O'JACARÉ

une interface objet entre Objective Caml et Java

Emmanuel Chailloux et Grégoire Henry

Equipe PPS (UMR 7126)

Université Pierre et Marie Curie (Paris VI)

`http://www.pps.jussieu.fr`

LMO 2004 - Lille

Sommaire

- Motivations
- Objective Caml
- Comparaison Java / O'Caml
- Interfaces de "bas-niveau" (JNI + external)
- Description d'O'Jacaré :
  - IDL + générateur de code
  - appel Java depuis O'Caml
  - appel O'Caml depuis Java
- Exemples d'utilisation :
  - *Design patterns*
- Discussion et travaux futurs

Motivations

1. augmenter les caractéristiques des langages : Java et O'Caml
   - faciliter l'emploi de bibliothèques
2. sans sacrifier la sûreté : typage statique, GC
   - tout en conservant une efficacité raisonnable
3. sans dénaturer son langage de prédilection

Caractéristiques d'O'Caml

- Langage fonctionnel, exceptions, extension impérative

Caractéristiques d'O'Caml

- Langage fonctionnel, exceptions, extension impérative
- Types de données de haut niveau + filtrage de motifs,

Caractéristiques d'O'Caml

- Langage fonctionnel, exceptions, extension impérative
- Types de données de haut niveau + filtrage de motifs,
- Polymorphisme + typage *implicite*,
  - statiquement typé,
  - inférence de types,
  - types polymorphes (choix du type le plus général)

Caractéristiques d'O'Caml

- Langage fonctionnel, exceptions, extension impérative
- Types de données de haut niveau + filtrage de motifs,
- Polymorphisme + typage *implicite*,
  - statiquement typé,
  - inférence de types,
  - types polymorphes (choix du type le plus général)
- Différents styles de programmation (dans un cadre commun de typage).
  - Programmation orientée objet (structuration par classes),
  - Modules paramétrés (style SML)

Caractéristiques d'O'Caml

- Langage fonctionnel, exceptions, extension impérative
- Types de données de haut niveau + filtrage de motifs,
- Polymorphisme + typage *implicite*,
  - statiquement typé,
  - inférence de types,
  - types polymorphes (choix du type le plus général)
- Différents styles de programmation (dans un cadre commun de typage).
  - Programmation orientée objet (structuration par classes),
  - Modules paramétrés (style SML)

Exemples d'inférence de types

- type fonctionnel :
  `let compose f g = fun x -> f (g x);;`
  $(\alpha \to \beta) \to (\gamma \to \alpha) \to \gamma \to \beta$

- type fonctionnel sur les listes :
  `List.map` : $(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`

- type objet et fonctionnel :
  `let toStringNL o = o#toString() ^"\n";;`
  `< toString : unit` $\to$ `string ; .. >` $\to$ `string`

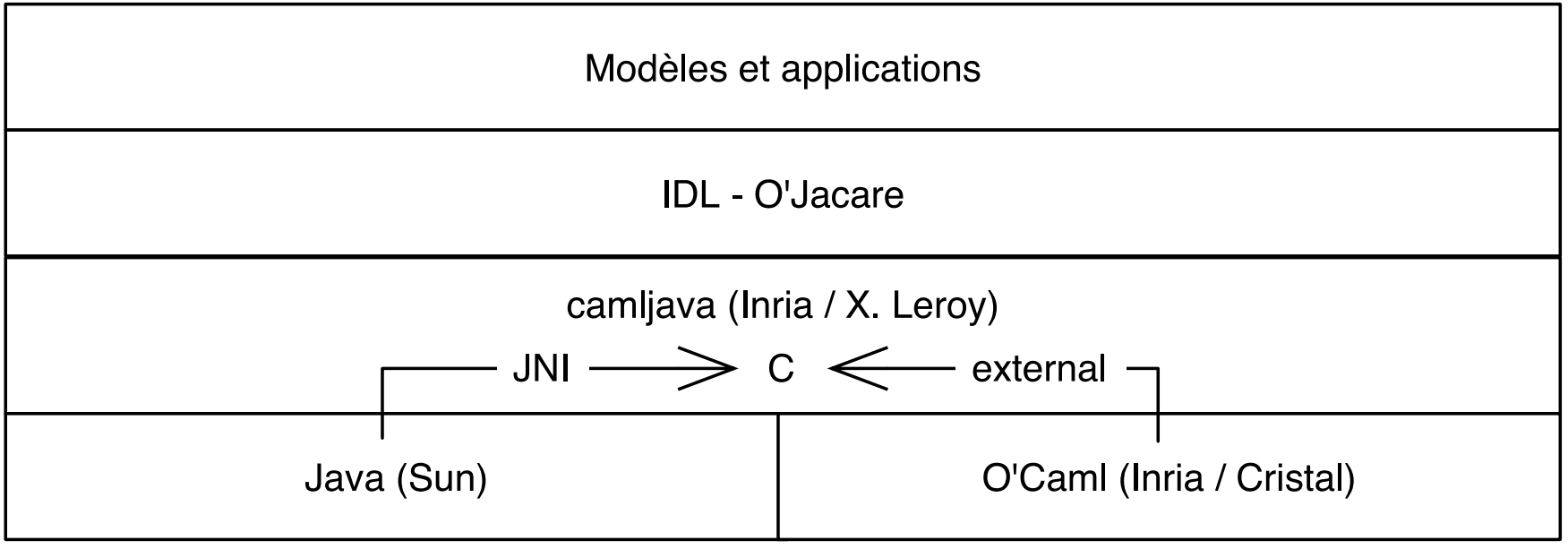
Comparaison Java / O'Caml

caractéristiques	Java	O'Caml	caractéristiques	Java	O'Caml
classes	√	√	sous-typage	√	√
accès champs	√		héritage ≡ sous-typage?	oui	non
liaison tardive	√	√	surcharge	√	
liaison précoce	√		héritage multiple		√
typage statique	√	√	classes paramétrées		√
typage dynamique	√		paquetages/modules	√	√
			classes mutuellement récursives	√	

O'Caml n'est pas un langage objet, mais a une extension objet

- une déclaration de classe définit un type objet et un constructeur;
- type objet = noms et types des méthodes.

Architecture de l'interface



- point d'entrée d'un programme en O'Caml
- compilateur O'Caml : byte-code ou natif
- compilateur Java : byte-code

JNI et bibliothèque `camljava`

Initialisation de la JVM

Manipulation des classes et introspection

- Recherche par nom qualifié
- Identifiant d'une méthode par sa signature

Appel de méthodes

- Appel avec un tableau d'arguments
- Assure la conversion des type de base

Type objet unique `jobject` (C), `Jni.obj` (O'Caml)

Exception Transmise d'un langage à l'autre

GC Les références transmises sont mises en racines

Callback Appel de méthodes O'Caml sans introspection

O'Jacaré, un *IDL* simple - 1/2

Faire correspondre 1 objet Java à 1 objet O'Caml

À l'intersection des deux modèles

- définition de classe, classe abstraite et interface
- héritage simple pour les classes
- héritage multiple pour les interfaces
- pas de surcharge
  (mais un mécanisme d'alias de nom)
- pas de classe paramétrée

O'Jacaré, un générateur de code - 2/2

Passage des arguments

- par partage pour les objets (ex : java.lang.Object)
- par recopie pour les types de base (ex : int, string)

Typage La cohérence des type de l'IDL est vérifié :

- à la compilation avec O'Caml (et en partie avec Java)
- au chargement avec Java (introspection)

En conclusion partielle :

- privilégie le sens d'appel Java depuis O'Caml
- sens inverse via un *callback*

Le fichier d'IDL est une description simplifiée de classe Java

O'Jacaré : Classe Point

Fichier `point.idl`

```
class Point {
    int x;

    [name default_point] <init> ();
    [name point] <init> (int);

    void moveTo(int);
    string toString();
    boolean eq(Point);
}
```

Engendre : `point.ml`

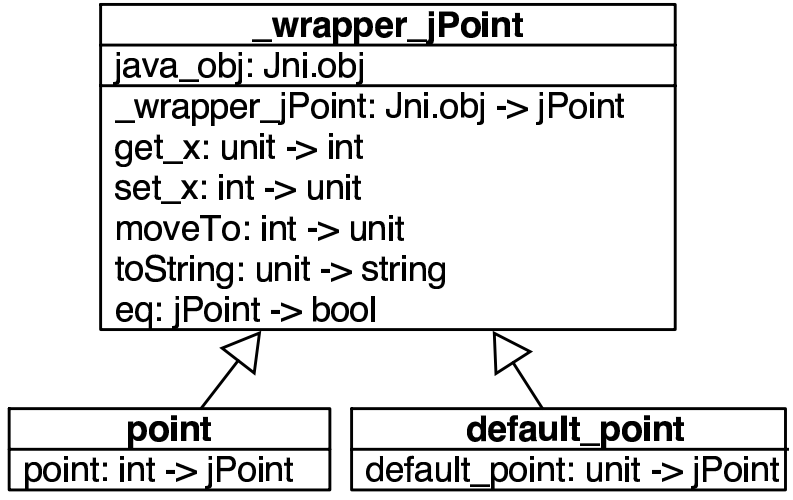
Type abstrait `_jni_jPoint`

Type objet `jPoint`

Capsule `_wrapper_jPoint`

Classes utilisateurs

`default_point`, `point`

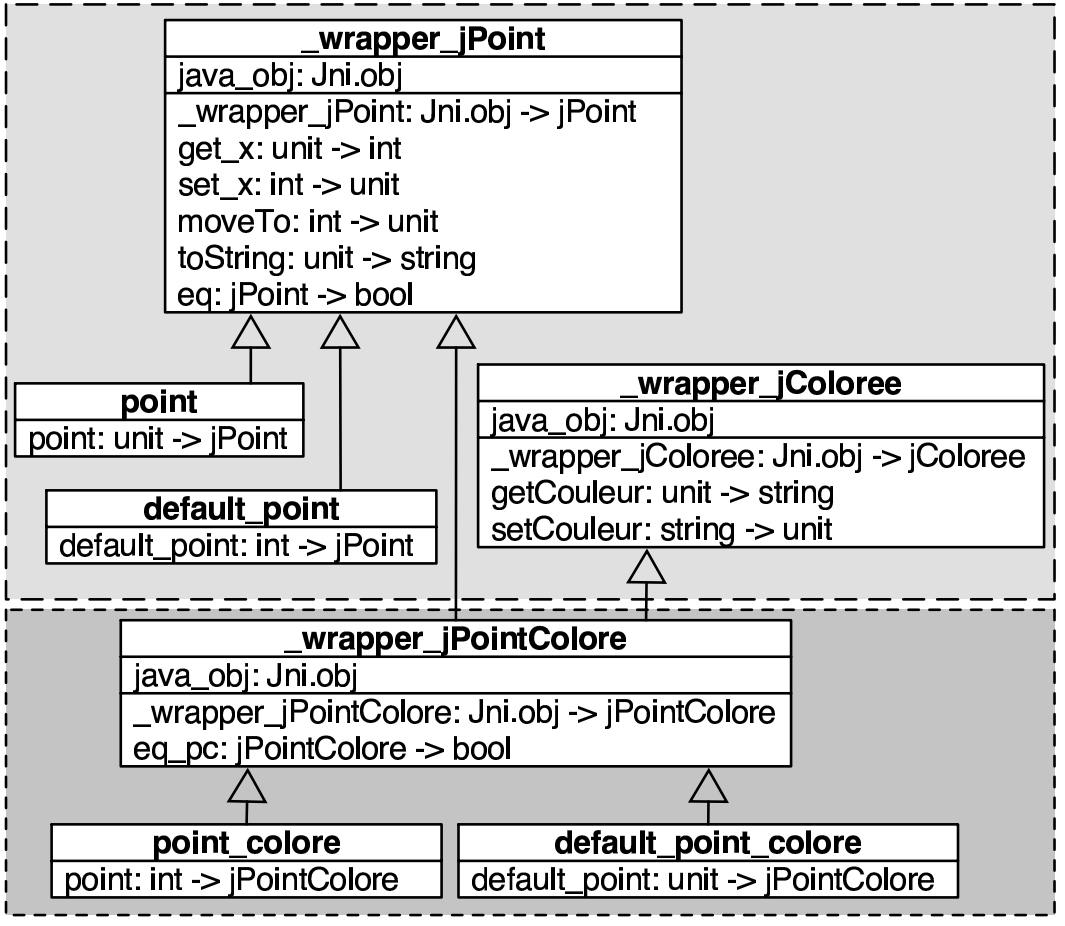


O'Jacaré : Classe PointColore

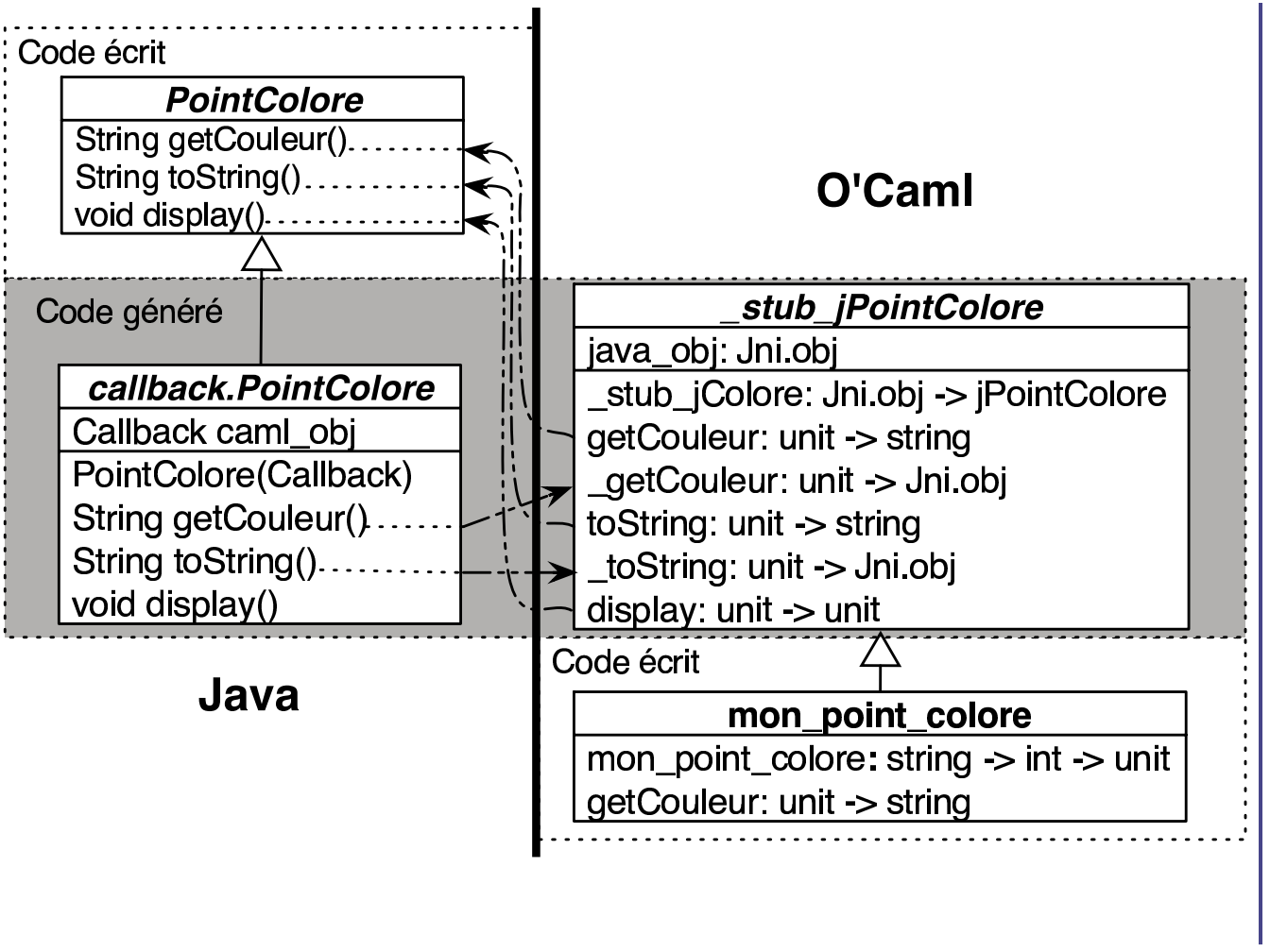
Fichier `point.idl`

```
class PointColore extends Point
    implements Coloree {

    [name default_point_colore]
        <init> ();

    [name point_colore]
        <init> (int,string);

    [name eq_pc]
        boolean eq(PointColore);
}
```

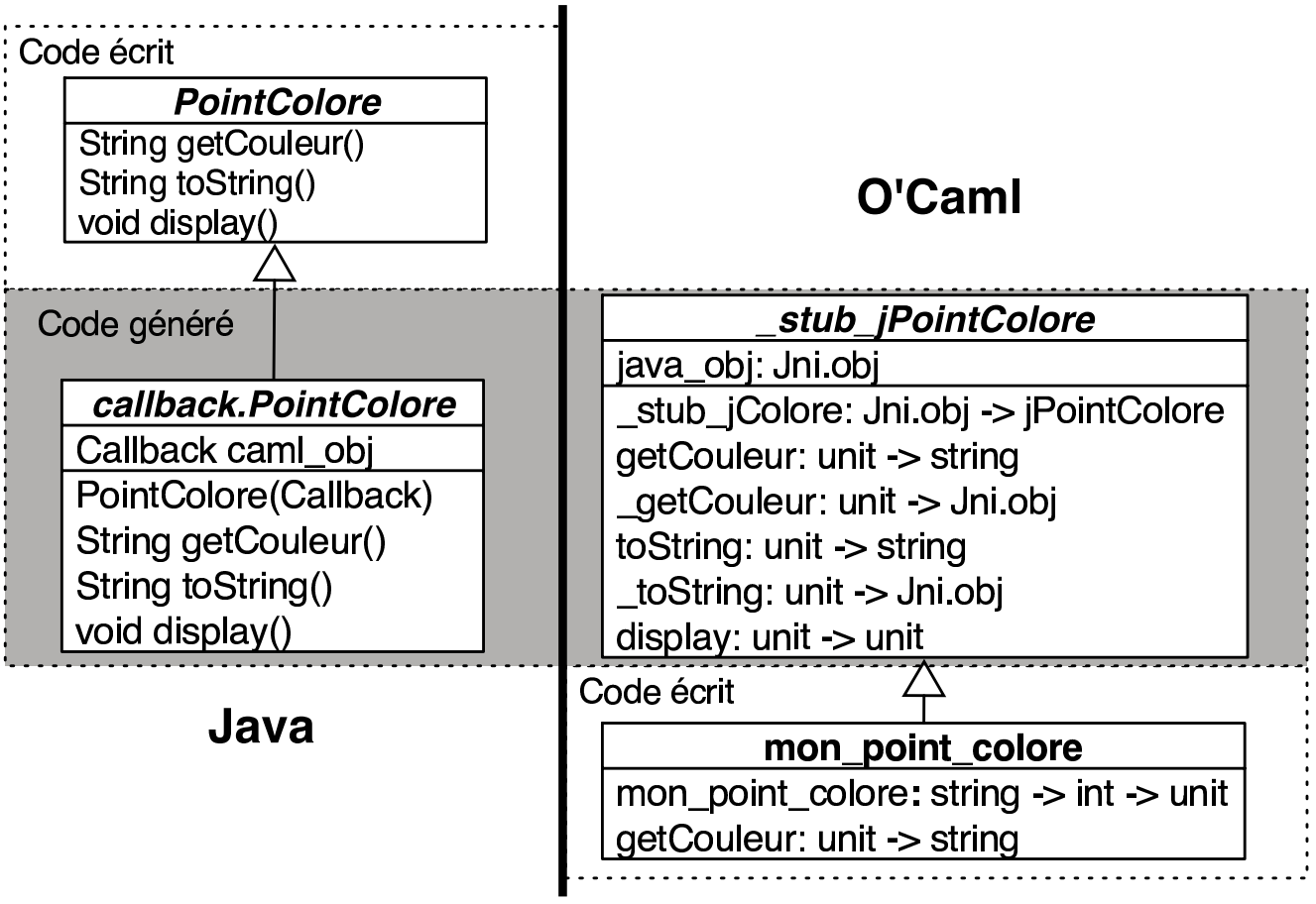
Engendre : `point.ml`



O'Jacare : Attribut *callback*

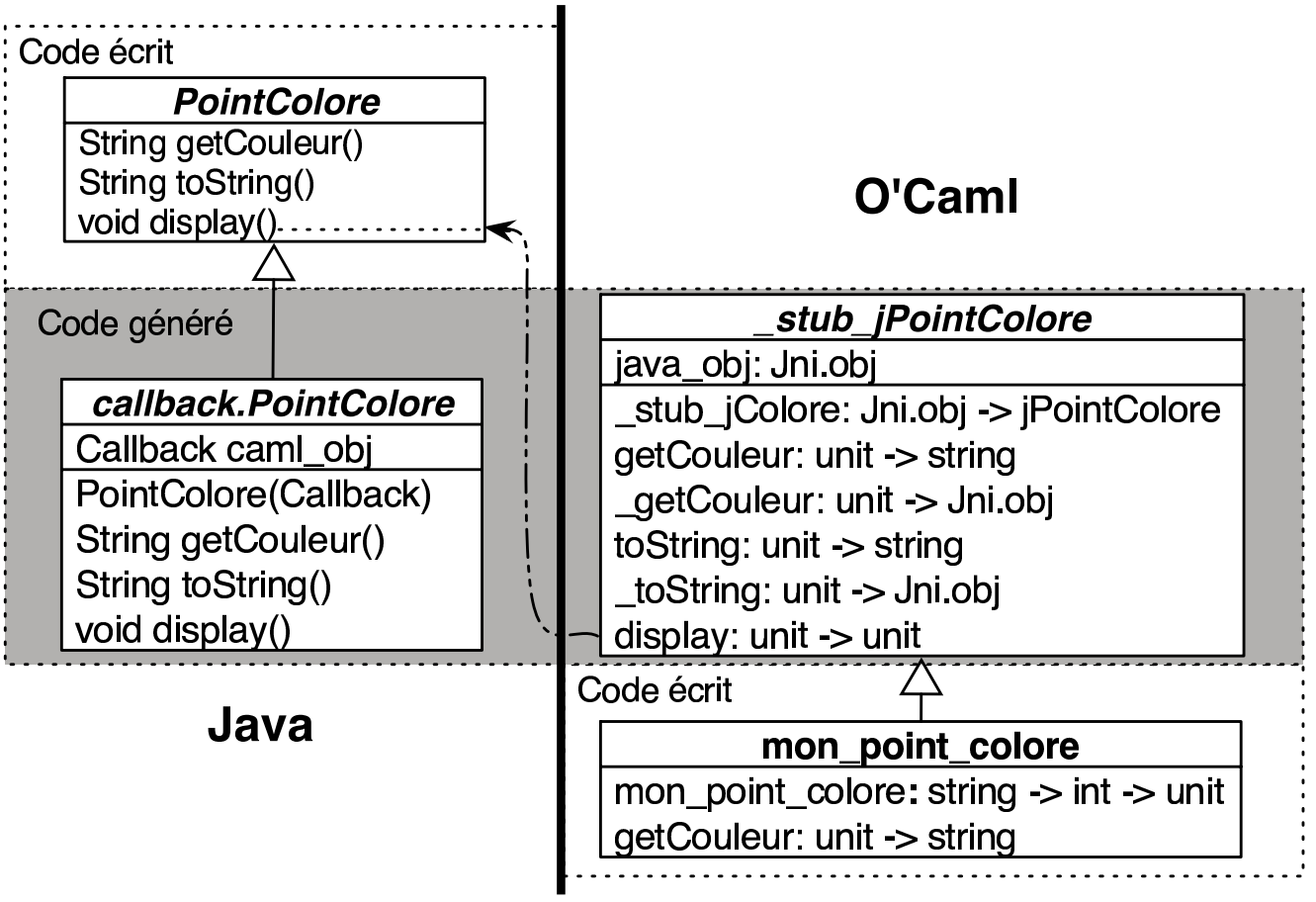


O'Jacare : Attribut *callback*



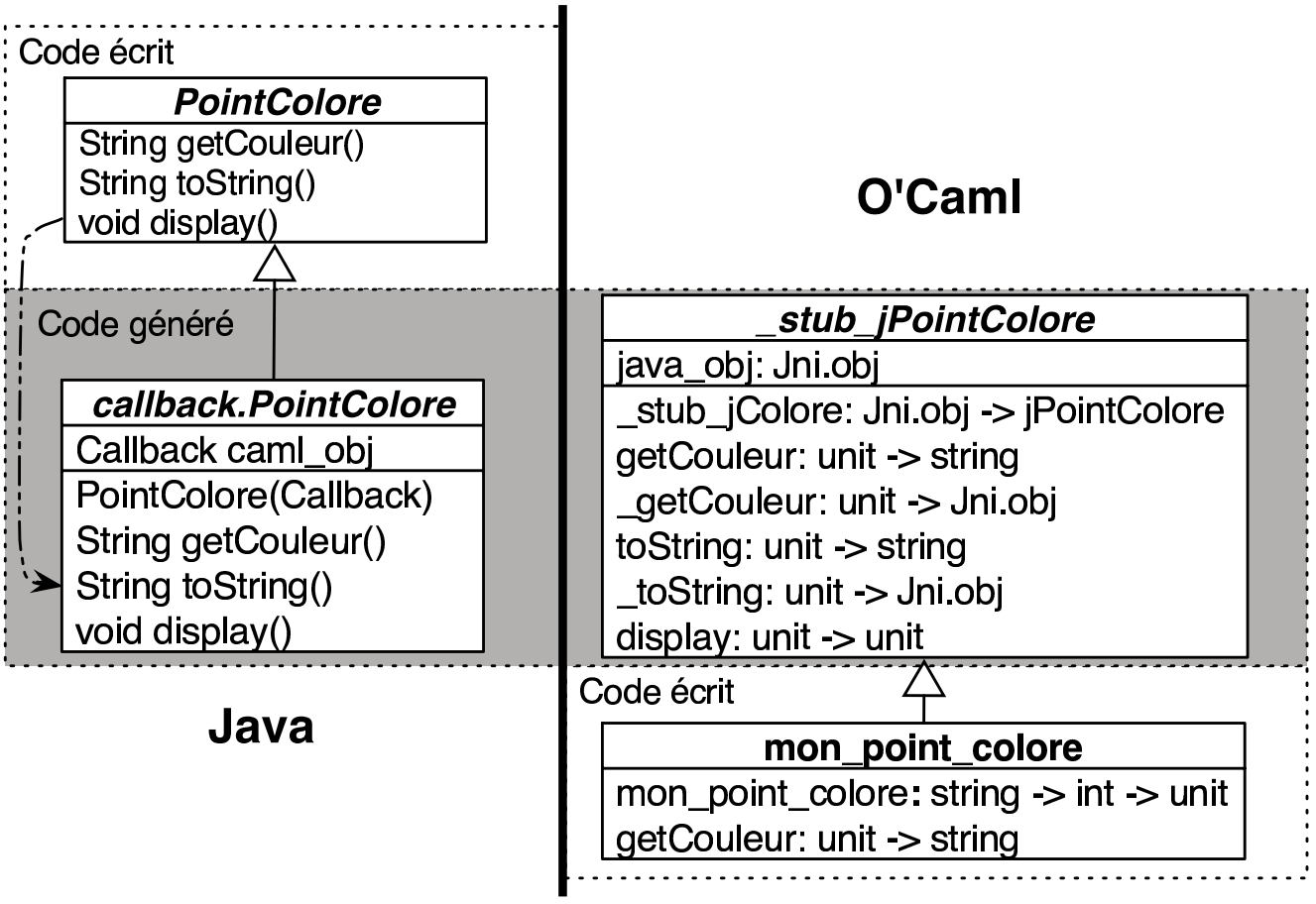

```
let mpc = new
 mon_point_colore
 "bleu" 1
```

O'Jacare : Attribut *callback*



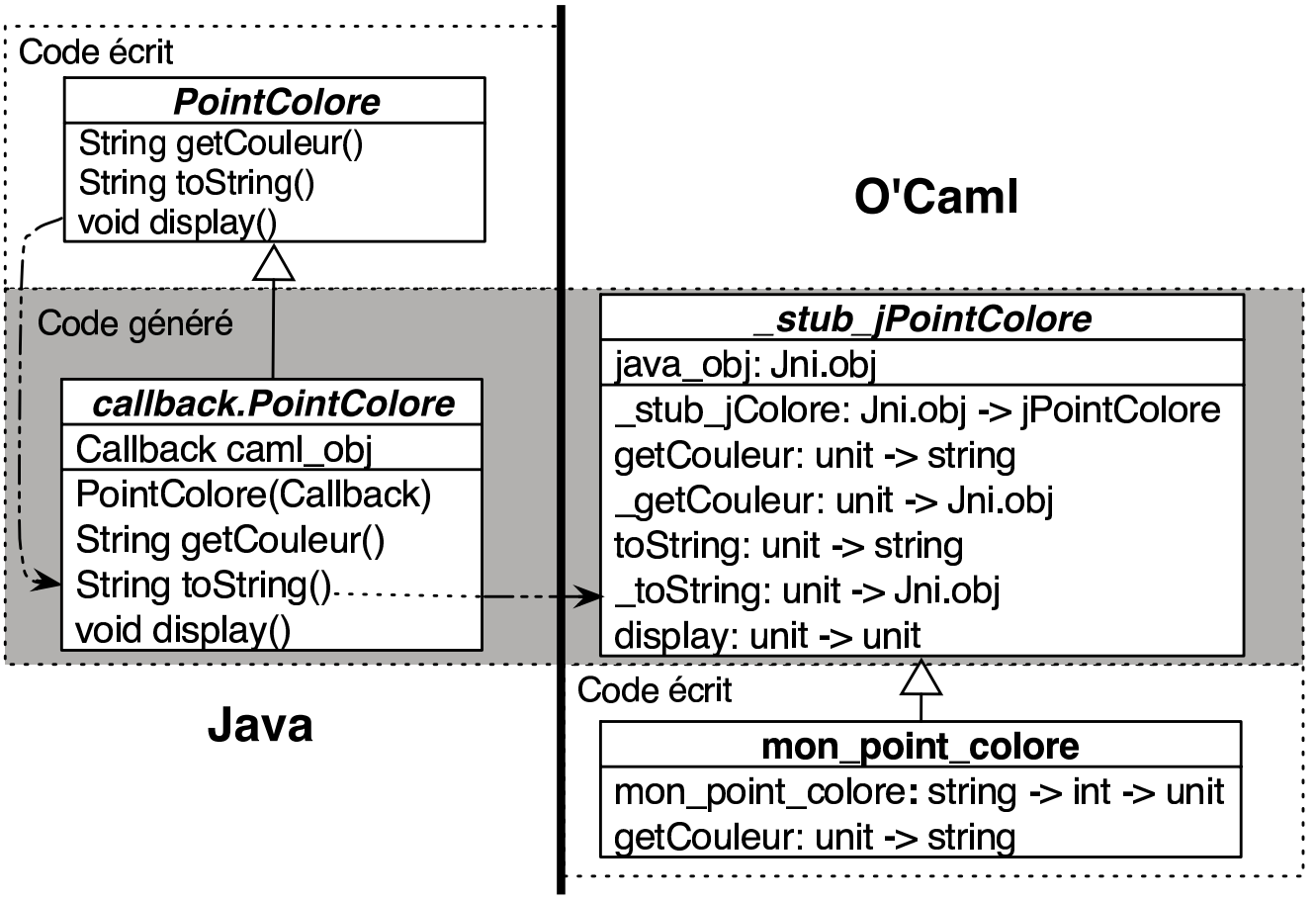

```
mpc#display()
```

O'Jacare : Attribut *callback*



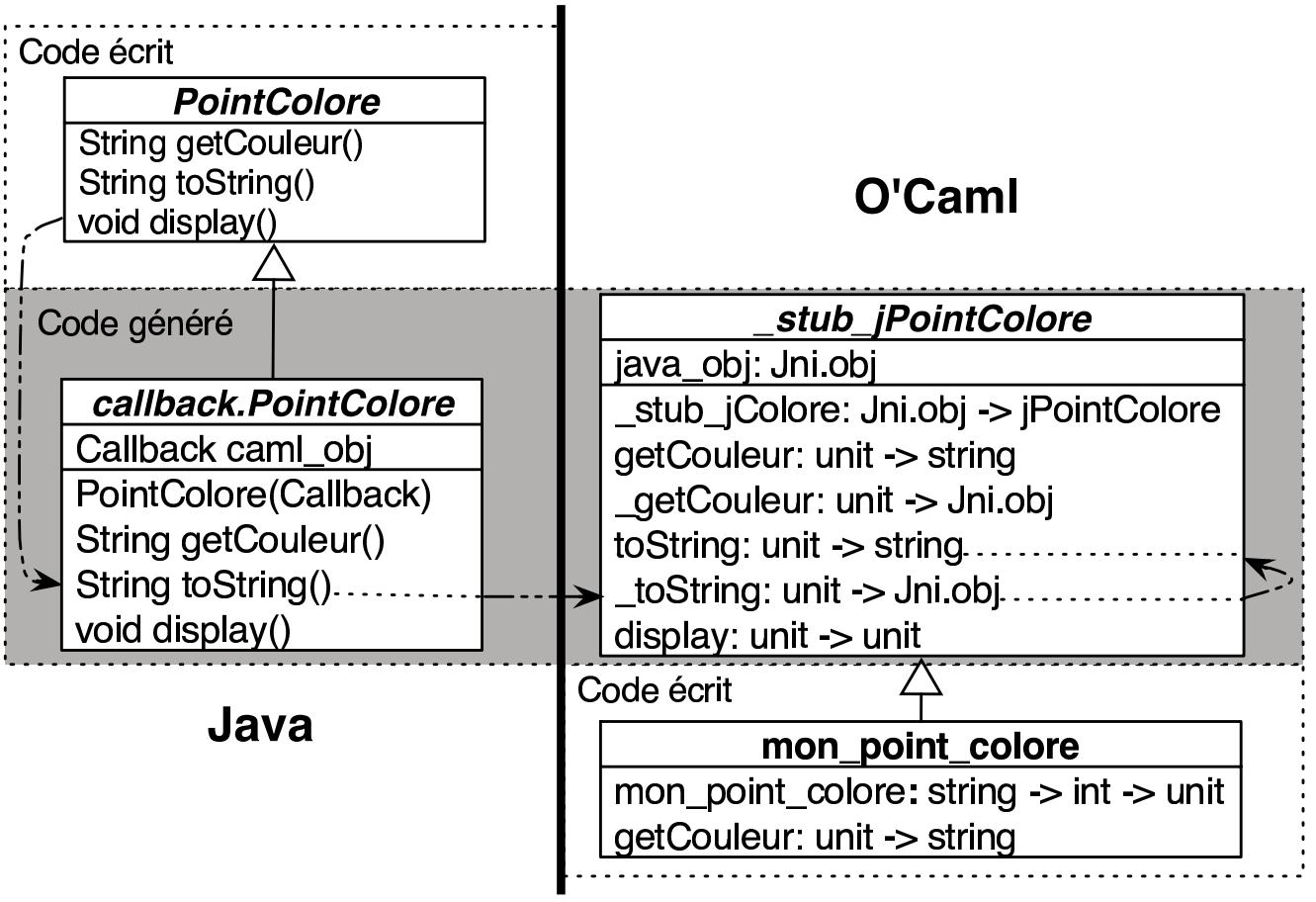

```
mpc#display()
this.toString()
```

O'Jacare : Attribut *callback*



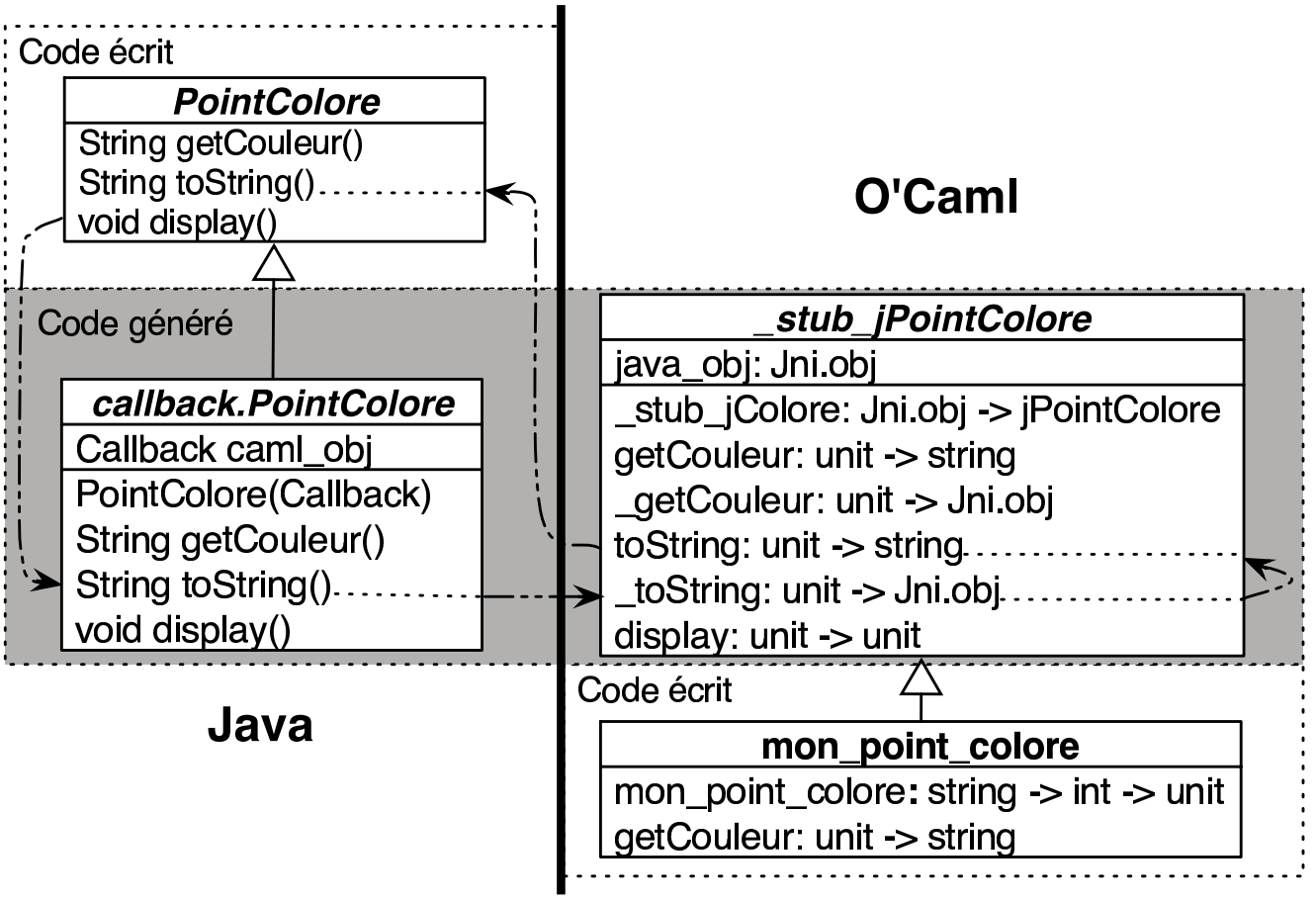

```
mpc#display()
this.toString()
```

O'Jacare : Attribut *callback*



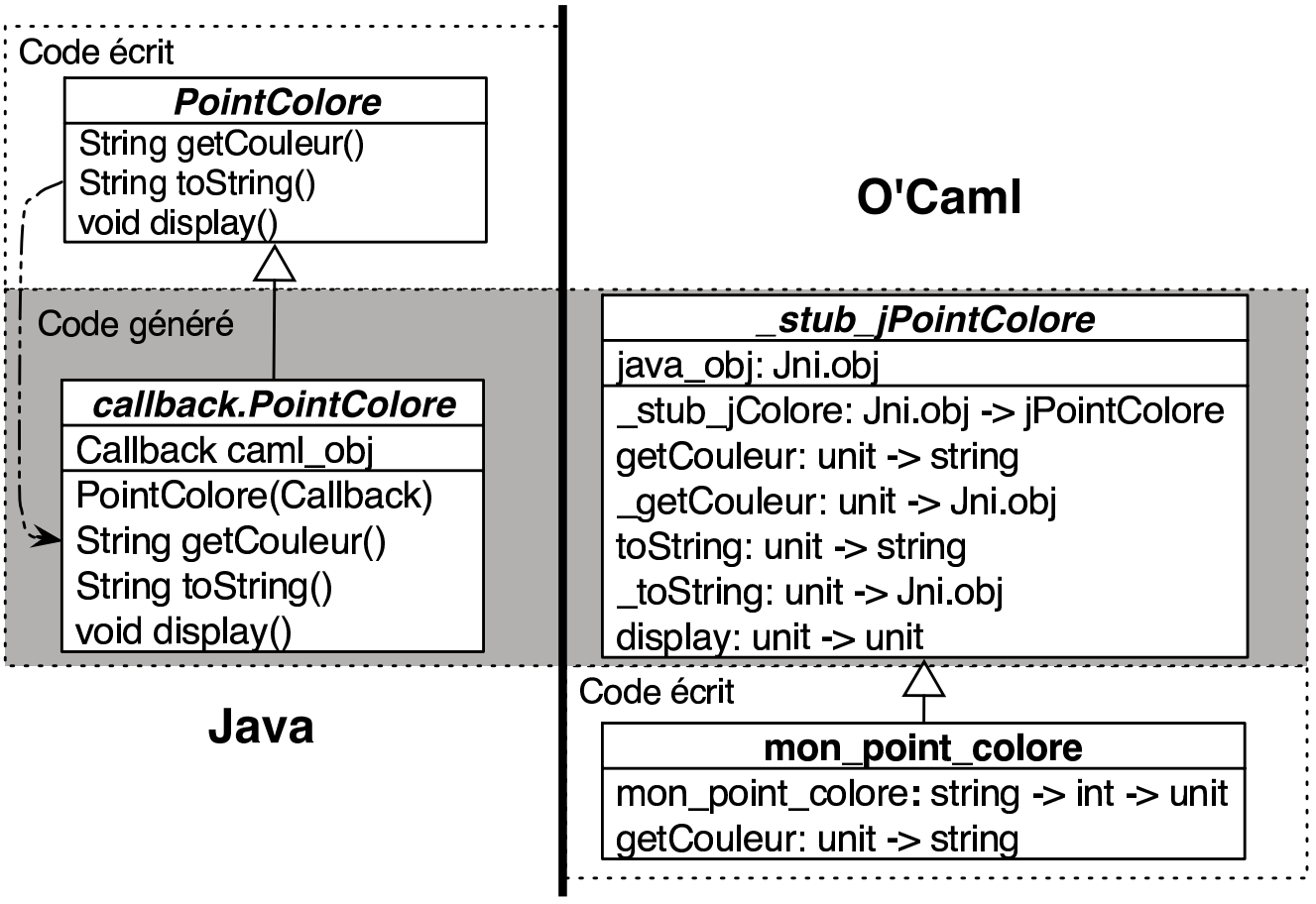

```
mpc#display()
this.toString()
```

O'Jacare : Attribut *callback*



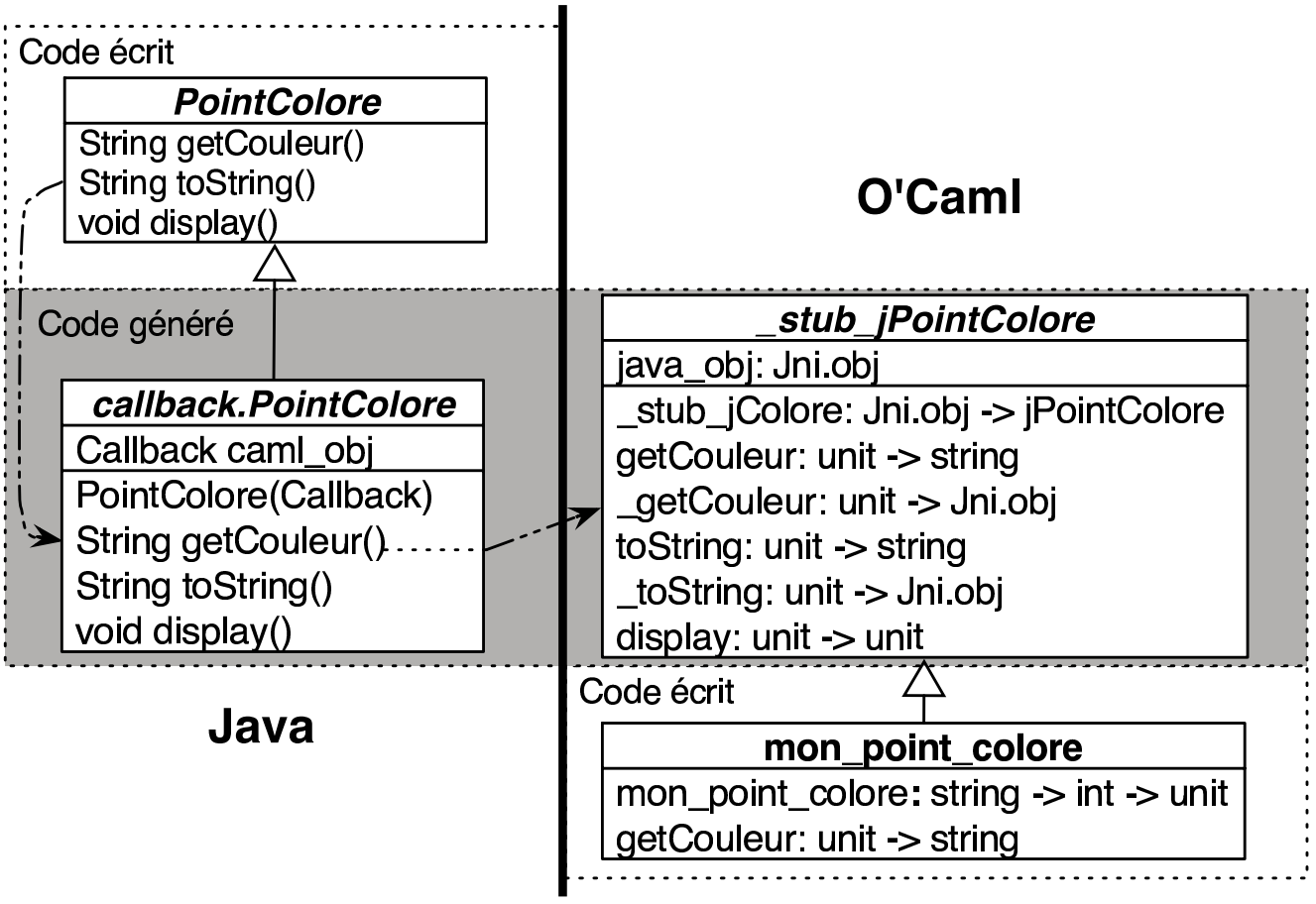

```
mpc#display()
this.toString()
```

O'Jacare : Attribut *callback*



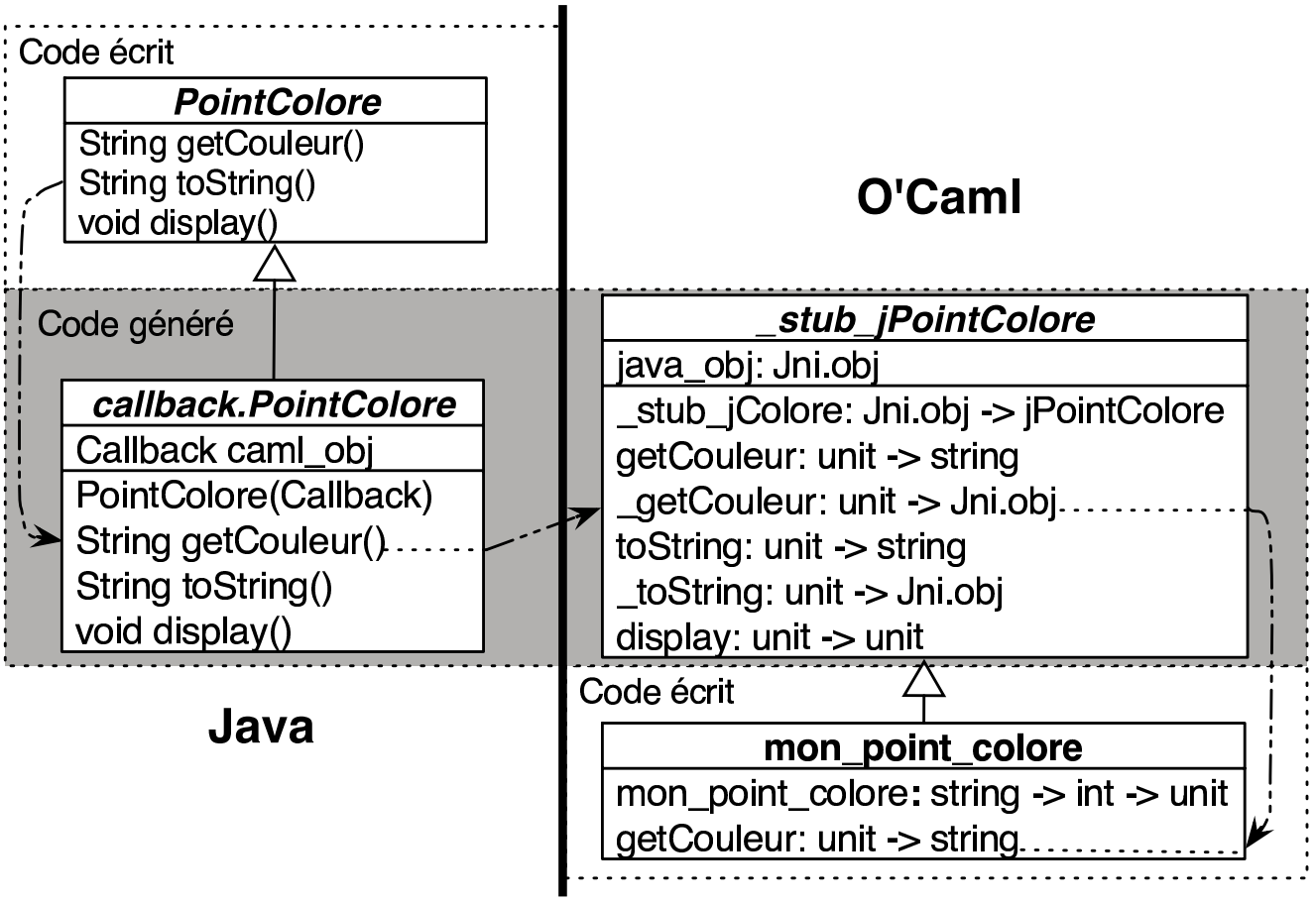

```
mpc#display()
this.toString()
this.getCouleur()
```

O'Jacare : Attribut *callback*



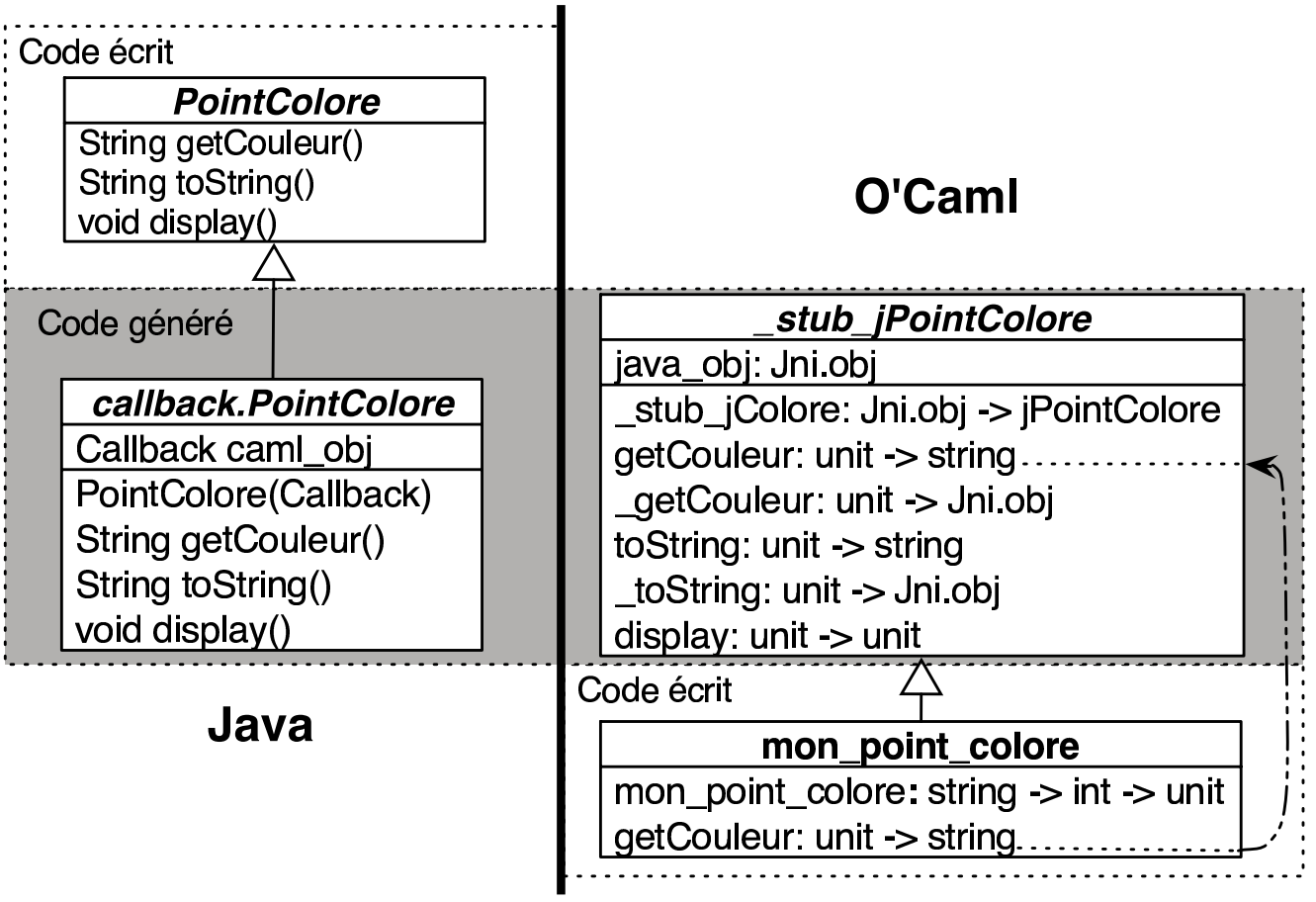

```
mpc#display()
this.toString()
this.getCouleur()
```

O'Jacare : Attribut *callback*



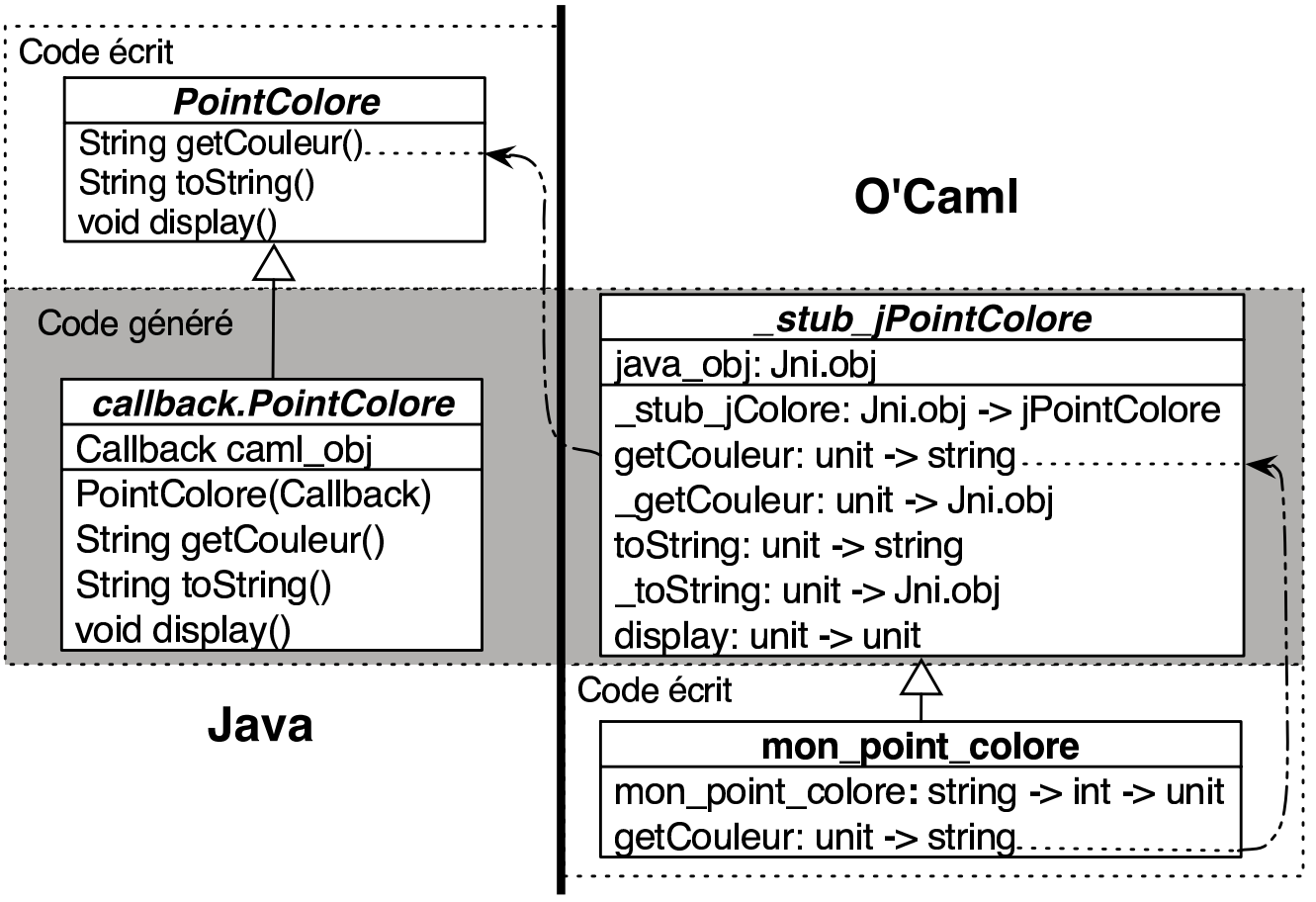

```
mpc#display()
this.toString()
this.getCouleur()
```

O'Jacare : Attribut *callback*



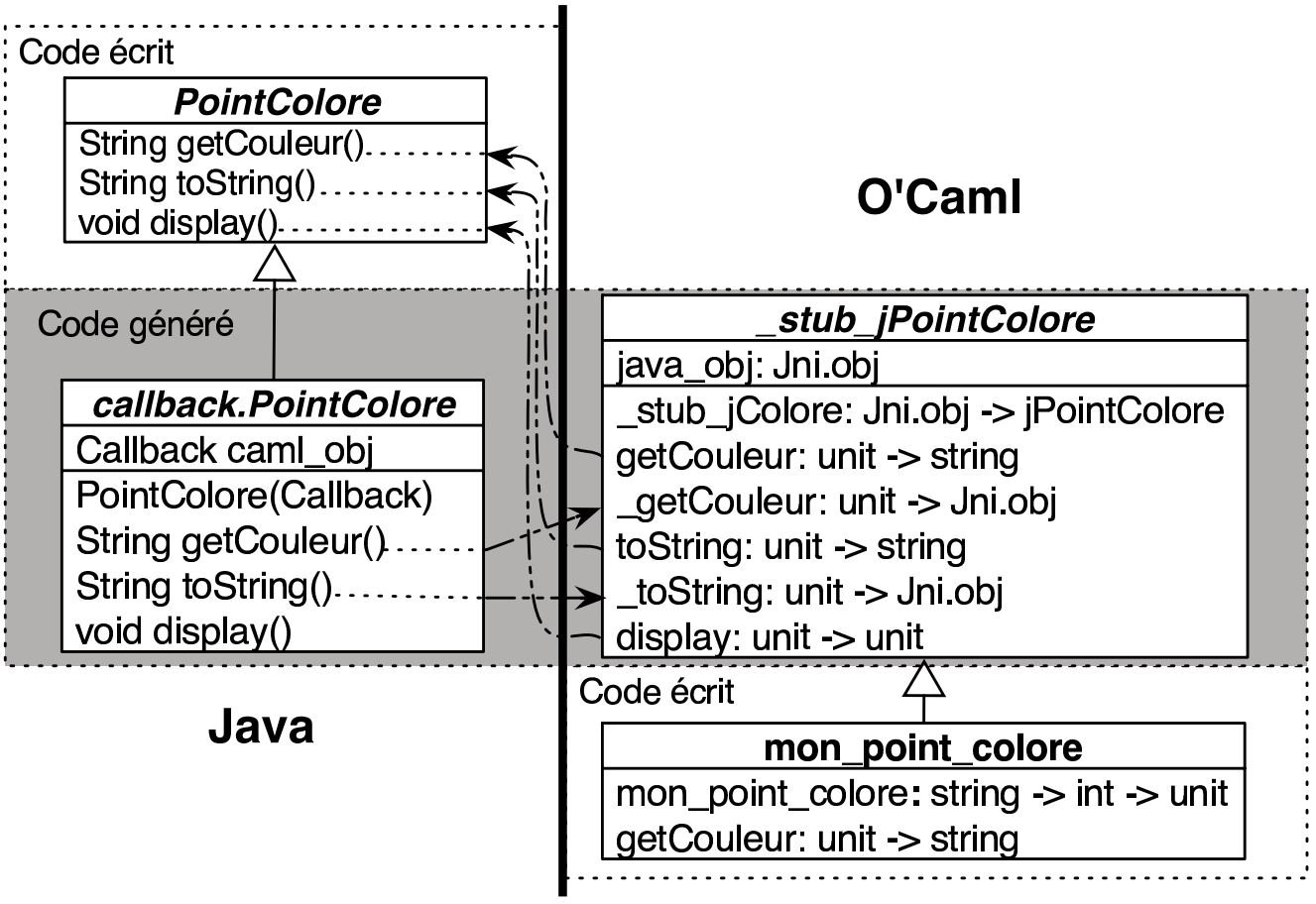

```
mpc#display()
this.toString()
this.getCouleur()
super#getCouleur()
```

O'Jacare : Attribut *callback*




```
mpc#display()
this.toString()
this.getCouleur()
super#getCouleur()
```

O'Jacare : Attribut *callback*




```
mpc#display()
this.toString()
this.getCouleur()
super#getCouleur()
```

Exemple - Visiteur en O'Caml 1/2

Modèle de formules logiques en Java : visiteurML.idl

```
abstract class Formule {
  abstract void accepte(Visiteur v);
}

class Constante extends Formule {
  [name constante] <init>(boolean cst);
  boolean valeur();
  void accepte(Visiteur v);
}
// Variable, OpBin ...

interface Visiteur {
  [name visite_cst] void visite(Constante c); // Variable, OpBin, ...
}

[callback] interface VisiteurML extends Visiteur {
  string get_res();
}
```

Exemple - Visiteur en O'Caml 2/2

Modèle de formules logiques en Java : visiteur.ml

```
class visiteur_ml =
  object (self)
    inherit _stub_jVisiteurML ()
    val buf = Buffer.create 80

    method visite_cst cst =
      Buffer.add_string buf (if cst#valeur () then "true" else "false")

    method visite_non non =
      let sf = non#sous_formule () in
      Buffer.add_string buf "!(";
      sf#accepte (self :> jVisiteur);
      Buffer.add_string buf ")"

    (* ... *)
    method get_res () = (* ... *)

  end
```

Exemple - Visiteur en Java 1/3

Modèle de formules logiques en O'Caml : visiteurJava.idl

```
[callback] interface SyntAbs {
  void accepte(Visiteur v);
}

[callback] interface MainML {
    SyntAbs creerConstante(boolean valeur);
    SyntAbs creerNon(SyntAbs sous_formule); // Variable, OpBin, ...
}

interface Visiteur {
    void visite_cst(boolean valeur);
    void visite_non(SyntAbs sous_formule); // Variable, OpBin, ...
}

interface MainJava {
    static void main(String [] argv, MainML mainMl);
}
```

Exemple - Visiteur en Java 2/3

Modèle de formules logiques en O'Caml : MainJava.java

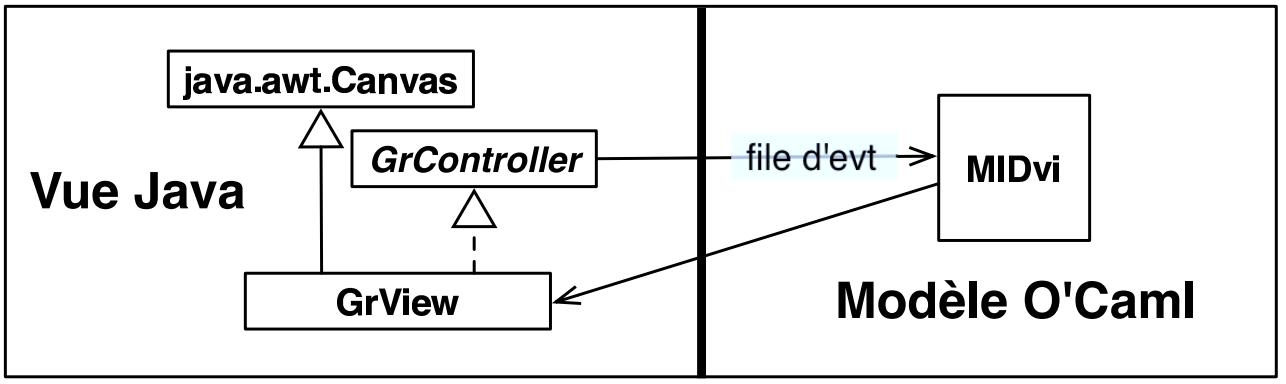
```
public class VisiteurToString {
   public String res;
   public VisiteurToString() { res = ""; }
   void visite_cst(boolean b) { res += b; }
   void visite_non(SyntAbs sf) {
     res += "!("; sf.accepte(this); res += ")";
   }
   // ...
}

public class MainJava {
  public static void main(String[] argv, MainML mainML) {
    SyntAbs f = mainML.creerNon(mainML.creerConstante(true));
    VisiteurToString v = new VisiteurToString();
    f.accepte(v);
  }
}
```

Exemple - Visiteur en Java 3/3

Modèle de formules logiques en O'Caml : syntAbs.ml

```
type syntAbs =
  | Cst of bool | Var of string
  | Non of syntAbs
  | Et of syntAbs * syntAbs | Ou of syntAbs * syntAbs

class syntAbs t =
  object (self)
    inherit VisiteurJava._stub_jSyntAbs ()
    val synt_abs = t
    method accepte v =
      match synt_abs with
      | Cst b -> v#visite_cst b
      | Var nom -> v#visite_var nom
      | Non sf -> v#visite_non (new formule sf :> jFormule)
     (* ... *)
  end
```

Exemple - MVC : visionneuse DVI



`jDvi` tout en Java - AWT

`mlDvi` tout en O'Caml - X11 (Graphics)

`ojDvi` mixte : O'Caml - AWT

	1p	8p	50p	100p	200p	300p
`jDvi`	3,4	6,0	11,7	16,0	26,5	35,5
`ojDvi`	2,0	3,0	6,9	10,0	18,0	25,0
`mlDvi`	0,3	0,6	1,1	1,6	2,4	3,0

JDK de Sun sous Linux

Discussion

simplicité d'emploi

extensions des modèles objets :

- héritage multiple
- *downcast*

limitations de l'implantation :

- Threading : communication limitée au thread principal
- GC : les objets `callback` ne sont jamais désalloués

Travaux futurs

amélioration de l'implantation : objets circulaires, threading

applications :

- interfaces graphiques Java, ...
- parsers O'Caml

plateforme d'exécution commune : `.net`

- O'Caml.NET : projet ocamil (PPS)
- O'Jacaré pour C#