Introduction à Ada pour programmeur Java

Le but de ce texte n'est pas de convertir au langage Ada les programmeurs Java. Bien d'autres notamment sur Internet apporteront l'argumentaire pour cela bien mieux que nous pourrions le faire. Les questions sont plutôt tournées vers les programmeurs en Java qui s'essaye à Ada, comme:

- où trouver un compilateur ?

Nous prendrons un compilateur en utilisation libre JVM-GNAT générant du code compatible avec JVM la machine virtuelle Java.

- quelles sont les structures de programmation ?

Nous montrerons qu'Ada comporte des structures de programmation au moins équivalentes et plus sûre d'emploi même dans le domaine de la programmation objet. Cela donne ainsi lieu au traducteur j2ada qui automatise la transformation du source Java en Ada (voir sur Blady).

- quelles sont les bibliothèques ?

Pour une part Ada vient avec une bibliothèque standard minimale qui assurément ne fait pas le poids avec la gargantuesque bibliothèque de Java. Cela étant le compilateur JVM-GNAT va nous permettre de programmer en Ada les appels à la bibliothèque Java.

Rien ne nous empêche alors de profiter de l'énorme potentiel d'Ada même venant de Java, ce que nous allons voir en détail.

(La trame et les exemples de ce texte sont tirés du livre "Introduction à Java" aux éditions O'Reilly)

1) Le compilateur

Nous emploierons le compilateur libre JVM-GNAT disponible sur le site https://libre.adacore.com. Il est téléchargeable pour Windows après un simple enregistrement sur le site. Pour Mac OS X, son installation est décrite sur Blady.

Au lieu de "javac" vous utiliserez "jvm-gnatmake hello.adb" qui va compiler le fichier hello.adb ainsi que l'ensemble de ses dépendances comme le ferai javac avec le fichier hello.java.

L'option "-I" ajoute une bibliothèque pendant la compilation. Nous ajouterons dons la bibliothèque Java transformée en Ada que nous supposerons située sur /usr/local/gnat/lib/jre (voir son installation sur Blady).

```
$ cat > hello.adb << EOF
with Text_IO; use Text_IO;
procedure Hello  is
begin
   Put_Line ("Hello Java World.");
end Hello;
EOF
$ jvm-gnatmake -I/usr/local/gnat/lib/jre hello.adb
jvm-gnatcompile -c -I/usr/local/gnat/lib/jre hello.adb
jvm-gnatbind -I/usr/local/gnat/lib/jre -x hello.ali
jvm-gnatlink hello.ali
$ ls
hello.adb          hello.class        ada_hello.class        hello.ali
```

Le compilateur a produit trois fichiers :

- hello.class : contient le ByteCode correspondant à hello.java
- ada_hello.class : contient le ByteCode d'initialisation d'Ada
- hello.ali : contient les informations de compilatations Ada

L'utilisation se fait tout naturellement en utilisant la commande java :
(initialiser CLASSPATH=/usr/local/gnat/lib/jgnat.jar:.)

```
$ java hello
Hello Java World.
```

2) Les équivalences Ada aux structures Java

a) Le code source

JVM-GNAT utilise par défaut le codage ISO 8859-1 (Latin-1).
L'option "-gnatW8" permet l'utilisation du codage UTF-8.

b) Les commentaires

Les commentaires Ada sont uniquement sur une seule ligne débutant par "--".

```
// commentaire d'une ligne
```
est traduit par :
```
-- commentaire d'une ligne
```

```
/* commentaire
            sur
                  plusieurs lignes */
```
est traduit par :
```
-- commentaire
--            sur
--                  plusieurs lignes
```

c) Les types de bases

Le package Java fournit l'ensemble des types de bases de Java comme des équivalents des types correspondants Ada :

```
   subtype boolean is Standard.Boolean;
   subtype char    is Standard.Wide_Character;
   subtype byte    is Standard.Short_Short_Integer;
   subtype short   is Standard.Short_Integer;
   subtype int     is Standard.Integer;
   subtype long    is Standard.Long_Integer;
   subtype float   is Standard.Float;
   subtype double  is Standard.Long_Float;
```

L'utilisation se fait naturellement en Ada avec la mention explicite du package Java au début du code source Ada :

```
with Java;
procedure Hello  is
  I : Java.Int := 1230; -- décimal
  J : Java.Int := 8#1230#; -- octal équivalent à 01230
  L : Java.Int := 16#1230#; -- hexadécimal équivalent à 0x1230
  A : Java.Char := 'a';
  NL : Java.Char := Java.Char'Val (10); -- équivalent à '\n'
  U : Java.Char := Java.Char'Val(16#263a#); -- équivalent à '\u263a'
begin
   Put_Line ("Hello Java World.");
end Hello;
```

Ada définit bien plus de types de données détaillé (nature, étendue ou structure) sans besoin de définir des classes creuses amenant des structures de données répondant à d'innombrable cas d'emploi avec beaucoup de précision.

d) Les chaînes de caractères

L'utilisation des chaînes de caractères a toujours été particulière à chaque langage. L'objet Java String est traduit par un objet équivalent en Ada. Pour être exact nous verrons plus loin que les objets Java sont en fait des pointeurs donc traduit comme tel en Ada.

```
String S = "une chaîne";
```

est traduit par :

```
S : Java.Lang.String.Ref := +"une chaîne";
-- '+' effectue la conversion et la création de l'objet
```

Le package Java.Lang.String doit être explicitement mentionné au début du code source Ada comme tout package Ada. Cela permet avec une simple recherche sur "with package" de lister l'ensemble des sources utilisant le package en question.
L'insertion du guillemet \" se traduit par un double guillemet "".

e) Les instructions

. Les blocs de code Java mélangent déclaration et instructions. Ada sépare les deux au sein d'un bloc "declare <déclarations> begin <instructions> end;" qui peut être imbriqué. D'autre part Ada inverse l'ordre d'écriture du type et du nom de la variable.

```
{ int taille = 5;
  taille = taille + 3; }
```

est traduit par :

```
declare
taille : java.int := 5;
begin
taille := taille + 3;
end;
```

. Les méthodes sont traduites soit par des procedures lorsqu'elles ne retourne pas de valeur (void) soit par des fonctions sinon.

```
void ajouteDelta (int delta) {
  int taille = 5;
  i = taille + delta; }
```

est traduit par :

```
procedure ajouteDelta (delta_k : java.int) is
taille : java.int := 5;
begin
i := taille + delta_k;
end;
```

```
int retourneTaille (int delta) {
  int taille = 5;
  i = taille + delta;
  return i; }
```

est traduit par :

```
function retourneTaille (delta_k : java.int) return java.int is
taille : java.int := 5;
begin
i := taille + delta_k;
return i;
end;
```

Remarquez que delta est un mot réservé d'Ada. Il est traduit par un identificateur différent, ici nous lui avons ajouté _k.

. L'expression conditionnelle if n'a qu'une seule forme en Ada "if condition then instruction(s) end if;" quelque soit le nombre d'instructions, ce qui évite les embûches lors de l'ajout d'une instruction à un if à une seule instruction Java.

```
if (i < 4)
  i = 0;
```
est traduit par :
```
if i < 4 then
  i := 0;
end if;
```

```
if (i < 4) {
  i = 0;
  j = 0; }
```
est traduit de même par :
```
if i < 4 then
  i := 0;
  j := 0;
end if;
```

La clause else est traduite directement :
```
if (i < 4)
  i = 0;
else
  i = 99;
```
est traduit par :
```
if i < 4 then
  i := 0;
else
  i := 99;
end if;
```

. L'instruction itérative while est traduite par son équivalent Ada :

```
while (i < 10)
  i = i + 1;
```
est traduit par :
```
while i < 10 loop
  i := i + 1;
end loop;
```

. L'instruction itérative do est traduite par son équivalent Ada :

```
do
  i = i + 1;
while (i < 10);
```

est traduit par :

```
loop
  i := i + 1;
  exit when i < 10;
end loop;
```

. L'instruction itérative for est plus complexe car l'instruction for d'Ada a été rendue très simple toujours pour des raisons de sûreté. Notamment le changement de variable de boucle se fait une seule fois dans le for d'Ada contre au moins trois fois dans le for de Java. Ainsi le for de Java se traduit plus directement avec un while Ada :

```
for (int k = 0; k < 10; k++)
  s = s + k;
```

est traduit par :

```
k : java.int := 0;
while k < 10 loop
  k := k + 1;
  l := l + k;
end loop;
```

. L'instruction de sélection multiple switch est traduite par son équivalent case étendu à tous les types discrets. Remarquez que le break de Java (pratiquement toujours présent) est le comportement intrinsèque du case d'Ada.

```
switch (i) {
  case 0 : i *= 30;
           break;
  case 3 : i *= 33;
           break;
  case 5 : i *= 35;
           break;
  default : i *= 0;
           break;
}
```

est traduit par :

```
case i is
  when 0 => i := i *30;
  when 3 => i := i *33;
  when 5 => i := i *35;
  when others => i := i *0;
end case;
```

f) Les expressions

. Java définit une pléthore d'opérateurs, comme en C, avec des équivalents plus ou moins directs en Ada.

Les opérateurs suivants sont traduits par leurs équivalents :
(par ordre de priorité décroissant avec le type concerné)

i++, i--	(numérique) sont traduits par	i:=i+1, i:=i-1
+i, -i	(numérique) sont traduits par	+i, -i
~i	(numérique) est traduit par	not i
!i	(booléen) est traduit par	not i
(type) i	(tous) sont traduits par	type (i)
i*j, i/j, i%j	(numérique) sont traduits par	i*j, i/j, i mod j
i+j, i-j	(numérique) sont traduits par	i+j, i-j
i+j	(String) est traduit par	i & j
i<<j	(numérique) est traduit par	qt_Right(i,j)
i>>j	(numérique) est traduit par	Shift_Left(i,j)
i>>>j	(numérique) est traduit par	Shift_Right_Arithmetic(i,j)
(inclus dans le package Interfaces)		
i<j, i<=j, i>j, i>=j	(arithmétique) sont traduits par	i<j, i<=j, i>j, i>=j
i instance of j	(objet) sont traduits par	i in j
i==j, i!=j	(tous) sont traduits par	i=j, i/=j
i&j	(booléen, numérique) est traduit par	i and j
i^j	(booléen, numérique) est traduit par	i xor j
i|j	(booléen, numérique) est traduit par	i or j
i&&j	(booléen) est traduit par	i and then j
i||j	(booléen) est traduit par	i or else j
i<j?i=1:j=1	(booléen) est traduit par	if i<j then i:=1; else j:=1; endif;
i=j	(tous) est traduit par	i := j
i*=j, i/=j, i%=j, i+=j, i-=j, i<<=j, i>>=j, i>>>=j, i&=j, i^=j, i|=j	(numérique) sont traduit par	i:=i*j, i:=i/j, i:=i mod j, i:=i+j, i:=i-j, i:=Shift_Right(i,j), i:=Shift_Left(i,j), i:=Shift_Right_Arithmetic(i,j), i:=i and j, i:=i xor j, i:=i or j

. L'affectation en Ada est traduite par := différente de l'égalité =. À l'opposé de Java qui affecte avec = et test l'égalité avec ==. Le choix d'Ada, issu du Pascal, est beaucoup plus sûr car oublier le : de l'affectation est repéré plus aisément par le compilateur qu'oublier le = du test d'égalité!
Par contre, les affectations multiples sont à dissociées en Ada :

```
s = k = 4;
```
est traduit par :
```
k := 4; s := 4;
```

. La constante null d'un objet Java sans référence est traduit de façon similaire par null d'un pointeur Ada sans référence.

. Les variables sont utilisées de façon identiques.

. En Ada, l'appel aux méthodes sans paramètres se fait sans () car Ada distingue les fonctions par le mot réservé "function".

```
L = monObjet.nom.substring(5,10).length();
```
est traduit par :
```
L := monObjet.nom.substring(5,10).length;
```

. En Ada, pour la création d'un objet, nous utiliserons toujours le constructeur défini par convention avec une fonction nommée "New_Objet" retournant un pointeur sur l'objet et non le mot réservé "new" :

```
H = new Date().getHours();
```
est traduit par :
```
H := New_Date.getHours;
```

g) Les exceptions

Java comme Ada définit des exceptions mais de manières plus complexe proche du C++. Tout d'abord, les exceptions Java sont des objets instances de classes prédéfinies hiérarchisées ou dérivés de ces classes prédéfinies, alors que pour Ada, il n'y a qu'un type d'exception avec peu d'exceptions prédéfinies.

L'activation d'une exception Java par "throw" se traduit en Ada par "raise".

Le message associé à l'exception est traduit avec "raise ... with ..." :

```
throw new Exception("message");
```
est traduit par :
```
raise Exception with "message";
```

La gestion d'une exception Java est locale dans le code d'une méthode ce qui se traduit par une bloc Ada local "begin ... exception ... end;" :

```
class BlewIt extends Exception {}
...
void blowUp() throws BlewIt {
throw new BlewIt();
}
...
try {
blowUp();
} catch (BlewIt b) {
System.out.println("BlewIt");
} finally {
System.out.println("Toujours affiché");
}
```

est traduit par :

```
BlewIt : exception;
...
procedure blowUp is
begin
raise BlewIt;
end;
...
begin
blowUp;
exception
  when b : BlewIt => -- catch
        System.out.println("BlewIt");
  when others => -- finally
        System.out.println("Toujours affiché");
        raise;
end;
-- finally
System.out.println("Toujours affiché");
```

À chaque "catch" correspond à un "when". "finally" a la particularité d'être toujours exécuté exception ou pas mais si exception elle resurgit, il correspond alors à "when others => ..." avec un "raise;" pour réactiver l'exception et le même code après le "end;" ce qui permet de toujours exécuter l'instruction du "finally".

Ada comme en Java, si l'exception n'est pas traitée elle se propage à l'appelant.

À noter qu'en Ada, les méthodes n'ont pas l'obligation de déclarer les exceptions qu'elles sont susceptibles de propagées (fonctionnalité de documentation intéressante en théorie mais très contraignante en pratique).

h) Les tableaux

La traduction des tableaux Java est délicate. Ils sont considérés des objets Java particuliers ce qui n'est pas vrai pour les tableaux Ada qui sont des types ordinaires. Quant aux index Java, nombres entiers à partir de 0, ils ne posent pas de difficultés en Ada mais attention l'indice de fin est égal à la longueur - 1.

Il faudrait encapusler le tableau Ada dans un objet. La simple déclaration :

```
int [ ] tableauEntiers;
```

est traduite par :

```
  package TabEntiers is
    type TÉlémentsEntiers is array (Natural range <>) of int;
    type Typ (Max : Natural) is new java.Lang.Object.Typ with record
    -- Max est la longeur - 1 !!!
    -- Ada n'autorise pas de calcul avec un discriminant :-(
      Éléments : TÉlémentsEntiers (0 .. Max) := (others => 0);
    end record;
    type Ref is access all Typ'Class;
  end TabEntiers;
  tableauEntiers1 : TabEntiers.Ref;
```

Ce qui est, il est vrai, très lourd pour un simple tableau de nombres entiers ! On peut s'en tenir à une déclaration plus simple mais en conservant le type accès pour tenir compte de l'allocation dynamique des tableaux Java :

```
int [ ] tableauEntiers;
```

est alors traduit par :

```
  type TÉlémentsEntiers is array (Natural range <>) of int;
  type TTabEntiers is access all TÉlémentsEntiers;
  tableauEntiers2 : TTabEntiers;
```

L'utilisation est également plus simple :

```
int [ ] altitude = new int [30];
altitude[0] = 99;
altitude[1] = 72;
// altitude[2] == 0
```

est traduit par :

```
    type TÉlémentsEntiers is array (Natural range <>) of int;
    type TTabEntiers is access all TÉlémentsEntiers;
    subtype TAltitude is TÉlémentsEntiers (0 .. 29); -- indice fin = longueur - 1
    altitude : TTabEntiers := new TAltitude'(others => 0);
  begin
    altitude (0) := 99;
    altitude (1) := 72;
    -- altitude[2] = 0
  end;
```

Un tableau d'objets sont des références de cet objets, ils sont donc initialisés par la valeur null :

```
String noms [ ] = new String [4];
noms[0] = new String();
noms[1] = "Hello Java with Ada";
noms[2] = unObjet.toString();
// noms[3] == null
```

est traduit par :

```
   type TÉlémentsChaines is array (Natural range <>) of java.Lang.String.Ref;
   type TTabChaines is access all TÉlémentsChaines;
   subtype TNoms is TÉlémentsChaines (0 .. 3); -- indice fin = longueur - 1
   noms    : TTabChaines          := new TNoms'(others => null);
   unObjet : java.Lang.Object.Ref := New_Object;
 begin
   noms (0) := java.Lang.String.New_String;
   noms (1) := +"Hello Java with Ada";
   noms (2) := java.Lang.String.Ref (unObjet.ToString);
   -- noms(3) = null
 end;
```

Les initialisations explicites sont également permises :

```
int [ ] premier = {1, 2, 3, 5, 7, 7+4}; // premiers[2] == 3
```

est traduit par :

```
type TÉlémentsEntiers is array (Natural range <>) of int;
type TTabEntiers is access all TÉlémentsEntiers;
premier : TTabEntiers := new TÉlémentsEntiers'(1, 2, 3, 5, 7, 7 + 4); -- premier(2) = 3
```

Le champ donnant la taille d'un tableau Java trouve un équivalent dans l'attribut donnant la taille d'un tableau Ada :

```
String [ ] mousquetaires = {"Aramis", "Atos", "Portos"};
int nombre = mousquetaires.length; // nombre == 3
```

est traduit par :

```
type TÉlémentsChaines is array (Natural range <>) of Java.Lang.String.Ref;
type TTabChaines is access all TÉlémentsChaines;
mousquetaires : TTabChaines := new TÉlémentsChaines'(+"Aramis", +"Atos",
+"Portos");
nombre : int := mousquetaires'length; -- nombre = 3
```

i) Les tableaux anonymes

En Ada le tableau anonyme est plutôt à éviter si l'on ne bénéficie pas d'un ramasse-miètes car il alloue de la mémoire que l'on ne pourra plus libérer explicitement. Malgré tout voici une traduction possible en utilisant un agrégat Ada dont le tableau est pré-déclaré ainsi qu'une fonction d'allocation mémoire du tableau :

```
Chien pokey = new Chien("gris");
Chat squiggles = new Chat("noir");
Chat jasmine = new Chat("orange");
setAnimaux(new Animaux [ ] {pokey, squiggles, jasmine});
```

est traduit par :

```
pokey     : Chien.Ref := New_Chien (+"gris");
squiggles : Chat.Ref  := New_Chat (+"noir");
jasmine   : Chat.Ref  := New_Chat (+"orange");
type TÉlémentsAnimaux is array (Natural range <>) of Animaux.Ref;
type TTabAnimaux is access all TÉlémentsAnimaux;
function New_TTabAnimaux (Éléments : TÉlémentsAnimaux) return TTabAnimaux is
  begin
     return new TÉlémentsAnimaux'(Éléments);
  end New_TTabAnimaux;
begin
    setAnimaux
      (New_TTabAnimaux
         ((Animaux.Ref (pokey), Animaux.Ref (squiggles), Animaux.Ref (jasmine))));
end;
```

j) Les tableaux multi-dimensionnels

Les tableaux multi-dimensionnels Java correspondent en Ada à des tableaux de tableaux et non pas à des tableaux à plusieurs indices qui sont réellement des tableaux à multi-dimensions.

```
pièceÉchecs [ ][ ] jeuÉchecs;
jeuÉchecs = new pièceÉchecs [8][8];
jeuÉchecs[0][0] = new pièceÉchecs("Tour");
jeuÉchecs[1][0] = new pièceÉchecs("Pion");
```

est traduit par :

```
type TÉlémentsPièceÉchecs is array (Natural range <>) of pièceÉchecs.Ref;
type TTabPièceÉchecs is access all TÉlémentsPièceÉchecs;
type TÉlémentsPièceÉchecs_2 is array (Natural range <>) of TTabPièceÉchecs;
type TÉchiquier is access all TÉlémentsPièceÉchecs_2;
subtype TLignes is TÉlémentsPièceÉchecs (0 .. 7); -- indice fin = longueur - 1
```

```
    subtype TJeuÉchecs is TÉlémentsPièceÉchecs_2 (0 .. 7); --indice fin=longueur-1
    jeuÉchecs : TÉchiquier;
  begin
    jeuÉchecs          := new TJeuÉchecs'(others => new TLignes'(others => null));
    jeuÉchecs (0) (0)  := New_PièceÉchecs (+"Tour");
    jeuÉchecs (1) (0)  := New_PièceÉchecs (+"Pion");
  end;
```

Ce qui permet de traduire la création d'un tableau triangulaire :

```
int [ ][ ] triangle = new int [5][ ];
for (int i=0; i < triangle.length; i++) {
  triangle[i] = new int [i+1];
  for (int j=0; j<i+1; j++)
    triangle[i][j] = i + j;
}
```

est traduit par :

```
type TÉlémentsEntiers is array (Natural range <>) of int;
type TTabEntiers is access all TÉlémentsEntiers;
type TÉlémentsEntiers_2 is array (Natural range <>) of TTabEntiers;
type TTriangle is access all TÉlémentsEntiers_2;
subtype TLignes is TÉlémentsEntiers_2 (0 .. 4); -- indice fin = longueur - 1
triangle : TTriangle := new TLignes'(others =>  null);
begin
for i in 0..triangle'length-1 loop -- indice fin = longueur - 1
  triangle(i) := new TÉlémentsEntiers(0..i); -- indice fin = longueur - 1
  for j in 0..i loop -- indice fin = longueur - 1
    triangle(i)(j) := i+j;
  end loop;
end loop;
end;
```

3) Les objets Java

a) Les classes

Une classe Java n'a pas d'équivalent direct en Ada. Elle s'apparente à la fois à un type objet étiquetté hérité de l'objet parent de toute les classes Java (Java.Lang.Object) qui contient les champs de l'objet (variables d'instance) et un paquetage qui contient le type objet et ses méthodes.

```
class Pendule {
	float masse;
	float longueur = 1.0;
	int cycles;
	float position (float date) {
		...
	}
}
```

est traduit par :

```
package Pendule is
	type Typ is new Java.Lang.Object.Typ with record
		masse : float;
		longueur : float := 1.0;
		cycles : int;
		end record;
	function position (this : access Typ; date : float) return float;
end;
package body Pendule is
	function position (this : access Typ; date : float) return float is
		begin
		...
		end;
end;
```

Si un champ est privé alors tous le sont en Ada :

```
package Pendule is
	type Typ is new Java.Lang.Object.Typ with private;
	function position (this : access Typ; date : float);
private
	type Typ is new Java.Lang.Object.Typ with record
		masse : float;
		longueur : float := 1.0;
		cycles : int;
		end record;
end;
```

Un champ statique (variable de classe) est déclaré comme variable du paquetage hors du type objet :

```
class Pendule {
        ...
        static float gravitation = 9.80;
        ...
}
```

est traduit par :

```
package Pendule is
        ...
        gravitation : float := 9.80;
        ...
end;
```

De même, pour une variable de classe constante :

```
class Pendule {
        ...
        static final float GRAVITATION = 9.80;
        ...
}
```

est traduit par :

```
package Pendule is
        ...
        GRAVITATION : constant float := 9.80;
        ...
end;
```

b) Les méthodes

Les méthodes se trouvent naturellement dans le paquetage de la classe en tant que primitives du type objet et donc elles comportent en Ada une référence à l'objet en paramètre. Le code des primitives se trouve dans le corps du paquetage. Les primitives sont soit des fonctions retournant un type soit des fonctions avec void, les premières se traduise par des fonctions Ada, les dernières par des procédures Ada. Par contre, en Ada, un appel à une fonction retournant un type doit obligatoirement utiliser cette valeur contrairement à Java.

```
class Oiseaux {
	int xPos, yPos;
	double voler (int x, int y) {
		double distance = Math.sqrt(x*x + y*y);
		battreDesAiles (distance);
		xPos = x;
		yPos = y;
		return distance;
	}
}
```

est traduit par :

```
package Oiseaux is
	type Typ is new Java.Lang.Object.Typ with record
		xPos, yPos : int;
		end record;
	function voler (this : access Typ; x, y : int) return double;
end;
package body Oiseaux is
	function voler (this : access Typ; x, y : int) return double is
		distance : double := Java.Lang.Math.sqrt (double(x*x + y*y));
		begin
			battreDesAiles (distance);
			this.xPos := x;
			this.yPos := y;
			return distance;
		end;
end;
```

Comme la référence à l'objet fait toujours partie des paramètres, le paramètre this est ainsi déclaré explicitement. Par contre, une méthode statique fera partie du paquetage sans faire référence à l'objet.

```
class Oiseaux {
	...
	static String [ ] recupérerTypesOiseaux () {
		String [ ] types;
		// Créer une liste des types
	return types;
	}
	...
}
```

est traduit par :

```
package Oiseaux is
        ...
        function recupérerTypesOiseaux  return Java.Lang.String.Arr;
        ...
end;
package body Oiseaux is
        function recupérerTypesOiseaux  return Java.Lang.String.Arr is
                types : Java.Lang.String.Arr;
                begin
                        -- Créer une liste des types
                        return types;
                end;
end;
```

c) Les constructeurs

Ada, comme Java, permet de créer du objet dynamiquement avec le mot-clé new, par contre, il n'y a pas d'appel à un constructeur par défaut ou explicite en Ada standard. JVM-GNAT va nous aider en permettant de définir des fonctions constructeurs effectuant les deux. Le constructeur est alors une méthode particulière de l'objet dont le nom par convention est la concaténation de New_ et du nom de la classe, dont le dernier paramètre est une référence à l'objet avec null comme valeur par défaut (ainsi nous pouvons toujours appeler un constructeur avec la référence d'une instance d'objet dans ce paramètre), dont les premiers paramètres sont libres, dont la première instruction est l'appel explicite au constructeur parent (implicite en Java), enfin qui retourne l'allocation mémoire de l'instance. En effet, lors de l'appel du constructeur si la référence de l'objet est null, JVM-GNAT alloue de la mémoire pointée par cette référence qui devient donc différente de null. Ce mécanisme est déclenché par la particularisation du constructeur avec le pragma Java_Constructor. Le constructeur peut être surchargé suivant la présence des paramètres. Il doit en exister au moins un. Il sera appelé explicitement à la création d'une nouvelle instance de classe.

```
class Date {
  long heure;
  Date() {
    heure = heureCourante();
  }
  Date(String date) {
    heure = convertitDate(date);
  }
}
```

est traduit par :

```
package Date is
   type Typ is new java.Lang.Object.Typ with record
      heure : long;
   end record;
   type Ref is access all Typ'Class;
   function New_Date (This : Ref := null) return Ref;
   function New_Date (date : java.Lang.String.Ref; This : Ref := null) return Ref;
   pragma Java_Constructor (New_Date);
end Date;
package body Date is
   function New_Date (This : Ref := null) return Ref is
      Super : java.Lang.Object.Ref := java.Lang.Object.New_Object
(java.Lang.Object.Ref (This)); -- appel au constructeur parent
   begin
      This.heure := heureCourante;
      return This;
   end New_Date;
   function New_Date (date : java.Lang.String.Ref; This : Ref := null) return Ref is
      Super : java.Lang.Object.Ref := java.Lang.Object.New_Object
(java.Lang.Object.Ref (This)); -- appel au constructeur parent
   begin
      This.heure := convertitDate (date);
      return This;
   end New_Date;
end Date;
```

La première des deux fonctions définit le constructeur par défaut qui doit être appelé explicitement en Ada.

```
Date maintenant = new Date();
Date noël = new Date("25 décembre 2010");
```

est traduit par :

```
   maintenant : Date.Ref := Date.New_Date;
   noël       : Date.Ref := Date.New_Date (+"25 décembre 2010");
```

Un constructeur peut aussi faire référence à un autre constructeur de la même classe. L'utilisation en Java de la forme spéciale this() est traduite en Ada directement par le nom du constructeur déterminé par le méchanisme de surcharge Ada. Dans ce cas, l'appel au constructeur parent doit se faire au travers du constructeur appelé :

```
class Voiture {
  String modele;
  int portes;
  Voiture(String m, int d) {
    modele = m;
    portes = d;
  }
  Voiture(String m) {
    this(m, 4);
  }
}
```

est traduit par :

```
      package Voiture is
        type Typ is new java.Lang.Object.Typ with record
          modele : java.Lang.String.Ref;
          portes : int;
        end record;
        type Ref is access all Typ'Class;
        function New_Voiture
          (m    : java.Lang.String.Ref;
           d    : int;
           This : Ref := null)
           return Ref;
        function New_Voiture (m : java.Lang.String.Ref; This : Ref := null) return Ref;
        pragma Java_Constructor (New_Voiture);
      end Voiture;
      package body Voiture is
        function New_Voiture
          (m    : java.Lang.String.Ref;
           d    : int;
           This : Ref := null)
           return Ref
        is
          Super : java.Lang.Object.Ref :=
java.Lang.Object.New_Object(Java.Lang.Object.Ref(This)); -- appel au constructeur
parent
        begin
          This.modele := m;
          This.portes := d;
          return This;
```

```
    end New_Voiture;
    function New_Voiture (m : java.Lang.String.Ref; This : Ref := null) return Ref is
      Temp : Ref := New_Voiture (m, 4, This);
    begin
      return This;
    end New_Voiture;
  end Voiture;
```

Le mois prochain les blocs de code.
Pascal Pignard, octobre-décembre 2009, février-juin 2010, février 2013.