Le langage Ada – 5
Les enregistrements

Yves Bailly

EN DEUX MOTS **Les tableaux du mois dernier ne permettent de stocker que des informations d'un même type. Comme beaucoup d'autres langages, Ada offre un moyen de placer en une seule entité des données de types différents. Là où le langage C parle de structures ou Python de classes, Ada a repris la terminologie du Pascal et parle plutôt d'enregistrements.**

L'enregistrement est la structure de donnée fondamentale en Ada pour représenter les idées du programmeur. Doté d'une grande souplesse, on imagine mal comment un programme non trivial pourrait s'en passer, à moins de tolérer une effroyable complexité dans l'écriture du code. Pour l'essentiel, ce que nous allons voir ici n'est pas propre à Ada. On retrouve nombre des aspects dans nombre de langages – mais quelques possibilités sont uniques à Ada.

Déclaration

Supposons que nous souhaitions réaliser un programme pour jouer aux Tours de Hanoï (toute ressemblance avec d'autres articles n'aurait rien de fortuit). Il nous faut une structure pour représenter une tour, qui contiendra essentiellement un tableau de disques et le nombre de disques actuellement empilés sur la tour. Ce qui pourrait ressembler à ceci :

```
1 type Disque is new Integer range 0..10 ;
2 type Étage is new Integer range 0..10 ;
3 type Pile_Disques is array (Étage) of Disque ;
4
5 type Tour is
6 record
7    dernier_étage: Étage := 0 ;
8    pile: Pile_Disques := (others=>0) ;
9 end record ;
```

Les trois premières lignes définissent simplement quelques types qui nous seront utiles par la suite, la pile de disque étant représentée par un simple tableau (ligne 3). Le véritable objet de notre attention aujourd'hui se situe entre les lignes 5 et 9. Nous définissons ici un type nommé `Tour`, qui est un enregistrement (`record`) contenant deux champs nommés `dernier_étage` (de type `Étage`) et `pile` (de type `PileDisques`).

Petite remarque en passant : vous aurez peut-être remarqué les accents présents dans ce code source, ce qui peut paraître incongru. Je me suis permis cette fantaisie pour montrer une possibilité du compilateur GNAT, qui est justement d'accepter ce genre de caractères dans les identifiant.

En fait, cela fera bientôt partie de la norme Ada officielle : les compilateurs de la prochaine version du langage devraient être capables de traiter un code source contenant des noms de variables composés de caractères grecs et turcs (ce qui ne manquerait pas de piquant), ou autres, naturellement. Il est pour l'instant nécessaire de donner l'option `-gnatif` au compilateur, par exemple :

```
$ gnatmake -gnatif hanoi_decl.adb
```

Mais revenons à notre structure. Cela ressemble fort à une déclaration `struct` du langage C, avec toutefois cette différence notable qu'il est possible de donner une valeur par défaut pour les champs, ce qui est vivement recommandé.

On reproduit ainsi un comportement comparable à celui du constructeur d'une classe C++ ou Java, quoiqu'en plus limité.

Naturellement, rien ne nous empêche d'utiliser ce type pour composer d'autres types, par exemple :

```
1 type Num_Tour is new Integer range 1..3 ;
   2 type Rangée_Tours is array (Num_Tour) of Tour ;
   3 type Num_Mouvements is new Integer range 0..Integer'Last ;
   4 type Jeu_Hanoï is
   5 record
   6    tours: Rangée_Tours ;
   7    nb_mouvements: Num_Mouvements := 0 ;
   8 end record ;
```

La ligne 2 déclare un type tableau `Rangée_Tours` dont les éléments sont de type `Tour`, ce tableau étant lui-même utilisé dans un enregistrement `Jeu_Hanoï` contenant différents paramètres du jeu. Remarquez que nous n'avons pas donné de valeur initiale pour le premier champ.

Non pas que cela ne serait pas possible (nous allons bientôt voir comment), simplement ce n'est pas strictement nécessaire car les éléments du tableau sont des enregistrements eux-mêmes déjà initialisés. On pourrait objecter à cela que le jeu risque de se retrouver ainsi dans un état incohérent, sans aucun disque sur aucune tour...

Utilisation

La déclaration d'une variable d'un type enregistrement se fait comme n'importe quelle autre :

```
jeu: Jeu_Hanoï;
```

De même que pour les tableaux, on peut donner un agrégat pour initialiser les composants de l'enregistrement, sauf qu'au lieu de donner l'indice de l'élément on donne son nom.

Dans notre cas, nous avons un enregistrement qui contient un tableau d'enregistrements contenant un tableau. Une initialisation complète pourrait ressembler à ceci :

```
jeu: Jeu_Hanoï := (
   tours => (
      1 => (
         dernier_étage => 10,
         pile => (0, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)),
      others => (
         dernier_étage => 0,
         pile => (others => 0))),
   nb_mouvements => 0) ;
```

L'accès aux éléments d'un enregistrement se fait en utilisant la notation pointée usuelle. Voici par exemple une procédure qui affiche le contenu d'une instance de **Jeu_Hanoï** :

```
 1 procedure Afficher(jeu: in Jeu_Hanoï) is
 2 begin
 3  Loop_Tours:
 4  for ind_tour in Num_Tour
 5  loop
 6    Put("Tour " & Num_Tour'Image(ind_tour) & " : ") ;
 7    Loop_Etages:
 8    for ind_etage in 1..Étage'Last
 9    loop
10      Put(Disque'Image(jeu.tours(ind_tour).pile(ind_etage))) ;
11    end loop Loop_Etages ;
12    New_Line ;
13  end loop Loop_Tours ;
14 end Afficher ;
```

Remarquez que rien dans cette portion de code ne suggère que nous ayons un maximum de 10 disques, selon la définition des types `Étage` et `Tour`. La ligne 10 montre l'accès aux différents éléments : `jeu.tours` est le tableau des tours, `jeu.tours(ind_tour)` désigne l'enregistrement de type `Tour` à l'indice `ind_tour` dans ce tableau, etc.

Les enregistrements supportent de plus les opérations élémentaires que sont l'affectation (`:=`) et la comparaison (`=` et `/=`). Ces opérations sont appliquées récursivement sur chacun des éléments de l'enregistrement. Par exemple, si la longue valeur d'initialisation de la déclaration plus haut vous rebute, voici une procédure qui permet d'initialiser une variable de type `Jeu_Hanoï`, en fonction du nombre initial de disques que l'on veut sur la première tour :

```
 1 procedure Init(jeu:     out   Jeu_Hanoï ;
 2                nb_init: in    Étage) is
 3    jeu_tmp: Jeu_Hanoï ;
 4 begin
 5    jeu_tmp.tours(1).dernier_étage := nb_init ;
 6    for ind_etage in 1..nb_init
 7    loop
 8       jeu_tmp.tours(1).pile(ind_etage) :=
 9          Disque(Étage'Last-(ind_etage-1)) ;
10    end loop ;
11    jeu := jeu_tmp ;
12 end Init ;
```

Le passage par une variable temporaire nous assure que tous les éléments auront au moins une valeur par défaut. Le traitement se limite donc aux éléments à modifier selon le paramètre `nb_init`. Le résultat de l'initialisation est renvoyé dans le paramètre en sortie `jeu` grâce à l'affectation de la ligne 11.

Remarquez le transtypage ligne 9 : il est nécessaire, car le membre `pile` contient des éléments de type `Disque`, tandis que `ind_etage` et `Étage'Last` sont de type `Étage`. Et encore une fois, nous n'avons pas utilisé explicitement la limite de 10 du nombre de disques : elle est tout simplement récupérée par l'attribut `Last` du type `Étage`.

Représentation

Une différence fondamentale entre des langages comme le C et Ada est que ce dernier n'impose rien quant à la représentation en mémoire des structures de données déclarées dans le programme. Le compilateur a toute latitude pour stocker tout cela en mémoire comme bon lui semble, en fonction des options qui lui sont données pour minimiser l'encombrement mémoire ou optimiser la vitesse d'exécution. Par exemple, un enregistrement ne contenant que deux entiers d'un type borné entre 1 et 10 pourrait parfaitement n'occuper qu'un seul octet, ce qui est binairement suffisant ; un tableau de six booléens (type `Boolean`) pourrait n'occuper que six bits, c'est-à-dire moins d'un octet ; au contraire, notre enregistrement de deux entiers pourrait s'étaler sur 16 octets, si on demande la meilleure performance possible et que le matériel sous-jacent est plus efficace pour accéder aux données tous les 8 octets... Ne soyez pas étonnés de cela, Ada a été conçu pour pouvoir être utilisé pour des logiciels embarqués (comme les sondes spatiales) où les contraintes et limites matérielles sont parfois très fortes.

Une différence fondamentale entre des langages comme le C et Ada est que ce dernier n'impose rien quant à la représentation en mémoire des structures de données déclarées dans le programme.

Mais parfois ces « aléas » dans la représentation mémoire ne sont pas souhaitables : on peut vouloir contrôler très précisément comment les données sont agencées en mémoire. Ada propose différents moyens pour cela, par le biais d'attributs et de directives `pragma`. Cela peut s'avérer fort utile si le programme doit manipuler directement un périphérique par ses ports d'entrées/sorties.

Voyons un petit programme d'exemple :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure test_bits is
 3    type Petit is new Integer range 1..5 ;
 4 --   for Petit'Size use 5 ;
 5    type Enreg is
 6    record
 7       a: Petit ;
 8       b: Boolean ;
 9       c: Petit ;
10       d: Petit ;
11       e: Boolean ;
12    end record ;
13 --   pragma Pack(Enreg) ;
14    type Tab1 is array (Petit) of Petit ;
15 --   for Tab1'Component_Size use 10 ;
16    type Tab2 is array (Petit) of Enreg ;
```

```
17 --   pragma Pack(Tab2) ;
18 e: Enreg ;
19 begin
20    Put_Line("Petit'Size = " &
21       Integer'Image(Petit'Size)) ;
22    Put_Line("Enreg'Size = " &
23       Integer'Image(Enreg'Size)) ;
24    Put_Line("Enreg.a'First_Bit = " &
25       Integer'Image(e.a'First_Bit)) ;
26    Put_Line("Enreg.b'First_Bit = " &
27       Integer'Image(e.b'First_Bit)) ;
28    Put_Line("Enreg.c'First_Bit = " &
29       Integer'Image(e.c'First_Bit)) ;
30    Put_Line("Enreg.d'First_Bit = " &
31       Integer'Image(e.d'First_Bit)) ;
32    Put_Line("Enreg.e'First_Bit = " &
33       Integer'Image(e.e'First_Bit)) ;
34    Put_Line("Enreg.a'Position = " &
35       Integer'Image(e.a'Position)) ;
36    Put_Line("Enreg.b'Position = " &
37       Integer'Image(e.b'Position)) ;
38    Put_Line("Enreg.c'Position = " &
39       Integer'Image(e.c'Position)) ;
40    Put_Line("Enreg.d'Position = " &
41       Integer'Image(e.d'Position)) ;
42    Put_Line("Enreg.e'Position = " &
43       Integer'Image(e.e'Position)) ;
44    Put_Line("Tab1'Size = " &
45       Integer'Image(Tab1'Size)) ;
46    Put_Line("Tab1'Component_Size = " &
47       Integer'Image(Tab1'Component_Size)) ;
48    Put_Line("Tab2'Size = " & Integer'
49       Image(Tab2'Size)) ;
50    Put_Line("Tab2'Component_Size = " &
51       Integer'Image(Tab2'Component_Size)) ;
52 end test_bits ;
```

Répétons-le : tout n'est que souplesse en Ada.

Les lignes commentées (4, 13, 15 et 17) contiennent des instructions précisant la représentation mémoire à adopter pour différents objets. Le programme affiche ensuite simplement des informations sur la taille et la position de ces objets. Aspect essentiel à bien garder en mémoire : les tailles et positions sont données en bits et non en octets.

Les attributs impliqués ici sont :

- `Size` donne la taille (en bits, donc – voir [AARM 13.3-40,45]) de l'objet lorsqu'il est interrogé (par exemple ligne 21) ou permet de la spécifier (par exemple ligne 4) ;
- `Component_Size` ne concerne que les types tableaux et permet d'obtenir (ligne 47) ou de définir (ligne 15) la taille occupée par chaque élément dans le tableau ;à noter qu'il est permis à l'implémentation [AARM 13.3-73] d'adapter la valeur donnée ;
- `First_Bit` [AARM 13.5.2] est appliqué à un élément d'un enregistrement et donne le décalage du premier bit de cet élément par rapport au début de l'octet mémoire qui le contient (en nombre de bits) ; l'attribut `Last_Bit` donne le décalage du dernier bit ; ces deux attributs ne peuvent pas être définis (enfin, pas directement) ;
- Enfin, `Position` donne la position d'un élément dans un enregistrement, cette fois exprimée en octets ; cet attribut ne peut pas être défini.

Tout cela nous donne déjà pas mal de souplesse. Mais ce n'est pas tout. Le langage Ada fait un usage assez intensif des directives `pragma`, dont le rôle est essentiellement de donner des informations ou des conseils au compilateur. La première que nous rencontrons est la directive `pragma Pack` (lignes 13 et 17) : elle indique au compilateur qu'il doit tenter de minimiser l'encombrement mémoire des instances du type auquel elle est appliquée, qui doit être un type composé (tableau ou enregistrement). C'est une recommandation forte, qui doit être suivie même au prix de la vitesse d'exécution.

Pour fixer les idées, exécutez le programme précédent en jouant sur les lignes en commentaires.

La troisième version du programme, si elle devait créer beaucoup d'instances du type `Enreg` dans un tableau, occuperait beaucoup moins de mémoire (presque neuf fois moins) que la première version. Par contre, elle serait sensiblement plus lente, car il est nécessaire d'ajouter des instructions de calcul et manipulations binaires pour accéder aux éléments. Un test rapide montre que le parcours d'un tableau de 10000 éléments est presque quatre fois plus lent dans la première version (plus de huit fois si on compile avec l'optimisation -O2).

Il existe encore d'autres possibilités pour affiner la représentation mémoire : consultez la norme pour plus de détails.

Erratum

Je profite de ce chapitre sur les représentations des données pour m'excuser d'une erreur qui s'est glissée dans l'article du moins dernier, au sujet des types énumérés, précisément ce membre de phrase : « il n'est pas possible d'associer explicitement une valeur numérique à une valeur symbolique ». C'est tout simplement faux : la section 13.4 du AARM expose justement une syntaxe permettant d'effectuer une telle association. Par exemple :

```
type TT is (AA, BB, CC) ;
for TT use (AA=>25, BB=>3213, CC=>6598) ;
```

Les symboles AA, BB et CC du type TT seront représentés en mémoire par les entiers 25, 3213 et 6598.

Paramétrisation

Répétons-le : tout n'est que souplesse en Ada. Il est ainsi possible de paramétriser un enregistrement, selon un ou plusieurs critères. Le cas le plus simple consiste à utiliser une valeur pour dimensionner un tableau non contraint. Prenons un exemple :

```
1 procedure Record_Param is
2    type TTab is array (Integer range <>) of Integer ;
3    type TRec(taille: Integer) is
4    record
5       t: TTab(1..taille) := (others => 0) ;
6    end record ;
7 nb: Integer := 10 ;
8 r1: TRec(nb) ;
```

Effets d'instructions de représentation

TOUTES LES LIGNES DÉSACTIVÉES	LIGNES 4 ET 17 ACTIVÉES	TOUTES LES LIGNES ACTIVÉES
Petit'Size = 3 Enreg'Size = 136 Enreg.a'First_Bit = 0 Enreg.b'First_Bit = 0 Enreg.c'First_Bit = 0 Enreg.d'First_Bit = 0 Enreg.e'First_Bit = 0 Enreg.a'Position = 0 Enreg.b'Position = 4 Enreg.c'Position = 8 Enreg.d'Position = 12 Enreg.e'Position = 16 Tab1'Size = 160 Tab1'Component_Size = 32 Tab2'Size = 800 Tab2'Component_Size = 160	Petit'Size = 5 Enreg'Size = 40 Enreg.a'First_Bit = 0 Enreg.b'First_Bit = 0 Enreg.c'First_Bit = 0 Enreg.d'First_Bit = 0 Enreg.e'First_Bit = 0 Enreg.a'Position = 0 Enreg.b'Position = 1 Enreg.c'Position = 2 Enreg.d'Position = 3 Enreg.e'Position = 4 Tab1'Size = 40 Tab1'Component_Size = 8 Tab2'Size = 200 Tab2'Component_Size = 40	Petit'Size = 5 Enreg'Size = 17 Enreg.a'First_Bit = 0 Enreg.b'First_Bit = 5 Enreg.c'First_Bit = 6 Enreg.d'First_Bit = 3 Enreg.e'First_Bit = 0 Enreg.a'Position = 0 Enreg.b'Position = 0 Enreg.c'Position = 0 Enreg.d'Position = 1 Enreg.e'Position = 2 Tab1'Size = 50 Tab1'Component_Size = 10 Tab2'Size = 88 Tab2'Component_Size = 17
Remarquez comme la taille de Petit est petite, seulement 3 bits : le compilateur a déterminé que c'était suffisant pour contenir les valeurs de ce type. Dans l'enregistrement, tous les First_Bit sont à zéro : les composants commencent tous à une limite d'octet. La taille des éléments du type Tab1 nous montre qu'en réalité Petit occupe 4 octets en mémoire.	La ligne 4 impose une taille pour Petit : celle-ci influe naturellement sur le reste. En particulier, l'enregistrement est plus petit : le fait de donner une taille à Petit réduit son encombrement, bien que cette taille soit plus grande que la taille minimum déterminée précédemment. Mais il y a encore beaucoup de vide dans tout cela...	Cette fois l'encombrement de l'enregistrement est véritablement minimisé : il n'occupe plus que 17 bits, soit à peine plus que deux octets. Il y a encore un peu de perte dans Tab2 (3 bits), parce que le compilateur refuse de créer des tableaux d'une taille non multiple de 8 étant donné le matériel sous-jacent (en l'occurrence, un Intel Pentium 4). L'augmentation de la taille de Tab1 vient du fait qu'on a imposé une taille « trop » grande à ses éléments (ligne 15).

```
 9 r2: TRec(nb) ;
10 r3: TRec(2*nb) ;
11 begin
12    r1 := r2 ;
13    r1 := r3 ;
14 end Record_Param ;
```

Nous déclarons ligne 2 un type tableau non contraint TRec. Il est alors normalement interdit d'utiliser ce type comme élément d'un autre tableau ou d'un autre enregistrement, à moins de le contraindre. Les lignes 3 à 6 définissent un type enregistrement TRec, qui contient justement un membre de ce type tableau. Celui-ci est contraint grâce à un paramètre donné à TRec, nommé taille. Dans le cadre d'un enregistrement, on appelle un tel paramètre un discriminant. Celui-ci fait partie intégrante de TRec, sa valeur étant obtenue lors de l'instanciation du type (lignes 8 à 10). Il joue alors le rôle d'un membre, que l'on peut consulter comme n'importe quel autre (par exemple avec r1.taille), mais dont on ne peut modifier la valeur : une instruction comme r1.taille := 5 est interdite. Remarquez que la valeur du discriminant n'est pas nécessairement statique, elle peut être issue d'une variable ou d'un calcul.

Les affectations entre instances des lignes 12 et 13 sont syntaxiquement valides. Toutefois, si la première ne pose aucun problème, la seconde provoquera la levée d'une exception (nommément Constraint_Error) : les affectations entre enregistrements contenant des discriminants ne sont en effet possibles que si les valeurs des discriminants sont égales. Les discriminants peuvent également être utilisés pour adapter l'apparence d'un type enregistrement. Imaginons un instant que nous voulions réaliser une petite application pour stocker notre vaste collection de livres et de films. Ces deux types d'information ont au moins une chose en commun : le titre. Par contre, on peut vouloir stocker le nombre de pages pour un livre, mais la durée pour un film. Il nous faudrait alors définir deux types différents pour les représenter... à moins d'utiliser un discriminant :

```
 1 type Type_Info is (Livre, Film) ;
 2 type Info(nature: Type_Info) is
 3 record
 4    titre: String(1..200) := (others=>' ') ;
 5    case nature is
 6       when Livre =>
 7          nb_pages: Integer ;
 8          isbn: String(1..13) ;
 9       when Film =>
10          durée: Float ;
11    end case ;
12 end record ;
```

La syntaxe utilisée est très similaire à celle du choix multiple. Selon la valeur du discriminant nature, l'enregistrement Info présentera

différents visages. Si cette valeur est le symbole `Livre`, alors `Info` contiendra (en plus du titre) les éléments `nb_pages` et `isbn`. Si cette valeur est `Film`, alors `Info` contiendra `durée`. Ce procédé est à rapprocher des types `union` du C/C++. Remarquez que les différents visages d'un enregistrement ainsi « discriminé » ne doivent pas nécessairement contenir des types compatibles ou un même nombre de champs.

La prochaine norme Ada apportera la notion d'« interfaces », inspirée par le langage Java, réalisant ainsi une forme limitée d'héritage multiple.

L'utilisation d'un tel type se fait ainsi :

```
1 info1: Info(Livre) ;
2 info2: Info(Film) ;
3 -- .....
4 info1.nb_pages := 200 ;
5 info2.durée := 1.5 ;
6 info2.nb_pages := 1 ;
```

La valeur du discriminant est donnée à la déclaration, puis on accède normalement aux éléments. Remarquez la dernière ligne : elle est syntaxiquement correcte, mais comme `info2` a été déclaré comme étant un `Film`, il ne montre pas de champs nommé `nb_pages`. À l'exécution, l'exception `Contraint_Error` sera levée. Il est également possible de donner une valeur initiale lors de la déclaration d'une variable, ce qui est en fait le seul cas où l'affectation au discriminant est autorisée :

```
1 info3: Info := (
2    nature => Livre,
3    titre => "Titre" & Chaine_Vide(1..195),
4    nb_pages => 100,
5    isbn => "1-2345-6789-A") ;
```

La valeur du discriminant est donnée dans la liste d'initialisation (bien qu'il n'eût pas été incorrect de la répéter dans le nom du type en ligne 1). Cette fois, si on donne une valeur à un élément qui n'existe pas selon le discriminant donné, on obtient une erreur de compilation. De même, si la valeur du discriminant n'est pas donnée avec le nom du type, la liste d'initialisation est obligatoire. Il est ainsi interdit de déclarer une variable simplement avec `info4: Info;`, sauf dans le cas d'un paramètre de sous-programme, par exemple :

```
1 procedure Afficher(i: Info) is
2 begin
3    Put_Line("Titre = " & i.titre) ;
4    case i.nature is
5       when Livre =>
6          Put_Line("ISBN = " & i.isbn) ;
7       when Film =>
8          Put_Line("Durée = " & Float'Image(i.durée)) ;
9    end case ;
10 end Afficher ;
```

Dernier mot, rien n'interdit de déclarer plusieurs discriminants de types différents pour un type enregistrement, de donner une valeur par défaut ou d'utiliser le mécanisme des paramètres nommés lors de la déclaration d'une variable. La syntaxe des discriminants d'enregistrement est en fait assez proche de celle des déclarations et appels de sous-programmes.

Extension

Dernier aspect des enregistrements que nous verrons aujourd'hui, en manière d'anticipation sur de prochains articles : les enregistrements marqués (`tagged`) et l'extension de leur contenu. Prenons un exemple classique, un enregistrement décrivant un cercle :

```
type Cercle is tagged record
   x: Float ;
   y: Float ;
   rayon: Float ;
end record ;
```

Un enregistrement classique, mais remarquez tout de même la présence du mot `tagged` dans la définition. Maintenant nous voudrions une ellipse, décrite par un centre, un petit rayon et un grand rayon. Il serait naturellement possible de recréer un type pour cela. Mais il serait peut-être plus intéressant de conserver le lien de parenté qui existe entre un cercle et une ellipse... c'est-à-dire, de construire notre ellipse comme une extension de notre cercle. Comme ceci :

```
type Ellipse is new Cercle with
record
   grand_rayon: Float ;
end record ;
```

Ceci définit un type `Ellipse` comme étant un nouveau `Cercle` contenant en plus un élément nommé `grand_rayon`. Selon cette représentation, une `Ellipse` « est aussi » un `Cercle` (ce qui est discutable sur le plan mathématique, mais c'est une autre histoire). Dit autrement, le type `Cercle` est l'ancêtre du type `Ellipse` ou encore le type `Ellipse` dérive du type `Cercle`... Certains auront probablement reconnu là un vocabulaire habituellement utilisé pour parler de programmation objet. Bonne nouvelle, c'est précisément de cela qu'il s'agit : ce mécanisme, introduit par la norme Ada95 (donc absent de la norme Ada83), est destiné à apporter à Ada les fonctionnalités d'un langage orienté objet. Ce que nous venons de voir n'est rien de plus que de l'héritage. Ada95 ne supporte que l'héritage simple. La prochaine norme Ada apportera la notion d'« interfaces », inspirée par le langage Java, réalisant ainsi une forme limitée d'héritage multiple. Rassurez-vous, les aspects objet de Ada feront l'objet d'un article spécifique, voire de plusieurs.

Conclusion

Voilà en ce qui concerne les enregistrements en Ada. Comme le laisse supposer la dernière section, nous les retrouverons bientôt – en fait, nous les utiliserons sans cesse au fil de nos découvertes. La prochaine fois, nous aborderons le mécanisme des paquetages, par lequel Ada permet les compilations séparées et la définition et la diffusion de composants réutilisables – autrement dit, nous découvrirons les bibliothèques en Ada.

Yves Bailly,