | Int32`  
 e.g. `Int32 < (Float64 | Int32)`  
**NB** conversion des entiers en flottants  $\neq$  sous-typage  
 car une telle conversion *change* la représentation en binaire.
- ▶ `Number` est un sur-type de tous les types de nombres :  
`(Float64 | Int32 | UInt8) < Number`
- ▶ `Object` est le type maximum, c'est le sur-type de tous les types :  
`(Int32 | Float64 | String?) < Object`  
**NB** on a `Class < Object` et `Object : Class`
- ▶ `NoReturn` est le type minimum, sous-type de tous les types :  
`NoReturn < Int32`

## Type statique versus type dynamique

- ▶ Tout *objet* `o` a un unique *type dynamique*, noté `o.class`, fixé définitivement à sa création dans l'exécution courante.
- ▶ Tout *objet* `o` a un *type statique*, noté `typeof(o)`, qui est un **sur-type du type dynamique** que le compilateur est capable de garantir **pour toute exécution** (en un point d'exec fixé).

### Exemple [04subtyping.cr](#)

```
def inspect(x : Object) : Nil
  puts("(#{x}) of #{x.class} <= #{typeof(x)} ")
end

[3.14, 25, Int32, "hello", nil].each { |x| inspect(x) }
```

## Type statique versus type dynamique

- ▶ Tout *objet* `o` a un unique *type dynamique*, noté `o.class`, fixé définitivement à sa création dans l'exécution courante.
- ▶ Tout *objet* `o` a un *type statique*, noté `typeof(o)`, qui est un **sur-type du type dynamique** que le compilateur est capable de garantir **pour toute exécution** (en un point d'exec fixé).

### Exemple équivalent

```
def inspect(x) : Nil
  puts("(#{x}) of #{x.class} < #{typeof(x)} ")
end
aux : Array(Float64 | Int32 | Int32.class | String | Nil) \
      = [ 3.14 , 25 , Int32 , "hello", nil ]
aux.each { |x| inspect(x) }
```

## Type statique versus type dynamique

- ▶ Tout *objet*  $\circ$  a un unique *type dynamique*, noté  $\circ$ .`class`, fixé définitivement à sa création dans l'exécution courante.
- ▶ Tout *objet*  $\circ$  a un *type statique*, noté `typeof`( $\circ$ ), qui est un **sur-type du type dynamique** que le compilateur est capable de garantir **pour toute exécution** (en un point d'exec fixé).

```

1 def inspect(x : Object) : Nil
2   puts("#{x}) of #{x.class} <= #{typeof(x)} ")
3 end
4 [3.14, 25, Int32, "hello", nil].each { |x| inspect(x) }
```

$x$  a type statique (`Float64 | Int32 | Int32.class | String | Nil`)  
son type dynamique étant un des types de base de cette union.

**NB** type statique de  $x$  en ligne 2 dépend du contexte d'appel, car inférence statique de type avec *inlining*, donc *non compositionnelle*.

## Introspection de type et typage statique flot de donnée

Au lieu de `if o.class <= Number`,  
 préférez `if o.is_a?(Number)` avec `o` dans une variable  
 pour effet de bord (à la compilation) très utile sur `typeof(o)`.

Exemple [05type\\_introspection.cr](#)

```
def dual(x : Bool | Float64) : Nil
  if x.is_a?(Number)
    r = -x # ici x et r de type statique Float64
  else
    r = !x # ici x et r de type statique Bool
  end
  inspect(r) # ici r de type statique Bool | Float64
end
[3.14, false].each { |x| dual(x) }
```

## La surcharge, une alternative à l'inspection de type

Surcharge = méthodes de même nom qui coexistent pour différents *types statiques* de paramètre.

En CRYSTAL : sélection de la méthode par dynamic multiple dispatch c-à-d. en fonction des params formels ayant des *sur-types statiques* les plus "proches" des *types dynamiques* des params effectifs.

Exemple [06overloading.cr](#) (alternative à l'ex. en diapo 16)

```
def dual(x : Float64) : Float64
  -x
end
def dual(x : Bool) : Bool
  !x
end
[3.14, false].each { |x|
  r = dual(x) # ici x de type statique Bool / Float64
  inspect(r)  # ici r de type statique Bool / Float64
}
```

## La surcharge, une alternative à l'introspection de type

Surcharge = méthodes de même nom qui coexistent pour différents *types statiques* de paramètre.

En CRYSTAL : sélection de la méthode par dynamic multiple dispatch c-à-d. en fonction des params formels ayant des *sur-types statiques* les plus "proches" des *types dynamiques* des params effectifs.

Exemple [06overloading.cr](#) (alternative à l'ex. en diapo 16)

```
def dual(x : Float64) : Float64
  -x
end
def dual(x : Bool) : Bool
  !x
end
[3.14, false].each { |x|
  r = dual(x) # ici x de type statique Bool / Float64
  inspect(r) # ici r de type statique Bool / Float64
}
```

**Attention** ambiguïtés potentielles en présence de plusieurs arguments, cf. détails dans [07overloading\\_hell.cr](#).

# Plan du cours

CRYSTAL dans le bestiaire des langages OO

Le typage statique de CRYSTAL

Introspection du demi-treillis des types de CRYSTAL

**Classes et héritage (en CRYSTAL)**

Illustrations avec la classe générique `Array(T)` prédéfinie

Conclusion

## Classe comme constructeur d'objets (cf. [08class\\_inheritance.cr](#))

```

class Person
  @name : String                # Exemple d'attribut
  def initialize(n : String) : Nil # Méthode constructeur
    @name = n                  # cf. (3) ci-dessous
  end
  def get_id : String           # Méthode (ordinaire)
    @name
  end
  def change(n : String) : Nil  # Méthode (ordinaire)
    @name = n
    puts("Now I'm #{self.get_id}") # self = objet courant
  end
end

```

- (1) Def. type `Person` dont objets `o` “répondent” à `o.get_id` et `o.change`.
- (2) Un tel `o` a un `@name` *personnel* uniquement visible en interne  
= **état encapsulé**, uniquement accessible par ses méthodes.
- (3) Def. **méthode** `new` de `Person` qui alloue `o` puis appelle `o.initialize`

## Classe comme constructeur d'objets (cf. [08class\\_inheritance.cr](#))

```

class Person
  @name : String           # Exemple d'attribut
  def initialize(n : String) : Nil # Méthode constructeur
    @name = n             # cf. (3) ci-dessous
  end
  def get_id : String      # Méthode (ordinaire)
    @name
  end
  def change(n : String) : Nil # Méthode (ordinaire)
    @name = n
    puts("Now I'm #{self.get_id}") # self = objet courant
  end
end

```

```

bob = Person.new("Bob") # crée un nouvel objet
tom = Person.new("Tom") # crée un nouvel objet
puts(bob.get_id)        # un appel de méthode
bob.change("Alice")     # un autre appel

```

- (1) Def. type `Person` dont objets `o` “répondent” à `o.get_id` et `o.change`.
- (2) Un tel `o` a un `@name` *personnel* uniquement visible en interne  
= **état encapsulé**, uniquement accessible par ses méthodes.
- (3) Def. **méthode** `new` de `Person` qui alloue `o` puis appelle `o.initialize`

## Classe comme constructeur d'objets (cf. [08class\\_inheritance.cr](#))

```

class Person
  @name : String           # Exemple d'attribut
  def initialize(n : String) : Nil # Méthode constructeur
    @name = n              # cf. (3) ci-dessous
  end
  def get_id : String      # Méthode (ordinaire)
    @name
  end
  def change(n : String) : Nil  # Méthode (ordinaire)
    @name = n
    puts("Now I'm #{self.get_id}") # self = objet courant
  end
end

```

```

bob = Person.new("Bob")      # crée un nouvel objet
tom = Person.new("Tom")     # crée un nouvel objet
puts(bob.get_id)            # un appel de méthode
bob.change("Alice")         # un autre appel

```

```

def meet(x : Person, y : Person) : Nil
  puts("Hi #{y.get_id}, I am #{x.get_id}")
end
meet(tom, bob) # puts "Hi Alice, I am Tom"

```

## Héritage comme spécialisation de type (cf. [08class\\_inheritance.cr](#))

```

class AgedPerson < Person
  @age : Int32
  def initialize(n : String) : Nil
    @age = n.size
    super(n) # super = méthode de la super-classe
  end
  def get_age : Int32 # nouvelle méthode
    @age
  end
  def get_id : String # redéfinition
    "#{@name} (#{@self.get_age} years old)"
  end
end # héritage implicite de change (et @name)

```

Def AgedPerson sous-type de Person tq (Person | AgedPerson) == Person.

Pour o : Person, l'action o.get\_id dépend de o.class.

```

jane = AgedPerson.new("Jane") # jane.class == AgedPerson
meet(jane, tom) # puts "Hi Tom, I am Jane (4 years old)"
jane.change("John") # puts "Now I'm John (4 years old)"

```

## Rédéfinition versus surcharge (cf. [08class\\_inheritance.cr](#))

```

class OtherAgedPerson < AgedPerson

  # même type que parent => redéfinition
  def change (o : String) : Nil
    @age += 1
    super(o) # super = méthode de la super-classe
  end

  # type différent => surcharge
  def change (x : Int32) : Nil
    @age += x
  end

end

sam = OtherAgedPerson.new("Sam")
meet(tom, sam)      # puts "Hi Sam (3 years old), I am Tom"
sam.change(10)
sam.change("Samy") # puts "Now I'm Samy (14 years old)"

```

**NB** pb de “pessimisme” du typage statique en cas de redéfinitions  
 cf. [09\\_typechecking\\_late\\_binding.cr](#)

# Plan du cours

CRYSTAL dans le bestiaire des langages OO

Le typage statique de CRYSTAL

Introspection du demi-treillis des types de CRYSTAL

Classes et héritage (en CRYSTAL)

**Illustrations avec la classe générique Array(T) prédéfinie**

Conclusion

## Intro à la classe `Array(T)`

Classe paramétrée par un type, e.g. `Array(Int32)` OU `Array(Float64)`  
cf. notion de généricité, approfondie en TP.

Comme en diapo 15, `[jane, tom, sam]` crée un `Array(Person)` car  
`(AgedPerson | Person | OtherAgedPerson) == Person`

Parcours du tableau via méthode `each` (avec bloc en param., cf TP)

```
[jane, tom, sam].each { |p| puts(p.get_id) }
```

Lire/modifier case `i` du tableau `a` par `a[i]`. Parcours équivalent :

```
a = [jane, tom, sam]
(0..2).each { |i| puts(a[i].get_id) }
```

En fait, opérateurs “`[]`” et “`[]=`” utilisés dans “`a[0]`” et “`a[0]=bob`”  
font des appels de méthodes.

**NB** utilisables comme “noms” de méthode dans vos propres classes,  
comme plupart des opérateurs, cf + et \* et ==.

Exemple d'utilisation, cf. [10array\\_class.cr](#)

```
def demo(c : Array(Int32 | Int64 | Float64).class) : Nil
  a = c.new()
  puts("demo on #{a.class}")
  (1..10).each {|x| a.push(x*x)} # insère carrés 1 à 100
  puts(a)
  a[2] -= a[3]
  puts(a)
end
```

Dans ce contexte, `demo(Array(Int32))` affiche :

```
demo on Array(Int32)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[1, 4, -7, 16, 25, 36, 49, 64, 81, 100]
```

Tandis que `demo(Array(Float64))` affiche :

```
demo on Array(Float64)
[1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
[1.0, 4.0, -7.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0, 100.0]
```

# “Instrumentations de code” par héritage et redéfinitions

cf. [10array\\_class.cr](#)

```
class MyArray < Array(Int32)
  def push(n : Int32) : Nil
    super(n)
    super(n+1)
  end
  def []= (i : Int, x : Int32) : Nil
    super(i, x)
    puts("affecte #{x} en case #{i}")
  end
  def [] (i : Int) : Int32
    x = super(i)
    puts("lu #{x} en case #{i}")
    x
  end
end
```

Avec le demo de la diapo 24, `demo(MyArray)` affiche :

```
demo on MyArray
[1, 2, 4, 5, 9, 10, 16, 17, 25, 26, 36, 37, 49, 50, 64, 65, 81, 82, 100, 101]
lu 4 en case 2
lu 5 en case 3
affecte -1 en case 2
[1, 2, -1, 5, 9, 10, 16, 17, 25, 26, 36, 37, 49, 50, 64, 65, 81, 82, 100, 101]
```

# Plan du cours

CRYSTAL dans le bestiaire des langages OO

Le typage statique de CRYSTAL

Introspection du demi-treillis des types de CRYSTAL

Classes et héritage (en CRYSTAL)

Illustrations avec la classe générique `Array(T)` prédéfinie

Conclusion

## Conclusion

Autres notions classiques des langages OO à voir en TP  
Classes et méthodes abstraites, Généricité, Espaces de noms...

## Conclusion

Autres notions classiques des langages OO à voir en TP  
Classes et méthodes abstraites, Généricité, Espaces de noms...

Au delà des aspects “langage”

Importance des techniques de conception.

Vues un peu en TP - surtout en 2A.

## Conclusion

Autres notions classiques des langages OO à voir en TP

Classes et méthodes abstraites, Généricité, Espaces de noms...

Au delà des aspects “langage”

Importance des techniques de conception.

Vues un peu en TP - surtout en 2A.

Limites du typage non-compositionnel

Difficulté à comprendre les erreurs de typage,

Longueur des temps de compil,

Pas de bon serveur LSP.

↪ une “feature wish” pour composer avec ce choix de conception !