# Types

Chapitre 4

# Notion de type de données

Définition

- Un type de données est un ensemble de valeurs et un ensemble d'opérations pouvant être effectuées sur ces valeurs.

Notion d'objet informatique

- Toute entité à laquelle un type est associé (variable, constante, fonction, procédure, exception, paquetage, ...) est un objet informatique.

# Intérêt du concept de type

Comment fournir à la machine les informations nécessaires pour qu'elle soit en mesure de vérifier que l'utilisation des objets d'un programme est bien conforme à l'intention préalable du programmeur ?

Lorsque nous déclarons un objet (variable, constante, fonction, ...), nous spécifions les contraintes de son utilisation, l'ensemble des valeurs qui pourront lui être associées et ainsi définir le cadre légitime de son utilisation.

L'utilisation de cet objet sera précédée d'une vérification effectuée par le compilateur. La valeur qu'il prend appartient-elle à cet ensemble ? Son utilisation est-elle conforme aux contraintes spécifiées ?

Le concept de type permet d'établir un lien sémantique entre la déclaration et l'utilisation d'un objet.

# Ada et le concept de type

Tout objet est introduit dans un programme au moyen d'une déclaration. Une déclaration permet d'associer un type à un objet. Toute expression du langage est donc typée.

En Ada, tout objet, toute expression possède un type et un seul. On dit qu'Ada est un langage fortement typé.

Il existe des types prédéfinis (`Integer`, `Character`, `Float`, `Boolean`).

# Le type **Integer**

Ensemble des valeurs

- Les valeurs du type `Integer` forment un sous ensemble de l'ensemble **Z**, symétrique par rapport à 0.

- L'ensemble des valeurs du type `Integer` dépend de la machine. Pour une machine dont les mots sont de 32 bits, les valeurs de l'ensemble `Integer` s'étalent entre $-2^{31}$ et $+2^{31}-1$.

- Pour des raisons de portabilité des programmes, c'est à dire pour faire en sorte que les programmes soit indépendants de la taille des mots de la machine sur laquelle ils doivent s'exécuter, Ada a introduit les attributs `first` et `last`.

# Attributs du type `Integer`

Pour le programmeur le plus petit entier est `Integer'first`, le plus grand est `Integer'last`. Pour une machine 32 bits, `Integer'first = -2147483648(-2`$^{31}$`)`et `Integer'last = +2147483647 (2`$^{31}$`-1)`.

L'ensemble des valeurs du type `Integer` est donc défini par l'intervalle : `Integer'first..Integer'last`

Une installation peut offrir les types prédéfinis : `Short_Integer` et `Long_Integer` pour coder les entiers sur, respectivement, 1/2 mot ou 2 mots.

# Opérations du type **Integer**

| **opérations** | **symboles** |
|---|---|
| affectation | := |
| opérateurs relationnels | <, <=, >=, > |
| opérateurs d'égalité | =, /= |
| opérateurs unaires | +, -, abs |
| opérateurs binaires | +, -, *, /, **, mod, rem |

# Les opérateurs **mod** et **rem**

Opérateur `mod` : le signe du résultat est celui du deuxième opérande

```
-9 mod 2 = -(9 mod 2) = -1
(-9) mod 2 = 1
9 mod 2 = 1
9 mod (-2) = -1
```

Opérateur `rem` : le signe du résultat est celui du premier opérande

```
-9 rem 2 = -(9 rem 2) = -(+1) = -1
(-9) rem 2 = -1
9 rem 2 = +1
9 rem (-2) = +1
```

# Autres type entiers

Ils sont obtenus en réduisant l'intervalle des valeurs possibles.

- on définit ainsi des sous-types d'entiers.
- l'ensemble des opérations reste le même.

**Exemple**

```
subtype Entier is Integer range 1..200;
```

# Types énumératifs

**Ensemble de valeurs** : c'est un ensemble ordonné d'éléments identificateurs et/ou littéraux.

```
type Voyelle is (a,e,i,o,u,y);
-- valide car les valeurs sont considérées
-- comme des identificateurs

type Voyellebis is ('a','e','i','o','u','y');
-- valide car les valeurs sont considérées
-- comme des littéraux
-- les quotes seront saisies en entrée et
-- reproduites à l'impression

type Reponse is (oui,non,peut_etre);
```

# Relation d'ordre sur les valeurs

```
oui<non<peut_etre
a<e<i<o<u<y
```

# Opérations sur les types énumératifs

| **opérations** | **symboles** |
|---|---|
| affectation | := |
| opérateurs relationnels | <, <=, >=, > |
| opérateurs d'égalité | =, /= |

# Attributs des types énumératifs

```
Voyelle'first=a
Voyelle'last=y
Voyelle'succ(a)=e
Voyelle'pred(e)=a
Voyelle'pos(a)=0
Voyelle'val(1)=e
Reponse'image(oui)= "oui"
```

Le successeur de `y` n'existe pas. Le prédécesseur de `oui` non plus.

**Remarque** : on notera la différence entre l'identificateur `a`, le caractère `'a'` et la chaîne de caractères `"a"`.

# Le type **Boolean**

Ensemble de valeurs

`FALSE, TRUE`

Relation d'ordre sur les valeurs

`FALSE<TRUE`

# Opérations sur le type **Boolean**

| **opérations** | **symboles** |
|---|---|
| affectation | := |
| opérateurs relationnels | <, <=, >=, > |
| opérateurs d'égalité | =, /= |
| opérateurs logiques | not, and, or, xor |
| opérateurs logiques spécifiques | and then, or else |

# Tables de vérité de opérateurs booléens

| **and** | false | true |
|---|---|---|
| false | false | false |
| true | false | true |

| **or** | false | true |
|---|---|---|
| false | false | true |
| true | true | true |

| **xor** | false | true |
|---|---|---|
| false | false | true |
| true | true | false |

# Exemples (1/2)

```
('a'<'y' and 'b'>'x') = FALSE

((X mod 400=0) or (X mod 4=0 and X mod
  100/=0))=TRUE
-- si X représente une année bissextile
```

# Exemples (2/2)

**`and then`** : `(A/=0 and then B/A=X)`

Contrairement à l'évaluation de l'opérateur `and`, Ada évalue d'abord l'expression `A/=0`. Si la réponse est `FALSE`, l'expression `B/A=X` n'est pas évaluée, ce qui protège d'une éventuelle division par `0` qui entraînerait une erreur à l'exécution.

**`or else`** : `(A=0 or else B/A=X)`

L'idée est la même. Si l'expression `A=0` vaut `TRUE`, il est inutile d'évaluer `B/A=X`. On évite aussi, dans ce cas, la division par `0`.

# Le type **Character**

**Ensemble de valeurs**

Il s'agit d'un type énuméré où chaque valeur est un caractère.

En Ada 83, c'est l'ensemble des caractères définis par le code ASCII 7 bits, c'est à dire les caractères usuels à l'exclusion des caractères accentués.

**Opérations**

| opérations | symboles |
|---|---|
| affectation | := |
| opérateurs relationnels | <, <=, >=, > |
| opérateurs d'égalité | =, /= |

# Les types réels

Il existe 3 manières de représenter des nombres réels en Ada :

1. les types `Float` permettent de spécifier le nombre de décimaux significatifs, mais ne garantissent pas un degré de précision sur l'ensemble des valeurs
2. les types `Fixed` maintiennent un degré de précision quelle que soit la valeur
3. les types `Decimal` combinent les deux exigences précédentes

Le plus communément utilisé est le type `Float`, mais les exigences de précision des applications scientifiques sont satisfaites par les différents choix offerts par Ada.

Seul le type `Float` sera utilisé dans ce cours.

# Le type **Float**

**Ensemble de valeurs**

Il s'agit du sous-ensemble de l'ensemble **R** des nombres réels. On peut écrire des littéraux réels selon 2 notations :

```
123.4 ou 1.234E+2
```

**Opérations**

| **opérations** | **symboles** |
|---|---|
| affectation | := |
| opérateurs relationnels | <, <=, >=, > |
| opérateurs d'égalité | =, /= |
| opérateurs arithmétiques | +, -, *, / , abs |

# Remarques (1/2)

Les conséquences du typage fort interdisent des opérations mixant le type `Float` et un type entier.

```
i:Integer:=2;
r:Float:=5.6;
...
if r=i then ...
-- opération illicite
else ...
end if;
```

# Remarques (2/2)

De même, il faut se méfier des comparaisons entre 2 valeurs flottantes. En effet, les réels sont représentés par une approximation binaire sur un nombre limité de bits.

```
r1:Float:=3.456387;
r2:Float:=3.456386999;
...
if r1=r2 then ...
else ...
end if;
```

On pourra constater que, bien que les valeurs de `r1` et `r2` soient différentes, la comparaison `r1=r2` sera évaluée à `TRUE` par Ada.

# Types discrets

Les types discrets définissent des ensembles de valeurs élémentaires énumérables et totalement ordonnées.

Des opérateurs prédéfinis, appelés attributs, sont applicables sur les valeurs de ces types : `first, last, succ, pred, pos, val, value, image.`

Ada fournit 3 types discrets prédéfinis :

```
Integer
Boolean
Character
```

# Attributs des types discrets

Le type `Jour` est un exemple typique de type discret, l'effet de chacun des attributs est décrit dans le tableau suivant :

| attribut | valeur | commentaire |
|---|---|---|
| `Jour'first` | `lundi` | 1ère valeur du type |
| `Jour'last` | `dimanche` | dernière valeur du type |
| `Jour'succ(mardi)` | `mercredi` | valeur suivante selon l'ordre d'énumération |
| `Jour'pred(mardi)` | `lundi` | valeur précédente selon l'ordre d'énumération |
| `Jour'pos(mardi)` | `1` | rang dans l'ordre des valeurs |
| `Jour'val(5)` | `samedi` | valeur de rang 5 |
| `Jour'value("mardi")` | `mardi` | conversion de la chaîne "mardi" en mardi |
| `Jour'image(mardi)` | `"mardi"` | chaîne correspondant au symbole mardi |

# Notion de tableau

**Définition formelle**

Un n-uplet est un élément d'un produit cartésien de n ensembles identiques (les composantes) :

$$\pi^n \; A=A*A*A*...*A=\{(a_1,a_2,...,a_n)|a_i \in A\}$$

- Un tableau Ada représente un n-uplet dont toutes les composantes sont de même type.
- Le nombre de composantes d'un tableau est appelé dimension.
- La position d'un élément dans une composante d'un tableau est appelée indice.
- La valeur d'un indice doit appartenir à un type discret.
- Le type tableau représente l'ensemble des tableaux :
  - de même dimension
  - dont les composantes ont le même type
  - dont les indices sont définis dans un intervalle discret

# Syntaxe de définition d'un type tableau

```
<définition_type_tableau>::=
        array(<borne_inf>..<borne_sup>) of <type_composante>
      | array(<type_indices>) of <type_composante>
      | array(<type_indices> range <borne_inf>..<borne_sup>)
                                          of <type_composante>
```

où

`<type_indice>` représente le type des indices.

`<type_indice>` est un type discret.

`<borne_inf>` et `<borne_sup>` représentent les bornes inférieures et supérieures d'un type d'indices.

`<type_composante>` représente le type des composantes du tableau.

`<type_composante>` est un type quelconque.

# Exemples

```
array(lundi..vendredi) of Visite;
-- l'intervalle lundi..vendredi appartient
-- au type discret Jour
-- le type Visite est un type quelconque
-- que l'on suppose défini
array(Jour) of Visite;
array(Integer range 1..7) of Matiere;
```

# Types tableau

**Ensemble des valeurs**

Les valeurs d'un type tableau contraint sont des objets tableaux.

Ce sont des objets composés d'éléments de même type et dont les bornes sont identiques.

**Ensemble des opérations**

| **opérations** | **symboles** |
|---|---|
| affectation | `:=` |
| opérateurs relationnels | `<, <=, >=, >` |
| opérateur de concaténation | `&` |
| opérateur de sélection | `( )` |

# Déclaration d'un type tableau

La déclaration d'un type tableau associe un nom à la définition du type.

```
<déclaration_type_tableau>::=
   type <ident_du_type> is <définition_type_tableau>;
```

avec

`<ident_du_type>` est un identificateur Ada.

**Exemples**

```
type Heure is (10_12,14_16,16_18,18_20);
type Motif is (reunion,visite,pot,rendez_vous,rien);
type Semaine is array(Heure,Jour) of Motif;
type Agenda is array(Positive range 1..52) of Semaine;
```

# Types tableaux anonymes

Il est possible de déclarer des objets tableaux dont le type n'a pas été défini.

La déclaration associe alors directement un identificateur d'objet tableau à la définition de son type.

```
X,Y: array('A'..'Z') of Integer;
-- il s'agit alors de deux définitions
-- différentes, x et y ont deux types distincts
```

# Déclaration de variables d'un type tableau

Une déclaration d'un objet de type tableau associe un nom de variable à un type tableau.

- Déclaration de variables d'un type tableau anonyme :

```
hebdo1,hebdo2:array(vendredi..dimanche) of Integer;
--Les variables hebdo1 et hebdo2 sont de type différent
```

- Déclaration de variables d'un type tableau explicitement déclaré

```
hebdo3,hebdo4:Semaine;
-- Les variables hebdo3 et hebdo4 sont de même type
A,B:Tab;
conge:Semaine;
C:Agenda;
```

# Agrégats (1/2)

Un agrégat est une valeur d'un type tableau.
C'est une liste d'associations entre un indice et une valeur.

Il peut être construit syntaxiquement de 3 manières différentes :

- Association implicite (d'après l'ordre de lecture)

```
(9,9,9,11,19)
-- valeur de tableau du type Semaine
```

# Agrégats (2/2)

- Association explicite (l'indice est explicite). Cette notation dispense de respecter l'ordre des indices

```
(jeudi=>11,lundi=>9,mardi=>9,vendredi=>7,mercredi=>9,
 samedi=>7,dimanche=>7)
```

ou bien

```
(lundi..mercredi=>9,jeudi=>11,others=>7)
```

ou bien

```
(lundi|mardi|mercredi=>9,jeudi=11,others=>7)
-- la construction others permet de compléter
-- l'initialisation des champs qui suivent selon
-- l'ordre du type des indices
-- les constructions a..b ou a|b permettent de
-- factoriser l'initialisation de plusieurs champs
-- successifs
```

- Panachage

```
(9,9,mercredi=>9,11,others=>7)
-- l'ordre est respecté avant l'association explicite
```

# Initialisation de variable d'un type tableau

La valeur d'initialisation d'un tableau est un agrégat :

```
RDV:array(lundi..mercredi) of Integer:=(9,13,10);
```

Après initialisation, on obtient le tableau

| 9 | 13 | 10 |
|---|---|---|
| lundi | mardi | mercredi |

# Affectation d'une variable tableau

```
type Vecteur5 is array(1..5) of Float;
monVecteur:Vecteur5;
monVecteur:=(1.0,2..3=>2.0,others=>0.0);
```

# Sélection d'un composant de tableau

Pour sélectionner le composant d'un tableau, il suffit de préciser son indice.

```
monVecteur(2):=2.5;
-- la valeur du second composant est modifiée
monVecteur(3):=monVecteur(1);
-- le 3ème composant prend la valeur du premier
-- Les types des expressions droite et gauche
-- sont les mêmes
hebdo1,hebdo2: array(vendredi..dimanche) of Integer;
hebdo1:=(8,7,14);
hebdo2:=(6,5,7);
hebdo1:=hebdo2;
-- affectation illégale
hebdo1(vendredi):=hebdo2(samedi);
-- affectation légale car il s'agit d'une
-- affectation d'un entier à une variable entière
```

# Sélection d'une tranche de tableau (sous-tableau)

Pour sélectionner un sous-tableau (suite de composants contigus), il suffit de préciser son intervalle d'indice :

```
hebdo1(samedi..dimanche):=(12,12);
monVecteur(1..3):=monVecteur(3..5);
-- les deux sous-tableaux sont de même dimension
-- leurs composants sont de même type
```

# Concaténation

L'opérateur **&** permet de concaténer 2 tableaux (et/ou sous-tableaux) dont les composantes sont de même type :

```
hebdo3,hebdo4,hebdo5:Semaine;
hebdo3:=(9,8,others=>16);
hebdo4:=(9,others=>10);
hebdo5:=hebdo3(lundi) & hebdo4(jeudi..vendredi) &
   hebdo3(mardi) & hebdo4(mardi);
```

# Attributs

Soit la déclaration :

```
tab:array (5..100) of
   Character:=('a','b','c','d','e',others=>'x');
```

| expression | valeur | signification |
|---|---|---|
| `tab'first` | `5` | borne inférieure |
| `tab'last` | `100` | borne supérieure |
| `tab'length` | `95` | longueur de l'intervalle d'indices |
| `tab'range` | `5..100` | l'intervalle d'indices |

# Types tableaux non contraints

Pour généraliser les types tableaux, il est possible de les paramétrer en évitant de borner l'intervalle des indices.

**Construction syntaxique**

```
<définition_type_tableau_non_contraint>::=
        array (<type_des_indices> range <>) of
      <type_des_composantes>;
```

Le type des indices doit être discret.

La construction **range <>** indique que la taille des tableaux ne sera connue que lorsqu'une valeur aura été spécifiée pour le <type_des_indices>.

# Exemple

```
type Vecteur is array(Integer range <>) of Jour;
vecteur_1:Vecteur(1..6);
-- l'intervalle des indices est fixé
vecteur_2:Vecteur(1..3);
```

**Remarque** : Le système ne peut pas allouer l'espace mémoire nécessaire à une variable dont le type est non contraint car il ne connaît pas sa taille.

# Calcul de la moyenne des valeurs d'un tableau de réels

```
declare
 type Vecteur_reels is array (Integer range <>) of
Float;
 V:Vecteur_reels(1..5):=(3.47,8.7,5.32,1.09,0.004);
 somme:Float:=0.0;
begin
 if V'length=0 then
   put( "le vecteur est vide" );
 else
   for I in V'range loop
     somme:=somme+V(I);
   end loop;
   put( "Moyenne des composants :" );
   put(somme/Float(V'last-V'first+1));
 end if;
end;
```

# Tableaux dynamiques

Un tableau dynamique est un tableau dont les bornes sont évaluées au cours de l'exécution.

```
procedure dynamique is
  type TabDyn is array(Integer range <>) of Float;
  N:Integer;
begin
  get(N);
  declare
    unTabDyn:TabDyn(1..N):=(others=>0.0);
    -- N n'est pas connu à la compilation
  begin
    -- suite d'instructions
  end;
end dynamique;
```

# Type **String**

Il définit l'ensemble des chaînes de caractères de longueur quelconque.

C'est un type non contraint prédéfini. Il est défini dans le paquetage Standard.

```
type String is array(Positive range <>) of
   Character;
ligne:String(1..80);
-- représente une chaîne de 80 caractères
```

# Lecture d'une chaîne de caractères de longueur variable (1/2)

```
with Ada.Text_io; use Ada.Text_io;
procedure chaine is
  ligne:String(1..80);
  -- on suppose une longueur maximum de la chaîne de 80
  -- caractères
  lg:Natural range 0..80;
begin
  put( "Tapez votre nom : " );
  get_line(ligne,lg);
  -- lg contient le nombre de caractères effectivement saisis
  put( "ligne= " );put(ligne);
  declare
    nom:String(1..lg):= ligne(1..lg);
  -- le type String est contraint
  begin
    put( "nom=" );put(nom);
  end;
end chaine;
```

# Lecture d'une chaîne de caractères de longueur variable (2/2)

**Résultat**

```
Tapez votre nom : Daniel
ligne=DanielèTÿ3 (ôôô%èxõõÃÿÔÔ%{êxõ%]
nom=Daniel
```

**Remarque** :

si une chaîne vide est saisie (on appuie seulement sur la touche entrée), le nom contiendra une chaîne vide ( "" ) et la valeur de `lg` sera `0`.

# Passage à la ligne (1/2)

Après une saisie, la procédure `skip_line` permet de sauter tout ce que l'utilisateur a tapé sur le clavier jusqu'à la fin de la ligne.

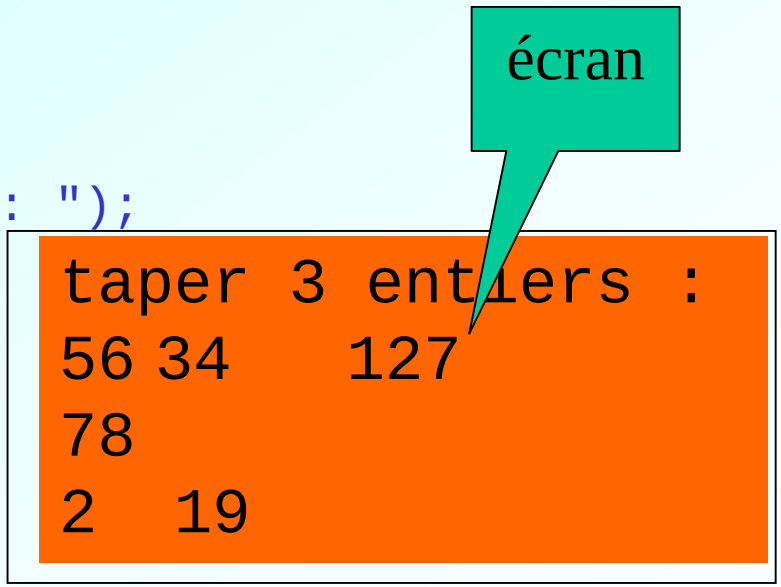
```
put( "Tapez votre nom : " );
get(nom);
skip_line;
put( "Tapez votre prénom : " );
get(prenom);
put("nom : ")put_line(nom);
put("prenom : ")put_line(prenom);
```

écran

```
Tapez votre nom : dupuy rené
Tapez votre prénom : rené albert
nom : dupuy
prenom : rené
```

# Passage à la ligne (2/2)

```
declare
   l,m,n:Integer;
begin
   put_line("taper 3 entiers : ");
   get(l);
   skip_line;
   get(m);
   skip_line;
   get(n);
end;
```




**résultat** : l=56, m=78, n=2

# Tableaux multidimensionnels

Le type des composants peut être quelconque, il peut donc être de type tableau.

On peut donc définir des tableaux de tableaux ou bien des tableaux à plusieurs dimensions.

# Tableaux de tableaux

Le type des composants est un tableau :

```
type M3 is array(1..3) of Vecteur(1..3);
mat_1:M3:=((3.5,4.0,1.0),(2.3,6.7,0.2),(1.2,8.9,0.0));
Ada.Float_Text_io.put(mat_1(2)(3));
-- affiche 0.2
```

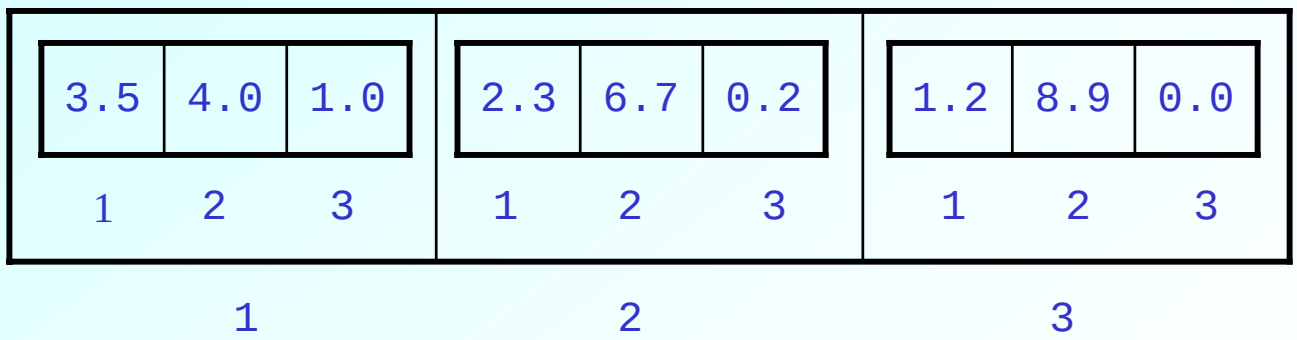



*tableau de tableau : mat_1*

# Tableaux à 2 dimensions

On indique les deux intervalles d'indices :

```
type M3_BIS is array(1..3,1..3)of Float;
mat_2:M3_BIS:=((3.5,4.0,1.0),(2.3,6.7,0.2),(1.2,8.9,0.0));
Ada.Float_Text_io.put.mat_2(2,3);
-- affiche 0.2
```

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 3.5 | 4.0 | 1.0 |
| 2 | 2.3 | 6.7 | 0.2 |
| 3 | 1.2 | 8.9 | 0.0 |

**Remarque** : Les agrégats sont les mêmes quels que soient le style de définition adopté.

# Tranche de tableaux de tableaux

`mat_1(2)(1..3), mat_1(1..2)(2)` en sont des exemples

```
declare
   type Vecteur is array(1..3) of Integer;
   type Matrice is array(1..3) of Vecteur;
   M:Matrice:=((6,7,8),(1,2,3),(4,5,6));
begin
   put( "M(1..2)(2)=" );
   for i in 1..2 loop
      put(M(i)(2),width=>2);
   end loop;
end;
```

**Résultat**

```
M(1..2)(2)=(7,2)
```

# Tranche de tableaux multidimensionnels

Il est illégal de définir des tranches de tableau multidimensionnels

```
mat_2(2,1..3) –- illégal
mat_2(1..2,2..3) -- illégal
```

# Attributs

Soit la déclaration :

```
tab:array (5..100,1..50) of Character;
```

| expression | valeur | signification |
|---|---|---|
| `tab'first(2)` | `1` | borne inférieure |
| `tab'last(1)` | `100` | borne supérieure |
| `tab'length(2)` | `50` | longueur de l'intervalle d'indices |
| `tab'range(1)` | `5..100` | l'intervalle d'indices |

Entre parenthèses, on précise la dimension concernée

# Convertibilité

Deux types tableaux sont convertibles s'ils ont :

1. même type d'éléments
2. même nombre de dimensions
3. mêmes types d'indices

# Exemple

```
type Tab1 is array(1..3) of Jour;
type Tab2 is array(5..7) of Jour;
mon_t1:Tab1:=(lundi,mardi,mercredi);
mon_t2:Tab2:=(vendredi,samedi,dimanche);
mon_t1:=Tab1(mon_t2);
mon_t2:=Tab2(mon_t1);
```

# Produit de matrices

Spécification formelle

Soit les matrices A(m,p), B(p,n)

=> C matrice telle que $C(i,j) = \sum_{k=1}^{k=p} A(i,k)*B(k,j)$

# Les déclarations

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Float_Text_io;use Ada.Float_Text_io;

procedure produit_matrice is
  m:constant Positive:= -- un entier Positif;
  n:constant Positive:= -- un entier Positif;
  p:constant Positive:= -- un entier Positif;
  type Matrice is array(Positive range <>,Positive range <>)
                                                  of Float;
  A:Matrice(1..m,1..p):= (....); -- agrégat d'intialisation
  B:Matrice(1..p,1..n):= (....); -- agrégat d'intialisation
  C:Matrice(1..m,1..n);
```

# Le calcul

```
begin
  -- calcul du produit des matrices A et B
  for i in A'range(1) loop
    for j in B'range(2) loop
      declare
        s:Float:=0.0;
      begin
        for k in 1..p loop
          s:=s+A(i,k)*B(k,j);
        end loop;
          C(i,j):=s;
      end;
    end loop;
  end loop;
```

# L'affichage

```
   -- affichage du résultat
    for i in A'range(1) loop
      for j in B'range(2) loop
        put(C(i,j),1,2,2);
        put( ", " );
      end loop;
      new_line;
    end loop;
end produit_matrice;
```

# Types composés : les articles

Les tableaux ne permettent pas d'associer des données de types différents. La notion d'article permet de rassembler plusieurs valeurs de types quelconques caractérisant une donnée particulière.

Les types Articles permettent de :

- Représenter des données complexes : outil de modélisation des données d'une application

- Manipuler globalement une donnée complexe tout en ayant accès à ses parties : possibilité de passer une donnée complexe en paramètre, de l'utiliser comme valeur de retour d'une fonction, de l'affecter à une variable.

# Exemple

Supposons que le type `Date` soit défini. Il associe les 3 valeurs : `jour, mois, an`

```
naissance: Date;
-- déclaration d'une variable
function quantieme(une_date:Date) return Natural;
-- utilisation en tant que paramètre d'une fonction
function fabrique_date(jj,mm,aa:Natural) return Date;
-- utilisation comme valeur résultat d'une fonction
```

# Définition d'un type article

Un type article Ada correspond à un produit cartésien d'ensembles différents.

**Construction syntaxique**

```
<définition_type_article>::=
record
  <ident_champ>:<type>[:=<expression>];
  {<ident_champ>:<type>[:=<expression>];}
end record;
```

Le type du champ d'un article peut être quelconque et optionnellement posséder une valeur par défaut.

# Déclaration d'un type article

**Construction syntaxique**

```
<déclaration_type_article>::=type <ident_type> is
   <définition_type_article>;
```

**Exemple**

```
type Date is
    record
      jour: Natural;
      mois: Natural;
      an: Natural;
    end record;
```

# Valeur d'un type article : agrégat

- Par association implicite champ-valeur

```
(val_1,val_2,...,val_n)
```

- Par association explicite champ-valeur

```
(champ1=>val_1,champ2=>val_2,....,champ n=>val_n)
```

- En panachant

```
(val_1,val_2,champ3=>val_3 ,....,champ n=>val_n)
```

# Exemple

```
naissance:Date:=(1,12,1985);
une_date:Date:=(jour=>28,mois=>03,an=>1989);
```

ou

```
une_date:Date:= (24,mois=>11,1989)
aujourd_hui:Date:=(mois=>11,an=>2002,jour=>14);
-- dans ce cas, on ne peut pas panacher
```

# Exemple

```
SMIC : constant Float := 1215.11;
type Employe is
   record
     nom:String(1..10);
     naissance:Date;
     salaire:Float:=SMIC;
   end record;

directeur:Employe:=("Patron   ",(5,8,1954),4512.23);
manu:Employe;
```

# Opération d'accès à un champ : .

**Syntaxe**

```
<opération_accès_champ>::=<ident_article>.<ident_champ>
```

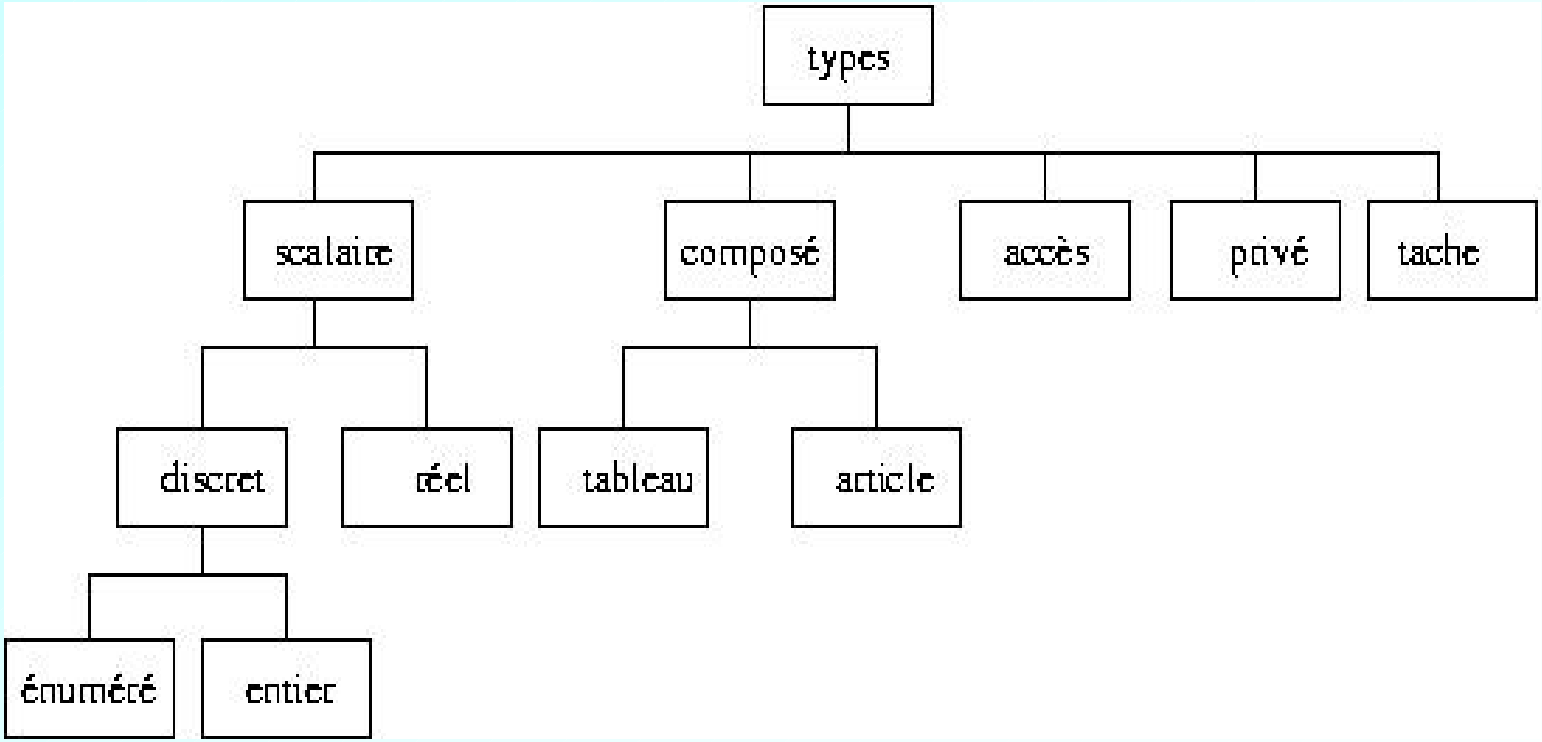
**Exemples**

```
directeur.naissance.an
-- représente l'année de naissance de directeur
Ada.Float_Text_io.put(directeur.salaire,1,2,2);
Ada.Float_Text_io.put(manu.salaire,1,2,2);
 -- affiche le salaire du directeur et de manu
```

résultat
```
4.51E+3
1.22E+3
```

```
if directeur.naissance.an<1900
then Ada.Text_io.put( "C'est l'heure de la retraite" );
else Ada.Text_io.put( "Toujours vaillant!!" );
end if;
```

# Classification des types Ada




Les types scalaires correspondent à des ensembles de valeurs atomiques (non composées).

Les types discrets sont des ensembles de valeurs ordonnées et énumérables
(on peut associer un entier à chacune des valeurs).