I.U.T. Aix
Département génie informatique

# Génie logiciel

**1er trimestre 2002/2003 (semaines 5 à 6)**

## Cours Ada (suite)

## Polycopié étudiants pour le cours deuxième partie (cours 6 à 9)

**Daniel Feneuille**

**Nom étudiant :**

# Cours 6 Ada les articles (2 heures)

## Thème : le type article.

Le type article (comme le type tableau) est un **type composite**. Un article est une structure de données qui possède des **composants** appelés **champs** ou éléments. Mais à la différence des tableaux, les articles (**`record`** en Ada) **peuvent contenir des éléments de types différents**. C'est là leur principal intérêt.

On peut distinguer la classification suivante :
- les types articles non paramétré (dits « simples »).
- les types articles paramétrés (en distinguant encore type article polymorphe et type article mutant).
- les types articles étiquetés (**`tagged`**) permettant les objets en Ada.

Nous examinerons successivement dans ce cours n°6 les deux premiers (non paramétrés puis paramétrés (avec discriminant). Les types étiquetés seront vus au moment de l'étude des objets et des classes (cours n°10).

## Les types articles « simples ».

### Description d'un type article (non paramétré).

Pour définir un type article il suffit **d'énumérer** ses différents composants, grâce à des **champs**, entre les mots réservés **`record`** et **`end record`**. Ces champs vont caractériser les éléments des objets instanciés de l'article.

Exemples (définition de deux types articles) :

```
type  T_UNE_PERSONNE  is
record
      NOM         :  STRING(1 .. 20); -- 3 champs
      PRENOM      :  STRING(1 .. 15);
      AGE         :  NATURAL  :=  0;
end record;

type  T_POINT_ECRAN    is
record
      X, Y        :  NATURAL;  -- attention : 4 champs
      VISIBLE     :  BOOLEAN := TRUE;
      COULEUR     :  T_COULEUR;
end record;
```

La déclaration des variables (ou **instanciation**) se fait de façon classique :

```
POINT_HAUT, POINT_BAS   :  T_POINT_ECRAN;-- deux points
PAPY, MOUGEOT     :  T_UNE_PERSONNE; -- deux personnes
```

### Les opérations associées aux types articles.

Un type article est un **type à affectation** (sauf si un champ est **`limited`**). Type à affectation signifie, rappelons-le, que : l'affectation, l'égalité et la non égalité sont définies **et c'est tout** a priori.
Exemples (d'utilisation globale) :

```
if POINT_HAUT = POINT_BAS
then .....
end if;

PAPY := MOUGEOT;
```

Comparaison globale et affectation globale

sont des instructions valides.

## Attribut associé aux articles (utile pour les articles paramétrés).

On retiendra l'attribut `CONSTRAINED` qui indique si l'objet instancié est polymorphe ou mutant. La réponse est évidemment de type booléen. Voir le fichier attribut1.doc (dans le CDRom) pour plus de détail et plus loin.

## Les valeurs par défaut.

Le type article est le **seul type dont les éléments peuvent posséder des valeurs initiales par défaut**. Ces valeurs sont seulement prises en compte si dans la déclaration de l'objet aucune initialisation n'est spécifiée. Dans l'exemple de type `T_UNE_PERSONNE`, le champ `AGE` a une valeur par défaut (ici la valeur nulle).
Un autre exemple (bien connu?) :

```
type    T_COMPLEXE  is
record
      RE    :  FLOAT := 0.0; -- partie réelle
      IM    :  FLOAT := 0.0; -- partie imaginaire
end  record;

Z    :    T_COMPLEXE;
```

la déclaration du complexe Z ci-dessus implique (par défaut) :

```
Z.RE  =  0.0    et  Z.IM  =  0.0
```

Autre déclaration :

```
B : T_COMPLEXE := (1.0,-1.0);  -- initialisation par agrégat!
```

## Accès à un champ.

L'accès aux composants d'un objet article se fait par la **notation pointée**. On sélectionne ainsi un composant de l'article. On ne **peut plus utiliser d'indice** comme avec les tableaux (évidemment !).
Exemple :

```
type  T_MOIS  is  (JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
            AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE);
type T_JOUR  is  range  1..31;
type T_ANNEE is  range 1_000..2_500 ;

type  T_DATE is
record
      JOUR  :  T_JOUR   := 31;
      MOIS  :  T_MOIS   := DECEMBRE;
      AN    :  T_ANNEE  := 2000;
end  record;

DEMAIN, AUJOURD_HUI, HIER  :  T_DATE;
```

Les trois dates sont initialisées au 31 DECEMBRE 2000. Dernier jour du siècle ! Notez le. Mais ceci est une vieille histoire !

```
DEMAIN.JOUR  :=  1;
DEMAIN.MOIS  :=  JANVIER;
DEMAIN.AN    :=  2001;
```

Qui équivaut à `DEMAIN := (1,JANVIER,2001); -- agrégat`

sont des instructions d'affectation des champs. Si on modifie l'objet en entier (c'est le cas ci-dessus en 3 instructions) on peut le faire en une seule instruction (globalement) avec un agrégat. Voir plus loin pour plus de détail.

De même avec :

```
X : T_COMPLEXE; -- a priori X = (0.0 , 0.0)
```

on peut écrire :

```
X.RE  :=  30.1 ;
X.IM  :=  25.7 ; -- ou X := (30.1,25.7) ;
```

Soit un type article avec des champs de type tableau (qui doivent être contraints) ou eux-mêmes de type article.

```
type   T_PERSONNE  is
record
  NOM                  :     STRING(1..10); -- tableau contraint
  PRENOM               :     STRING(1..10); -- tableau contraint
  AGE                  :     NATURAL  :=  0 ; -- valeur par défaut
  DATE_DE_NAISSANCE    :     T_DATE; -- article champ d'article
end  record;
```

Instanciation :

```
ANDRE, PIERRE  :  T_PERSONNE;
```

Les affectations suivantes :

```
ANDRE.NOM(2)                  :=  'A';
PIERRE.DATE_DE_NAISSANCE.MOIS :=  JUILLET;
DEMAIN.JOUR                   :=  T_JOUR'SUCC(AUJOURD_HUI.JOUR);
PIERRE.DATE_DE_NAISSANCE.JOUR :=  HIER.JOUR;
```

sont valides.

## Les agrégats d'articles.

L'agrégat permet de construire une valeur globale de type article, en donnant **une valeur à chaque champ** en une seule instruction (gain de temps et surtout grande lisibilité !). Les règles d'association sont identiques à celles des agrégats de tableaux (en particulier **tous les champs doivent être initialisés** ). Bonne révision !

- Association par position.

```
DEMAIN :=  (23, AVRIL, 1988);-- une date
ANDRE  :=  ("Durand    ","André     ",40,(30,JANVIER,1989));
X      :=  (30.1,25.7); --   un nombre complexe
```

- Association nominative.

```
DEMAIN :=  (AN  =>  1980, JOUR  =>  26, MOIS  => JUILLET);
ANDRE  :=  (DATE_DE_NAISSANCE  =>  (20,JUIN,1962), AGE  =>  39,
                NOM  => "Dupont    ",PRENOM  =>  "André     ");
```

- Association mixte (attention **ceci n'existait pas pour les agrégats de tableau**).

```
DEMAIN      :=  (23, AN => 1988, MOIS => DECEMBRE);
ANDRE       :=  ("Dupond    ","André     ",
                 DATE_DE_NAISSANCE  => AUJOURD_HUI, AGE => 20);
```

## Le type article comme paramètre d'un sous-programme.

**Exemple :** écriture d'une fonction qui effectue l'addition de deux complexes.

```
function ADD_COMPLEXE(Z1,Z2  :  in  T_COMPLEXE)  return  T_COMPLEXE  is
begin
      return  (Z1.RE + Z2.RE,  Z1.IM + Z2.IM);
end  ADD_COMPLEXE;
```

Utilisation de cette fonction :

```
Z := ADD_COMPLEXE  (COMPLEXE1, COMPLEXE2);
```

On peut également surcharger l'opérateur "+" ce qui est bien plus agréable.

```
function  "+"  (Z1, Z2  :  in  T_COMPLEXE)  return  T_COMPLEXE  is
begin
      return  (Z1.RE + Z2.RE  ,  Z1.IM + Z2.IM);
end;
```

L'utilisation peut se faire par :

```
Z  :=  "+" (COMPLEXE1,COMPLEXE2);
```

**ou mieux encore en utilisant pleinement la surcharge du ²+²** (opérateur binaire) :

```
Z  := COMPLEXE1 + COMPLEXE2; -- très lisible !
```

## Les types articles paramétrés (à partie variante)

Une déclaration de type article paramétré contient, après l'identificateur, une partie discriminante (c'est une liste de paramètres formels). De façon abstraite on a le schéma :

```
type  T_NOM(PARAMETRE : TYPE_DU_PARAMETRE{:= valeur_par_défaut}) is
      record
      --  partie fixe ici absente mais champs stables possibles;
           {case PARAMETRE  is
                 when  valeur_1  =>   -- champs_variants_1
                 when  valeur_2 =>    -- champs_variants_2
                 when  others   =>    -- autres champs
             end  case;}
      end  record;
```

La marque entre { et } indique une déclaration optionnelle. La **valeur par défaut du discriminant** « paramètre » sera vue plus loin ; il s'agira alors d'un type **article mutant** (plus précisément : pouvant muter).

Exemple :

```
type  T_SEXE  is  (MASCULIN, FEMININ, AUTRE); -- AUTRE ? no comment!

type  T_PERSONNE  (LE_SEXE  :  T_SEXE)  is
record
      NOM         :   STRING(1 .. 10);    -- partie fixe
      PRENOM      :   STRING(1 .. 10);
      AGE         :   NATURAL := 0;
      case  LE_SEXE  is       -- partie variante
            when MASCULIN  => CONSCRIT    : BOOLEAN ;
                              QUAND       : T_DATE;
            when FEMININ =>   NOM_DE_JEUNE_FILLE : STRING(1..10);
            when AUTRE =>     null;
      end  case;
end  record;
```

Le mot réservé **null** indique qu'il n'y a pas de champ associé à ce cas. Dans cet exemple il y a des champs "stables" (partie fixe) et une seule partie variante (case … end case). On verra des cas où il n'y aura pas de partie fixe et une partie variante, mais aussi (malgré un discriminant) pas de partie variante et une partie fixe !

**Un discriminant est un champ particulier accessible par la notation pointée en lecture seule (c'est-à-dire en consultation, aucune modification du discriminant d'un article polymorphe n'est possible).**

**Tout discriminant doit être de type discret.** La déclaration d'un objet de type article mutant se fait en associant une valeur a priori à un discriminant (voir plus loin bis !). C'est ce « signe » qui le différencie du type article polymorphe (ce dernier n'a pas de valeur initiale pour tout discriminant).

Exemples de déclaration d'objets d'article **polymorphe** (ils sont alors contraints) :

```
LUI   : T_PERSONNE (MASCULIN);
ELLE  : T_PERSONNE (FEMININ);
DRAG  : T_PERSONNE (AUTRE); -- no comment bis!
```

avec la variable `LUI` on a les « champs possibles » suivants :

- `LUI.LE_SEXE` (en consultation seulement avec **`if`**, **`case`** etc.. par exemple).
- `LUI.NOM, LUI.PRENOM, LUI.AGE` (en consultation et/ou en modification).
- `LUI.CONSCRIT, LUI.QUAND` (en consultation et/ou en modification).

avec la variable `ELLE` on a les « champs » suivants :

- `ELLE.LE_SEXE` (en consultation seulement).
- `ELLE.NOM, ELLE.PRENOM, ELLE.AGE` (en consultation et/ou en modification).
- `ELLE.NOM_DE_JEUNE_FILLE` (en consultation et/ou en modification).

avec la variable `DRAG` on a les « champs possibles » suivants :

- `DRAG.LE_SEXE` (en consultation seulement).
- `DRAG.NOM, DRAG.PRENOM, DRAG.AGE` (en consultation et/ou en modification).

Comme pour les types discrets, réels ou tableaux, il est possible de déclarer des sous-types de type article.
Par exemple :

```
subtype  T_MEC  is  T_PERSONNE (MASCULIN);

MACHO  :  T_MEC;
```

avec la variable `MACHO` on a évidemment les champs possibles suivants :

- `MACHO.LE_SEXE` (en consultation seulement rappel !).
- `MACHO.NOM, MACHO.PRENOM, MACHO.AGE` (en consultation et/ou en modification).
- `MACHO.CONSCRIT, MACHO.QUAND` (en consultation et/ou en modification).

## Utilisation du discriminant.

Le discriminant peut être utilisé de plusieurs façons :

- Pour établir une contrainte d'indice sur un composant de type tableau.

```
type  T_VECTEUR  is  array    (POSITIVE  range   <>)  of  FLOAT;
type  T_MATRICE  is  array    (INTEGER  range  <>,
                          INTEGER   range  <>)  of  FLOAT;
```

Il est possible de définir des types tableaux contraints qui peuvent « changer » de taille à l'instanciation. Dans ce cas là, il n'y a pas de champs variants. Pas de **`case`** mais .... un discriminant ! Attention !

```
type  T_MAT_CARREE  (COTE :  POSITIVE)  is
record
      LE_CARRE   :  T_MATRICE(1..COTE, 1..COTE);
end record;
```

```
type  T_TAB_REELS  (L : POSITIVE)  is
record
      VALEUR  :  T_VECTEUR(1..L);
end  record;

type  T_NOM (L : POSITIVE) is –- intéressant ? voire ! à revoir !
record
      LE_NOM : STRING (1..L);
end record;
```

Instanciation : on déclare des objets de ce type en fixant la taille effective :

```
TAB   :    T_TAB_REELS (10);  --    un tableau de 10 réels
MAT   :    T_MAT_CARREE (20); --    une matrice  20*20
NOM   :    T_NOM (20);        --    une chaîne de 20 caractères
```

- Pour choisir la valeur pour la partie variante de l'article.

```
type  T_REELS is  array  (POSITIVE  range  <>)  of  FLOAT;
type  T_ENTIERS is  array  (POSITIVE  range  <>)  of INTEGER;

type  T_VECT (REEL : BOOLEAN;L : POSITIVE) is -- il y a 2 discriminants!
record
      case    REEL  is
            when  FALSE  =>  V_ENT :  T_ENTIERS (1..L);
            when  TRUE   =>  V_REE :  T_REELS (1..L);
      end  case;
end  record;
```

Nous avons défini un type qui sera instancié soit par un tableau de `FLOAT`, soit par un tableau de `INTEGER`.

```
TAB_ENT :  T_VECT(FALSE,80);    -- 80 entiers
TAB_REL :  T_VECT(TRUE,80);     -- 80 réels
```

Mais cette utilisation **n'est pas « lisible** »! Que pensez vous de celle-ci :

```
type T_NATURE is (ENTIER, REEL);
type T_VECT (ELEM : T_NATURE ; L : POSITIVE) is
record
      case ELEM is
            when ENTIER => V_ENT : T_ENTIERS (1..L);
            when REEL   => V_REE : T_REELS (1..L);
      end case ;
end record;
```

et la déclaration :

```
TAB_ENT :  T_VECT(ENTIER,80);
TAB_REL :  T_VECT(REEL,80);
```

Un peu plus lisible? Oui ! Bien programmer c'est aussi être clair !

## Les types articles mutants (concept important) :

**Si on veut créer des objets non contraints (avec partie variante dynamique), c'est-à-dire des objets dont la structure n'est pas figée, la déclaration du type article avec discriminant(s) doit comporter une valeur par défaut pour un discriminant au moins. De cette façon l'objet n'est pas contraint par une valeur à sa déclaration. L'objet déclaré a la structure par défaut. Cette structure peut ensuite être modifiée, mais le changement de la valeur du discriminant ne pouvant se faire que par affectation globale (discriminant + champs).**

On peut reprendre l'exemple du type `T_PERSONNE` précédent (page 4). Le rendre **mutant** consiste à le déclarer ainsi :

Nouveau ! discriminant + affectation

```
type  T_SEXE  is  (MASCULIN, FEMININ, AUTRE);

type  T_PERSONNE  (LE_SEXE  :  T_SEXE := AUTRE)  is -- valeur par défaut
record
      NOM         :   STRING(1 .. 10);    -- partie fixe
      PRENOM      :   STRING(1 .. 10);
      AGE         :   NATURAL := 0;
      case  LE_SEXE  is       -- partie variante
            when MASCULIN  => CONSCRIT    : BOOLEAN ;
                              QUAND       : T_DATE;
            when FEMININ =>   NOM_DE_JEUNE_FILLE : STRING(1..10);
            when AUTRE =>     null;
      end  case;
end  record;
```

Il y a peu de différence : seule l'**initialisation** du discriminant `LE_SEXE` (ici à `AUTRE` mais **peu importe** !) marque la nature de **type mutant** (permettant des objets non contraints) pour ce nouveau type `T_PERSONNE`.

Une variable déclarée de type `T_PERSONNE` mais **sans contrainte** est par nature non figée donc non contrainte (elle peut changer en cours de programmation). On parlera de « **mutant** ». Ainsi :

```
HERMA : T_PERSONNE; -- pas de valeur entre parenthèse ⇒ non contraint
```

Une variable déclarée de type `T_PERSONNE` mais **avec une contrainte** est **définitivement typée** sans évolution possible. Exemples (on retrouve les mêmes que page 5) :

```
UN_MEC      : T_PERSONNE (MASCULIN);
UNE_FILLE   : T_PERSONNE (FEMININ);
DRAG        : T_PERSONNE (AUTRE);
```

Non mutants ! ⇒ contraints !

On ne peut pas écrire (évidemment) :

```
UN_MEC.NOM_DE_JEUNE_FILLE :=  "Augusta   ";
```

Pour ces trois instanciations il n'y a rien de changé par rapport à l'objet de type article contraint et c'est normal car on avait la possibilité de ne pas contraindre à la déclaration mais on a contraint quand même !

Par contre l'objet `HERMA` a été déclaré de «nature » `AUTRE` **a priori** (en l'absence de contrainte c'est l'initialisation par défaut). Il pourra changer de nature **en cours d'algorithme** ! Cette **mutation** ne pourra être qu'obtenue que par **affectation globale** (avec une variable contrainte ou non ou avec un agrégat).

```
HERMA := UN_MEC; -- HERMA devient masculin avec tous ces attributs[1]
ou
HERMA := (FEMININ, "Lovelace  ", "Ada       ",22,  "Byron     ");
-- devient féminin
```

[1] au propre comme au figuré ! (gag !).

**On ne peut pas changer seulement le discriminant sans changer** l'ensemble de la structure et c'est assez compréhensible. On peut par contre toujours consulter le discriminant pour savoir où on en est et, bien sûr, on peut changer isolément un champ particulier mais cohérent avec la structure !

```
HERMA.LE_SEXE := FEMININ; -- interdit (comme pour les contraints!)

case HERMA.LE_SEXE is – consulter c'est possible
      when MASCULIN     => ……..;
      when FEMININ      => …….;
      when AUTRE        => …….;
end case;
```

Si la structure du type **mutant** "contient" un tableau, sa taille pourra varier dynamiquement en cours d'algorithme. C'est le cas des `T_CHAINE` exemple assez intéressant (consultable en partie commune) et assez analogue aux chaînes de caractères variables comme les `Bounded_String`.

```
type  T_CHAINE  (L : NATURAL   :=  0)  is
record
          CHAINE : STRING (1..L);
end record;

EVOLU : T_CHAINE; -- objet article non contraint taille 0 a priori
FIGEE : T_CHAINE(20);-- objet article contraint (de taille 20)

EVOLU := FIGEE; -- EVOLU.CHAINE possède maintenant 20 éléments.
```

Ou encore

```
EVOLU := (5, "Merci");
```

Notez la différence !

puis

```
EVOLU := (0, ""); -- EVOLU redevient chaîne vide
```

Attention :

```
FIGEE := EVOLU; -- pas toujours possible
```

De façon générale il est **peu recommandé de mêler** objets de types contraints et objets de types non contraints.

**Remarques :**

Avec le paquetage `P_CHAINES` réalisant le type `T_CHAINE` le type est déclaré privé (voir les spécifications) et donc la structure n'est pas connue à l'utilisateur. Les exemples d'affectation par agrégat ci-dessus ne sont pas valables dans un programme utilisateur ! A revoir.

De très nombreux TD et TP vont permettre de se familiariser avec les articles mutants. (voir TD-TP polynôme)

## Types polymorphes, types mutables

Cette distinction et ce vocabulaire est une terminologie du groupe Ada-France. www.ada-France.org site à visiter pour des tas de bonnes raisons.

**Remarque** (qui sera plus compréhensible en avançant dans le cours) :

Un discriminant doit être de type discret (déjà dit !) mais aussi de type `access` anonyme et dans ce cas là il ne peut servir à définir des types mutables. A revoir donc quand on étudiera le type `access` (ou pointeur)

# Cours 7 Ada les paquetages (2 heures)

## Thème : les paquetages.

**Avertissement :** La notion de **paquetage** a déjà été, dans les premiers TD et TP (algorithmique ou programmation), utilisée avec plus ou moins de rigueur (on a surtout parlé de **ressources**). Il est venu le temps **d'approfondir** et de **structurer** cette notion. On verra d'abord des notions simples (quoique fortes !) puis ensuite les notions de **hiérarchie** de paquetages (père - fils) très **importantes** depuis la nouvelle norme.

**Introduction, généralités, etc. (bref tout ce qui faut savoir et retenir) :**

Dans les chapitres précédents nous avons écrit des sous-programmes (procédures et fonctions). Les procédures sans paramètres pouvant donner lieu à un programme exécutable (c'est-à-dire à un programme « appelable » à partir du système d'exploitation). Dans ces sous-programmes nous avons décrit des types, des variables, d'autres sous-programmes, qui sont nécessaires à notre sous-programme. Un certain nombre de ces descriptions reviennent fréquemment dans des applications différentes. Il serait donc intéressant de **disposer d'une méthode pour réutiliser ces déclarations déjà écrites**. La méthode consiste à regrouper un certain nombre de déclarations dans une **entité** (paquet ou module) et de travailler avec ce "paquet", à charge pour le **compilateur** de vérifier que le paquet existe, qu'il est à jour, que les déclarations que l'on utilise (on dit "importer") existent et qu'elles sont compatibles avec l'usage que l'on en fait. Cette **approche dépasse** le concept de **boîte à outils** que nous avons déjà utilisé en TP de programmation. On parle aussi de **module** (même si ce concept revêt encore des aspects plus compliqués que notre description sommaire).

Cette notion de « module » dans le langage Ada est supportée par le **paquetage**. Ces modules, **compilés séparément**, permettent à l'utilisateur de se **créer un environnement riche** en les accumulant dans des **bibliothèques**. Un paquetage est une collection d'entités et de « ressources » logiques. Il permet de regrouper dans une **même unité de programme** des déclarations : de constantes, de types, d'objets, de sous-programmes et le code d'initialisation de ces objets. Le lecteur habitué aux « unités » en Turbo-Pascal ne sera pas dépaysé (mais le paquetage Ada est bien plus qu'une unité du Turbo-Pascal). Le langage Java a repris le concept de paquetage (avec des nuances par rapport aux paquetages Ada). On verra ainsi un peu plus loin comment utiliser un paquetage pour réaliser un **type abstrait de données** (T.A.D.). Puis plus tard on passera à la programmation avec les objets « vrais » que permet la nouvelle version de Ada depuis 1995 (voir surtout le cours n°10).

Ada permet aussi **d'encapsuler** un ensemble d'informations et d'empêcher leur accès directement. Ce qui veut dire qu'il permet de **définir deux parties**, d'une part **spécification** donc « visibilité » avec l'extérieur et d'autre part **implémentation** (ou réalisation) donc détails **cachés et inaccessibles** à l'utilisateur.

**Description, structure, usage :**

Comme les sous-programmes, un paquetage comprend deux parties :

- La partie spécifications (déclaration ou profil).
- La partie corps ou **`body`** (implémentation ou réalisation).

G. BOOCH[1] dans sa méthode de conception orientée objet (C.O.O.) indique que la spécification sert à capter la sémantique statique de chaque objet ou classe d'objets et sert **d'interface** entre les "clients de l'objet" et l'objet lui-même. Ceci sera vu dans le cours n°10 « cours approfondi ». La spécification constitue **l'interface** entre le paquetage et l'environnement. Elle est vue comme un **contrat entre le réalisateur du paquetage et les utilisateurs** du paquetage. Le corps réalise la mise en œuvre des ressources définies dans la partie spécifications. Il permet d'implémenter chaque objet ou classes d'objets en faisant le choix d'une représentation adaptée.

Il est possible de distinguer deux usages des paquetages :

[1] Grady BOOCH : « Ingénierie du logiciel avec Ada » chez InterEdition (en bibliothèque).

- le paquetage est déclaré à l'intérieur de l'unité qui l'utilise. Le corps peut cependant être compilé séparément (indication **separate**). Cette **pratique est assez peu utilisée**.
- le paquetage est compilé séparément, et importé à l'aide de la clause **with**. Le paquetage est alors réutilisable dans un autre contexte (**cas le plus fréquent**). C'est celui que nous privilégions.

Exemple simple et connu [2]:

Supposons que vous avez déjà réalisé, dans un programme testé par ailleurs, un ensemble de sous-programmes permettant de travailler avec des nombres complexes. Il y a, d'une part la **déclaration** du type `T_COMPLEXE` puis la **définition** (déclaration et réalisation) des sous-programmes (comme +, -, etc.). La **réutilisation** de ce travail par un autre programme peut se faire par du copier coller (ah les bœufs !) ou évidemment plus intelligemment avec le **paquetage**. La première partie (spécifications) peut ressembler à cela :

```
package  P_COMPLEXE  is
type  T_NOMBRE_COMPLEXE  is -- définition du type et ses composants
record
     PARTIE_REELLE  :  FLOAT;
     PARTIE_IMAGINAIRE  :  FLOAT;
end record;
-- quelques outils ou opérateurs (on parle aussi de méthodes)
function  "+" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
function  "-" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
.....
end  P_COMPLEXE;
```

Ceci représente en quelque sorte un **contrat d'utilisation** (voir introduction). Ici la structure du type `T_NOMBRE_COMPLEXE` **n'est pas « cachée »** à l'utilisateur (à revoir). On rappelle qu'il est possible en Ada de déclarer des fonctions qui renvoient des variables de type composites (ici `T_NOMBRE_COMPLEXE`).

**Description d'un paquetage.**

Un paquetage contient un certain nombre de **déclarations** qui sont utiles à ses utilisateurs. Elles sont contenues dans la partie **spécifications**. On dit que **le paquetage les exporte**. Cette partie peut être compilée séparément de la partie suivante nommée la partie **réalisation** (corps ou **body**) du paquetage. Cette deuxième partie (le corps) contient le code des sous-programmes c'est à dire la façon de réaliser concrètement le problème : en fait, c'est ce qui est « caché » à l'utilisateur. Cette deuxième partie s'appelle le corps du paquetage ou **package body**. Dans notre exemple la partie réalisation (ou corps) ressemblerait à ceci :

```
package  body  P_COMPLEXE  is
function  "+"  (C1,C2  :  T_NOMBRE_COMPLEXE)  return T_NOMBRE_COMPLEXE is
begin
      return   (C1.PARTIE_REELLE + C2.PARTIE_REELLE,
            C1.PARTIE_IMAGINAIRE + C2.PARTIE_IMAGINAIRE);
end  "+";

function  "-"    (C1,C2  :  T_NOMBRE_COMPLEXE)  return T_NOMBRE_COMPLEXE is
begin
      return   (C1.PARTIE_REELLE - C2.PARTIE_REELLE,
            C1.PARTIE_IMAGINAIRE - C2.PARTIE_IMAGINAIRE);
end   "-";
.....
end  P_COMPLEXE;
```

[2] Le paquetage envisagé P_COMPLEXE existe déjà en Ada sous forme de deux paquetages que nous reverrons : `Generic_Complex_Types` et `Generic_Complex_Elementary_Functions`. Tout deux issus du paquetage `Ada.Numerics`. Pour le moment faisons comme s'ils n'existaient pas !

On peut bien sûr augmenter les «services » rendus par ce paquetage avec par exemple les traditionnelles opérations produit et quotient (il suffit d'appliquer les formules dans tous les livres de math de terminale !). On peut aussi ajouter les opérations élémentaires d'entrées - sorties (`LIRE` et `ECRIRE`). A faire ! ?

En résumé :

- **Partie déclaration (spécifications) :**

La forme générale d'une **spécification** de paquetage est la suivante :

{clause **`with`**} -- importation éventuelle d'un ou d'autres paquetages **nécessaires aux déclarations**
**`package`** NOM_DU_PAQUETAGE **`is`**

{ déclaration des types, constantes, variables, exceptions,
sous-programmes, sous-paquetages, tâches, etc. **à exporter** }

{ **`private`**
partie privée }
**`end`** NOM_DU_PAQUETAGE;

**Remarque** : la partie privée contient, entre autre, la description des types privés et limités privés exportés par le paquetage et déclarés avec « **`is private`** » dans les spécifications. Cette partie sera explicitée plus loin.

- **Partie corps ou réalisation (implémentation) :**

La partie **corps** a la forme suivante (notez bien le mot **`body`** en plus) :

{clause **`with`**} -- importation éventuelle d'un ou d'autres paquetages **nécessaires à l'implémentation**
**`package body`** NOM_DU_PAQUETAGE **`is`**

déclarations locales au corps du paquetage
réalisation complète des types
réalisation complète des S/P et sous-paquetages de la partie spécifications
{ **`begin`**
initialisation du corps (partie optionnelle) }
**`end`** NOM_DU_PAQUETAGE;

**Remarques** :

- Notez bien les deux clauses **`with`** respectives aux deux parties et **qui n'ont rien à voir entre elles**. Elles sont optionnelles. La première s'appuie sur des paquetages pour résoudre les problèmes liés aux **déclarations** tandis que la deuxième clause s'appuie sur des paquetages pour résoudre les **implémentations**.
- Le corps d'un paquetage "voit" (c'est-à-dire accède à) toutes les déclarations de la partie spécifications (**y compris les `with`** éventuels qu'il ne faut pas déclarer à nouveau).
- Le corps du paquetage est obligatoire si la déclaration du paquetage exporte des objets qui ont eux-mêmes un corps (S/P, tâches, sous paquetages). Si les spécifications n'exportent que des types, des constantes ou des renommages de sous-programmes le "**`body`**" ne s'impose évidemment pas.
- Les objets **déclarés dans un corps** de paquetage ne sont pas accessibles **par aucune autre unité**.
- Le principe : « séparation des spécifications, de la réalisation » existe déjà pour les sous programme (cours n°5) mais il est optionnel ! Tandis que c'est obligatoire pour les paquetages.
- Les spécifications et le corps d'un paquetage peuvent être **compilés séparément**. Pratique recommandée !

Avec cette dernière remarque le **corps** du paquetage (ou implémentation) peut être réalisé :
- longtemps après (sans retarder l'avancement du projet),
- par une autre équipe (qui respectera les spécifications),
- de différentes manières sans gêner l'utilisateur (car le contrat reste invariant).

**Utilisation d'un paquetage.**

Quelques rappels (déjà vus en TD-TP) :

- Pour rendre **visible** (donc **utilisable**) un paquetage par une unité de compilation utilisatrice (programme exécutable ou autre paquetage), on fait **précéder la déclaration de l'unité utilisatrice** d'une clause **`with`** qui **évoque** le paquetage à utiliser.
- Tout objet utilisé à l'extérieur d'un paquetage qui le définit doit être **préfixé** par le nom du paquetage. L'usage de la clause **`use`** simplifie les écritures (en n'obligeant pas le préfixage) mais, dans certains cas, génère des ambiguïtés possibles (à revoir).

Exemple d'utilisation de notre paquetage :

```
with P_COMPLEXE;

procedure TEST_COMPLEXE is
use  COMPLEXE;
Z, X : T_NOMBRE_COMPLEXE;
I  :  T_NOMBRE_COMPLEXE := (0.0,1.0); -- initialisation par agrégat
begin
      X := (1.0, 0.5);
      Z := X + I;
      -- il faudrait éditer le résultat!
end TEST_COMPLEXE;
```

L'utilisation d'un bloc pour limiter la portée d'un **`use`** est recommandée (nous en reparlerons).

## Paquetage et types privés.

Les types en Ada sont divisés en deux catégories :

- d'une part les types dont la **structure est connue** des utilisateurs (types énumératifs, types tableaux etc. sans oublier peut-être les types prédéfinis !). Ils sont dits publiques.
- d'autre part les types dits privés (ou fonctionnels) dont la **structure doit rester cachée (i.e. inaccessible)**.

**Déclaration de type privé.**

La déclaration d'un type privé doit **nécessairement** se trouver dans la partie **spécifications** d'un paquetage. La syntaxe de cette déclaration est :

```
type  T_NOM_DU_TYPE  is  private;
```

Comme nous l'avons vu précédemment, il est possible, dans la partie déclaration, de déclarer : des constantes, des sous-programmes, etc. D'où les constantes d'un type privé déjà déclaré :

```
C_DU_TYPE  :  constant  T_NOM_DU_TYPE;
```

et les sous-programmes portant sur les types privés :

```
function  MA_FONC (LISTE_DE_PARAMETRES) return T_NOM_DU_TYPE;
```

Exemple (reprenons « nos » complexes) :
La « visibilité » de la structure permettant d'implémenter nos complexes (avec partie réelle et partie imaginaire) **n'est pas nécessaire pour l'utilisateur**. Que lui importe de savoir si les complexes sont réalisés avec cette propriété plutôt qu'avec celle de module et argument par exemple !

D'où :

```
package  P_COMPLEXE  is

type  T_NOMBRE_COMPLEXE  is private;

function  "+" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
function  "-" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
-- d'autres opérateurs à faire!
procedure LIRE (C : out T_ NOMBRE_COMPLEXE);
procedure ECRIRE (C : in T_ NOMBRE_COMPLEXE);
```

Partie « visible » donc utilisable.

```
private
     type  T_NOMBRE_COMPLEXE  is
     record
          PARTIE_REELLE  :  FLOAT;
          PARTIE_IMAGINAIRE  :  FLOAT;
     end record;

end  P_COMPLEXE;
```

Partie « cachée » i.e. **inaccessible** pour l'utilisateur

**Remarques** (voir aussi le synoptique page 9 plus loin) **:**

- La « privatisation » de la structure réalisant le nombre complexe « cache » certes la **visibilité** (mais pas la lisibilité !) de l'objet mais en conséquence (et surtout) **empêche l'utilisateur d'accéder** à la structure. Aussi doit-on absolument fournir des outils pour (par exemple) construire (`LIRE`) et voir (`ECRIRE`) ces objets. Plus tard on verra aussi les notions d'accesseurs et de modifieurs.
- Pour que le compilateur puisse réaliser les variables du type privé il faudra bien, à un moment donné, décrire effectivement la structure de ce type privé. **Cette structure sera décrite dans la partie privée du paquetage.** Elle est lisible mais ... pas "visible" c'est-à-dire inaccessible à l'utilisateur du paquetage !
- La réalisation de notre nouveau paquetage `P_COMPLEXE` (c'est-à-dire le corps) restera la même que lors de la première réalisation (voir le corps pages précédentes) car la **structure « cachée »** à l'utilisateur (invariante !) est **accessible**, elle, au **concepteur** (ou réalisateur) du **paquetage**.

**Utilisation du type « privé » et des méthodes associées.**

A **l'extérieur** du paquetage et grâce à la clause **`with`** le **type devient utilisable** et il sera possible de déclarer des variables et utiliser les sous-programmes référençant le type privé. On voit donc que le paquetage **exporte donc non seulement** une structure d'objet **mais aussi un ensemble d'entités** qui lui sont connexes (sous-programmes par exemple). Ceci est la **base** de ce que l'on nomme le **type abstrait de données** (T.A.D.) nous y reviendrons (cours n°10).

**Opérations sur les types privés.**

Les seules opérations prédéfinies sur les types privés **sont l'affectation et la comparaison**. Un type privé n'autorisera donc que les opérations suivantes (rappel) :

- affectation d'une variable de ce type : `VAR := expression_du_type;`
- test d'égalité : `VAR1 = VAR2;`
- test d'inégalité : `VAR1 /= VAR2;`

Exemples :

On verra avec intérêt les T.A.D. suivants :

- « T_Rationnel » c'est l'objet du TD-TP Ada n°11 !
- « T_Chaine » dans le fichier chaine.doc (CDRom) à lire !

Un autre exemple[3] de type de données abstraites : le type « ensemble » de ..... (type discret)

```
package  P_ENSEMBLE_DE_.......  is
      type T_ENSEMBLE  is private;

      ENSEMBLE_VIDE : constant T_ENSEMBLE;
      function  "&" (E : in T_ENSEMBLE; ELEM : in T_DISCRET)
            return T_ENSEMBLE; -- ajoute un élément à l'ensemble;
      function "+"(E,F : in T_ENSEMBLE ) return T_ENSEMBLE; -- union
      function "*"(E,F : in T_ENSEMBLE ) return T_ENSEMBLE; -- intersection
      function APPARTIENT (ELEM : in T_DISCRET; E : in T_ENSEMBLE)
            return BOOLEAN; -- la surcharge avec "in² n'est pas possible
      private
            type  T_ENSEMBLE is array (T_DISCRET) of BOOLEAN;
            ENSEMBLE_VIDE : constant  T_ENSEMBLE := (others => FALSE);
end   P_ENSEMBLE_DE_......;
```

La présentation est générale (la réalisation peut d'ailleurs être «générique » : concept à revoir). Le type `T_DISCRET` est soit un type prédéfini discret (`CHARACTER` par exemple), soit énumératif, soit un sous type d'un type prédéfini (par exemple un sous type de `INTEGER`).

La réalisation peut simplement être réalisée ainsi :

```
package body  P_ENSEMBLE_DE_....... is
      function  "&" (E : in T_ENSEMBLE; ELEM : in T_DISCRET)
            return T_ENSEMBLE is
      begin
            return E(T_DISCRET'FIRST..T_DISCRET'PRED(ELEM)) & TRUE
            & E(T_DISCRET'SUCC(ELEM)..T_DISCRET'LAST);
            exception --..... à revoir
      end  "&";
      function "+"(E,F : in T_ENSEMBLE ) return T_ENSEMBLE is
      begin
            return  E  or  F;
      end  "+";
      function "*"(E,F : in T_ENSEMBLE ) return T_ENSEMBLE is
      begin
            return  E  and  F;
      end  "*";
      function APPARTIENT (ELEM : in T_DISCRET; E : in T_ENSEMBLE)
            return BOOLEAN is
      begin
            return  E(ELEM);
      end  APPARTIENT;
end   P_ENSEMBLE_DE_........;
```

L'**utilisation** (à finir et à mettre en œuvre) a l'allure suivante:

```
with  P_ENSEMBLE_DE_..........;

procedure  TEST_ENSEMBLE  is
      use  P_ENSEMBLE_DE_.......;
begin
      ..........
      à imaginer!
      ..........
end  TEST_ENSEMBLE;
```

[3] Fort classique qui « traîne » dans tous les bons ouvrages ! A connaître !

## Les types privés limités.

Les types privés **limités** sont des types privés dont l'utilisation est encore **plus restreinte**. Ils **ne possèdent plus ni la comparaison ni l'affectation**. La seule chose que l'on puisse faire avec un type privé limité, c'est de déclarer des variables et utiliser les sous-programmes fournis par le paquetage (sans affectation ni comparaison). Ces techniques permettent un **contrôle total** sur leur utilisation.

Un type privé limité se déclare de la façon suivante :

```
type  T_EXEMPLE  is  limited  private ;
```

Un exemple « classique » de type privé limité est le **type fichier**. Il est possible de déclarer des variables de type fichier, il est possible de réaliser certaines opérations (ouverture, fermeture, lecture etc.), mais la comparaison ou l'affectation de deux fichiers sont interdites (cela aurait-il d'ailleurs un sens ?). A revoir avec l'étude approfondie de `ADA.TEXT_IO`, `ADA.SEQUENTIAL_IO`, `ADA.DIRECT_IO` et `ADA.STREAM_IO` dans les TD-TP fichiers à venir. On verra aussi en TD TP n°10 l'exemple très classique de la réalisation d'une pile (type abstrait de données limité privé par excellence).

**Remarque :** Une bonne pratique de programmation consiste à n'exporter qu'un **unique** type privé (limité ou non) par paquetage avec les **opérations** applicables sur ce type. (C'est la notion de **type abstrait de données** et on se rapproche alors du concept **d'objet plus général encore**).

## Différents « emplacements » d'un paquetage :

- **Paquetage en tant qu'unité de compilation (le plus utilisé).**

```
package  NOM_DE_PAQUETAGE  is
      --  les spécifications
end  NOM_DE_PAQUETAGE;
```

Puis dans la « foulée » ou ailleurs (c'est mieux!), mais **séparément**, le corps du paquetage.

```
package body  NOM_DE_PAQUETAGE  is
      --  implémentation des spécifications
end  NOM_DE_PAQUETAGE;
```

- **Un paquetage peut être déclaré dans un bloc.**

```
      NOM_DU_BLOC :
      declare
            package  P_TRUC is
            ……--  spécifications
            end P_TRUC;
            package body P_TRUC is
            ……--  corps
            end P_TRUC;
      begin  -- début de portée de P_TRUC
            ……
            P_TRUC.OUTIL (X); -- on utilise P_TRUC
            ……
      end NOM_DU_BLOC;  -- fin de portée de P_TRUC
```

- **Un paquetage peut être déclaré dans une spécification de S/P.**

```
  procedure  NOM_DE_PROCEDURE  is
            package  NOM_DE_PAQUETAGE  is
            --   interface (partie visible par l'utilisateur)
            end  NOM_DE_PAQUETAGE;

            package  body  NOM_DE_PAQUETAGE  is
            --   implementation   (partie cachée)
            end  NOM_DE_PAQUETAGE;
      begin --   corps de procédure
       --  utilisation des fonctions et procédures du paquetage
      end  NOM_DE_PROCEDURE;
```

**Un paquetage peut être déclaré dans une spécification ou un corps d'un autre paquetage.**

```
package  NOM_DE_PAQUETAGE  is
      package  P_EXPORT  is
          --   spécifications
      end  P_EXPORT;
end  NOM_DE_PAQUETAGE;
```

Ce cas est assez utilisé voir notamment le paquetage `Ada.Text_Io` et ses 6 sous paquetages.
Ou encore :

```
package  body  NOM_DE_PAQUETAGE  is
     --   déclarations locales

       package  LOCAL  is
              - -  spécifications
       end  LOCAL;

       package  body  LOCAL  is
       --  implémentation locales
       end  LOCAL;
end  NOM_DE_PAQUETAGE;
```

## Introduction à la dérivation (notion très forte reprise au cours n°10).

De façon générale : si `T_PERE` est un type quelconque, alors, avec la déclaration :

```
type T_NOUVEAU is new T_PERE;
```

On a dérivé le type `T_PERE` et `T_NOUVEAU` est un **nouveau** type dit **dérivé** de `T_PERE` ; `T_PERE` est dit le type père de `T_NOUVEAU` (notez bien le **new**).

Un type dérivé appartient à la même classe de types que son père (si `T_PERE` est un type article alors `T_NOUVEAU` sera un type article). L'ensemble des valeurs d'un type dérivé **est une copie** de l'ensemble des valeurs du type père (sauf s'il y a un **range,** en plus, qui limitera l'intervalle des valeurs héritées). **Mais on a quand même des types différents** et les valeurs d'un type ne peuvent pas être affectées à des objets de l'autre type. Cependant la conversion entre les deux types est possible.

Soit :

```
LE_NEW      : T_NOUVEAU;
LE_VIEUX    : T_PERE;
```

alors : `LE_NEW := LE_VIEUX; -- interdit`
mais : `LE_NEW := T_NOUVEAU(LE_VIEUX); -- possible (conversion)`

**Règles d'héritage concernant les types dérivés.**

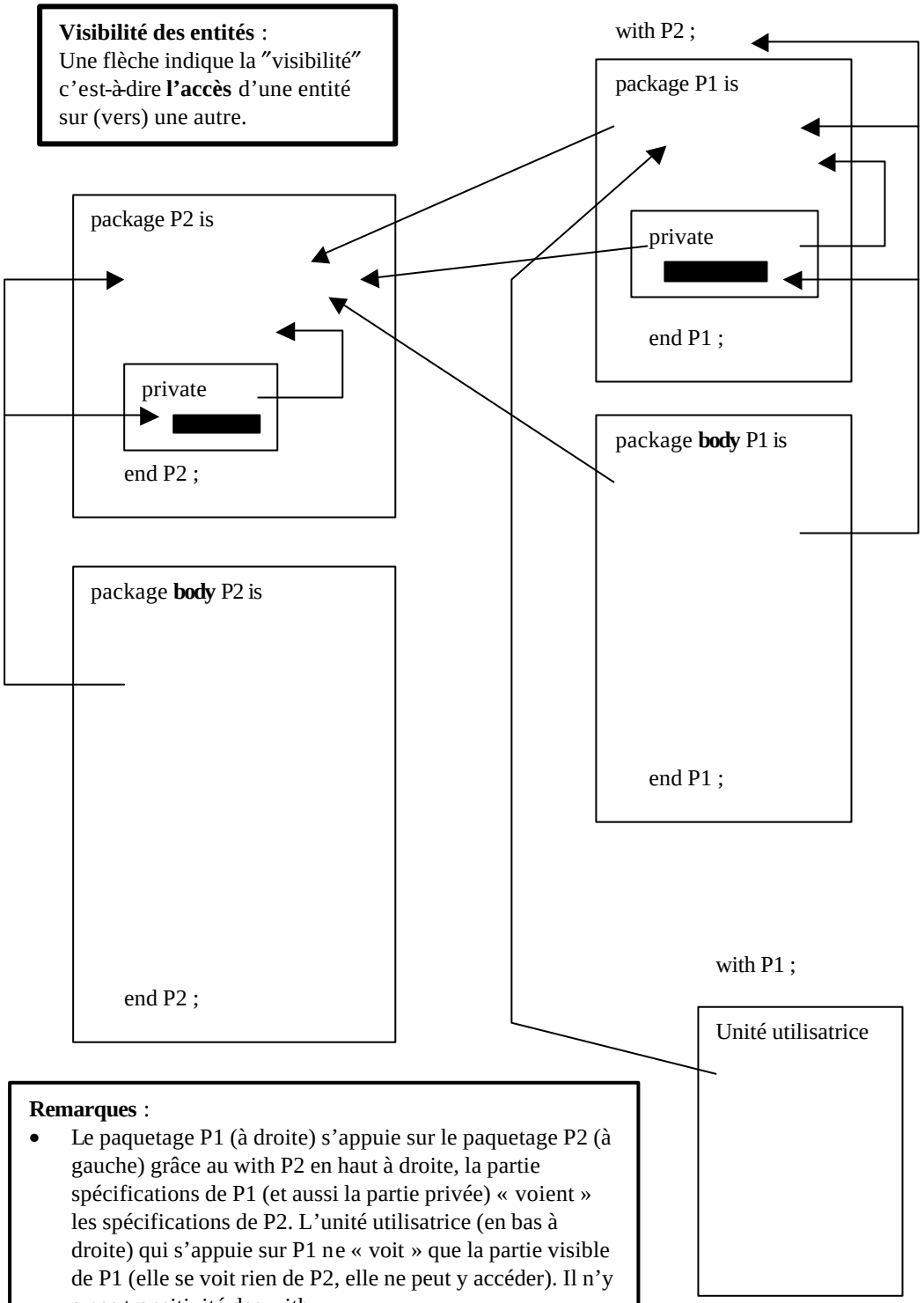
Les opérations applicables aux types dérivés sont les suivantes :

- Un type dérivé possède les mêmes attributs (s'ils existent) que le type père.
- L'affectation, l'égalité et l'inégalité sont également applicables sauf si le type père est limité.
- Un type dérivé héritera des sous-programmes (S/P) applicables au type père.

**Quand le type père est un type prédéfini** : les S/P hérités se réduisent aux S/P prédéfinis, et si le type père est lui même un type dérivé alors les S/P hérités seront de nouveau transmis.

**Quand le type père est déclaré dans la partie visible d'un paquetage** : tout S/P déclaré dans cette partie visible sera hérité par un type dérivé (le fils) **dès qu'il est déclaré**.

**Visibilité des entités** :
Une flèche indique la "visibilité" c'est-à-dire **l'accès** d'une entité sur (vers) une autre.




**Remarques** :

- Le paquetage P1 (à droite) s'appuie sur le paquetage P2 (à gauche) grâce au with P2 en haut à droite, la partie spécifications de P1 (et aussi la partie privée) « voient » les spécifications de P2. L'unité utilisatrice (en bas à droite) qui s'appuie sur P1 ne « voit » que la partie visible de P1 (elle se voit rien de P2, elle ne peut y accéder). Il n'y a pas transitivité des with.
- Le corps de P1 « voit » aussi la partie spécifications de P2 grâce au with P2 déclaré avec les spécifications de P1. (mais le corps de P1 ne voit pas la partie privée de P2)
- Les corps des paquetages (body) ont accès à leurs spécifications respectives ainsi qu'à leur partie privée.
- Les parties privées ont accès à leurs spécifications.
- Aucune entité n'a accès aux corps des paquetages.

## Le surnommage (renames).

Cette déclaration s'applique **généralement** aux sous programmes. Elle permet, dans la partie spécifications d'un paquetage notamment, de ne pas définir, dans le corps du paquetage correspondant, la réalisation du sous programme déclaré. Voir par exemple dans P_E_SORTIE (cours n°1 généralités III) :

```
procedure ECRIRE (CARAC : CHARACTER) renames ADA.TEXT_IO.PUT ;
```

La procédure ECRIRE est déclarée comme étant **la même** que la procédure PUT du paquetage ADA.TEXT_IO. Ceci nous dispense d'avoir à réaliser, dans le body de P_E_SORTIE, le corps de la procédure ECRIRE.

Cette déclaration **renames** est utile si l'on souhaite faire « remonter » à une unité utilisatrice d'un paquetage des ressources inaccessibles par transitivité d'un autre paquetage. Voir la page précédente où l'unité utilisatrice s'appuie sur P1 qui s'appuie lui même sur P2 et l'unité utilisatrice n'a pas accès à P2 ! La partie spécifications de P1 peut offrir à ses utilisateurs quelques ressources de P2 en les surnommant dans ses spécifications :

```
with P2 ;
package P1 is
…….
      procedure COUPER (paramètres formels) renames P2.COUPER ;
…….
end P1 ;
```

On voit que l'utilisateur de P1 a accès à COUPER de P1 qui n'est rien d'autre que le COUPER de P2. On remarque aussi que l'on n'est pas obligé de changer de nom comme nous l'avions fait avec ECRIRE et PUT.

On peut **surnommer aussi** : des unités de bibliothèque, des variables (commodes avec les champs des articles et les tranches de tableaux), des constantes et des exceptions (cours n°8). On ne peut pas surnommer un type ! Si l'on souhaite faire «remonter» un type, on utilise, toujours dans la partie spécifications, la technique de définition d'un sous type identique à un type tel que : `subtype T_JOUR is P2.T_JOUR ;`

## Conception orientée objet. (culturel : à lire pour le moment et à approfondir au cours n°10 !)

Les objets manipulés par un programme peuvent être très divers (nombres, tables, dictionnaires, texte, etc.). Ils peuvent être aussi très complexes. Nous avons vu :

- qu'un objet possède une valeur et une seule à un moment donné,
- que l'exécution d'une action tend à modifier un ou des objets (variables),
- que des fonctions et des opérateurs permettent de produire des valeurs à partir d'autres valeurs,
- que c'est le type de l'objet qui détermine l'ensemble des valeurs que peut prendre un objet, les actions applicables sur l'objet, ainsi que les fonctions opérant sur les valeurs.

En Ada les paquetages sont le support de la notion de "conception orientée objet". En effet les paquetages exportent des types d'objets ainsi que les actions applicables sur ces objets (c'est le concept de type abstrait de données). Ainsi Ada permet de définir des types abstraits de données par les notions de paquetage, de type privé et de compilation séparée. La notion de type abstrait de données permet de décrire un type en séparant la description du type et sa réalisation. Le type abstrait de données constitue la base théorique du concept de classe dans les langages orientés objets. Il y manque la dérivation avec héritage et l'évolution de la structure de données de base. Ceci est une autre histoire que nous verrons plus tard avec « l'objet » cours n°10.

**Exercice :**
Sur le modèle de la réalisation du type abstrait de données T_COMPLEXE (qui est à finir !) on peut réaliser le paquetage P_RATIONNELS en utilisant un type article pour la déclaration complète du type rationnel (en privé) ainsi que les classiques opérateurs (+,-,*,/). Sans oublier la séparation entre la spécification du type et sa réalisation. Voir le TD TP n°11. Voir aussi l'exemple (fichier chaine.doc dans le CDRom) du type T_CHAINE dont nous avons tant parlé. Ces exemples sont génériques et seront mieux compris après le cours n°9.

## Les bibliothèques hiérarchiques (fils publics, fils privés).

Cette partie est nouvelle (Ada 95) mais a déjà été utilisée en algorithmique avec le paquetage `P_MOUCHERON` qui était le (grand)père des autres paquetages `MOUCHERON1`, … Ce rappel pour bien fixer les idées !

**Position du problème.**

Avant d'aborder les objets et les classes (un peu de patience cours n°10) nous allons montrer une extension apportée au langage Ada pour le développement des grosses applications. En effet, si la distinction « spécifications/corps » donne de bons résultats pour des applications « modestes », cette propriété a montré ses limites dans le développement d'applications conséquentes (volumineuses) et incrémentales.

A priori quand on cherche à **étendre des fonctionnalités** (par exemple fournir quelques sous programmes supplémentaires) à un type abstrait de données (TAD) déjà défini dans un paquetage on peut :

- ajouter les nouvelles fonctionnalités dans la partie (initiale) des spécifications puis les réaliser dans le corps élargi. Ce faisant, on **oblige toutes les applications courantes ou passées** n'utilisant pas ces fonctionnalités à procéder malgré tout **à des (re)compilations fastidieuses** (la partie spécifications ayant changé). On arrive ainsi à des **paquetages trop importants à gérer**.
- Recréer un paquetage « clone » renommant les fonctionnalités premières et faisant remonter tous les types, exceptions, variables et constantes du premier paquetage. Ceci demande aussi à ce que le T.A.D ne soit pas (ou ne soit plus) privé **ce qui est à l'opposé de la saine notion d'abstraction**.
- Dériver le type premier dans un autre paquetage puis ajouter les fonctionnalités mais alors on a créé deux types différents obligeant l'utilisateur des deux « versions » à des conversions insupportables.

Mais tout ceci est extrêmement archaïque !

**Exemple** : On veut ajouter des fonctionnalités au type `T_COMPLEXE` pour certaines applications spécifiques plus gourmandes tout en gardant «l'intégralité et l'intégrité » des anciennes spécifications qui donnent satisfaction dans la plupart des applications anciennes, et qui, elles, ne souhaitent pas utiliser les nouvelles fonctionnalités.

**Solution.**

Il serait souhaitable de pouvoir étendre un T.A.D (représenté par un paquetage) **dans ses fonctionnalités** grâce à un paquetage **hérité (ou fils)** n'obligeant à aucune recompilation du paquetage **père** tout en permettant l'accès à la structure privée. Cette extension va être illustrée sur notre exemple de `T_COMPLEXE`.

Nous avons déjà notre paquetage père concrétisant le T.A.D (cf. page 5) :

```
package P_COMPLEXE is

 type T_COMPLEXE is private;

 function  "+" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
 function  "-" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
 function  "*" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
 function  "/" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
 procedure LIRE (C : out T_ NOMBRE_COMPLEXE);
 procedure ECRIRE (C : in T_ NOMBRE_COMPLEXE);

private
........ définition du type voir page 5
end P_COMPLEXE;
```

Nous allons « augmenter » les fonctionnalités de ce paquetage en **créant un « fils »** ainsi :

```
package P_COMPLEXE.ELARGI is
```

Notez le point

```
function Conjuguée (X : T_Complexe) return T_Complexe;
function Argument  (X : T_Complexe) return Float;
function Module    (X : T_Complexe) return Float;

.......etc.
end P_COMPLEXE.ELARGI;
```

On réalise ensuite le corps de ce paquetage comme d'habitude avec compilation séparée.

```
package body P_COMPLEXE.ELARGI is
```

Notez encore le point

```
function Conjuguée (X : T_Complexe) return T_Complex is
begin
........
end Conjuguée;

function Argument (X  : T_Complexe) return Float is
begin
.......
end Argument;
.......
end P_COMPLEXE.ELARGI;
```

Le paquetage `P_COMPLEXE.ELARGI` dénote le paquetage « fils » du paquetage `P_COMPLEXE` (cette notion est syntaxiquement illustrée par le préfixage en notation : « `père.fils` »). Il est évident (mais prudent de le signaler) que le **corps du paquetage fils a la visibilité sur la partie spécifications (publique et privée) du paquetage père** ! Sinon comment pourrait-il réaliser les fonctionnalités supplémentaires s'il ne peut accéder à la structure privée ? A noter aussi que si le fils possède une partie privée alors cette partie privée a aussi visibilité sur la partie privée du père. Bien sûr, la **partie visible du fils ne peut avoir accès à la partie privée du père**. Voir le nouveau synoptique ci-après.

Cette technique (père-fils) s'utilise surtout, non pas pour étendre, a posteriori, des fonctionnalités, mais pour construire, **a priori**, des **modules hiérarchiques** offrant des fonctionnalités allant des plus simples aux plus compliquées car la descendance peut être « infinie » !

**Utilisation par un « client » d'une ressource** (où `with` et `use` se différencient !) **:**

avec la clause **`with`** évidemment (et éventuellement avec la clause **`use`** en plus). Ainsi :
**`with`** `P_COMPLEXE.ELARGI;` permet, à une procédure client par exemple, et **en une seule déclaration**, d'utiliser les **deux** ressources, à savoir : le père : `P_COMPLEXE` et aussi le fils : `P_COMPLEXE.ELARGI`.
Cependant il faut préfixer : `P_COMPLEXE."+"` ou `P_COMPLEXE.ELARGI.Argument` respectivement soit avec l'identificateur du père soit avec celui du fils. La clause **`use`** permet soit de ne pas préfixer le père en écrivant **`use`** `P_COMPLEXE;` soit de ne pas préfixer le fils avec **`use`** `P_COMPLEXE.ELARGI;` En aucun cas on ne peut écrire **`use`** `ELARGI` même après avoir écrit **`use`** `P_COMPLEXE`. Il faut deux **`use`**.

Peut-on **ne recourir qu'à** un seul paquetage fils sans recourir à son père ? (puisque **`with`** `P_COMPLEXE.ELARGI` à valeur de : **`with`** `P_COMPLEXE, P_COMPLEXE.ELARGI;`). Oui, c'est possible, grâce à la possibilité qu'offre Ada de compiler un surnom. Soit les deux lignes :

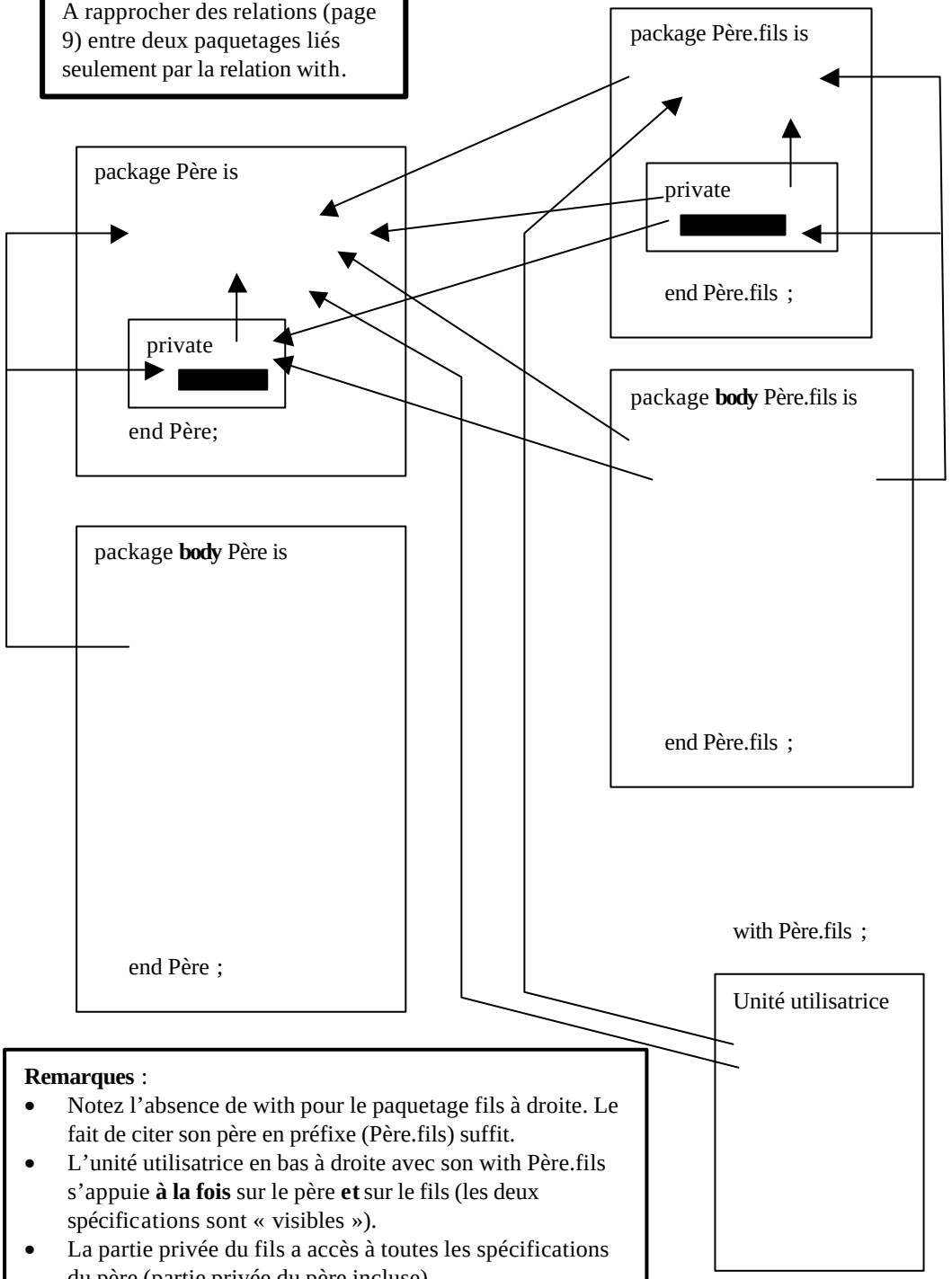
```
with P_COMPLEXE.ELARGI;
package P_COMPLEXE_ETENDU renames P_COMPLEXE.ELARGI;
..........
```

et avec :

```
with P_COMPLEXE_ETENDU;
```

l'utilisateur ne peut qu'accéder à `P_COMPLEXE_ETENDU` donc seulement au fils de `P_COMPLEXE`.

**Visibilité des entités** (Père fils).
A rapprocher des relations (page 9) entre deux paquetages liés seulement par la relation with.




**Remarques** :

- Notez l'absence de with pour le paquetage fils à droite. Le fait de citer son père en préfixe (Père.fils) suffit.
- L'unité utilisatrice en bas à droite avec son with Père.fils s'appuie **à la fois** sur le père **et** sur le fils (les deux spécifications sont « visibles »).
- La partie privée du fils a accès à toutes les spécifications du père (partie privée du père incluse).
- Le corps du fils a accès à toutes les spécifications du père (partie privée incluse).
- Les parties privées (ainsi que les corps) ont toujours accès à leurs spécifications !
- Aucune entité n'a accès aux corps des paquetages. Ce principe n'a pas changé !

Un paquetage père peut naturellement avoir **plusieurs fils** (ils sont alors **frères** !). On possède là une solution pour le **partage d'un type privé** entre plusieurs unités de compilation. En combinant cette approche d'extension de fonctionnalités avec celle d'extension de structure (voir cours n°10 : les types étiquetés) on possède un moyen de fournir des vues différentes d'un type où certains composants seront visibles à un utilisateur tandis que d'autres lui seront cachées. A voir prochainement.

**Remarques :**

Un fils peut avoir aussi une **partie privée** (on l'a dit) qui ne sera visible **que de ses propres descendants** (fils, petits-fils) mais **ni de son père ni de ses frères**.
Un **corps** de fils dépend de ses ascendants (père, grand-père etc.) donc n'a pas besoin de clause `with`.
Un **corps** de père peut avoir accès (avec `with`) à son fils ! (partie spécifications seulement) voir, comme exemple, les sources du reformateur de gnatform. De même un **corps** de fils peut aussi (avec `with`) avoir accès aux spécifications de son frère ! Notez bien : seulement le **corps** !
Le concept de hiérarchie (père-fils) ne s'applique pas qu'à l'unité de compilation qu'est le paquetage mais aussi aux procédures. Par exemple :

```
function P_COMPLEXE.Sqrt (X : T_Complexe) return T_Complexe;
```

permet d'accéder, **seulement pour cette fonctionnalité** bien précise, à la structure cachée des `T_Complexe`.

**Un exemple concret (la structure de Ada95).**

En Ada 95 le paquetage `STANDARD` est le père de toutes les unités de compilation. Il a trois fils distincts pour regrouper les éléments prédéfinis ce sont : `INTERFACES` (qui concerne la liaison avec les autres langages), `SYSTEM` (pour tout ce qui concerne la dépendance de l'implémentation) et `ADA` (lui-même père de très nombreux paquetages comme `TEXT_IO`, `STRINGS`, `CHARACTERS`, `CALENDAR`, `FINALIZATION`, `STREAMS`, `EXCEPTIONS`, `TAGS`, `NUMERICS`, `DECIMAL`, `COMMAND_LINE`, `SEQUENTIAL_IO`, `DIRECT_IO`, etc.). On retrouve de vieilles connaissances déjà utilisé en TD-TP mais beaucoup de petits nouveaux. Voir page suivante quelques détails et développements intéressants.

On remarque qu'il faut nommer `ADA.TEXT_IO` le paquetage bien connu sur les E/S textes. Mais la déclaration suivante peut nous en dispenser (parfois prédéfinie dans certains compilateurs) :

```
with ADA.TEXT_IO;
package TEXT_IO renames ADA.TEXT_IO;
```

Cette technique a déjà été évoquée précédemment page 12 (compilation d'un surnom).

**Les fils privés (restriction des fils publics).**

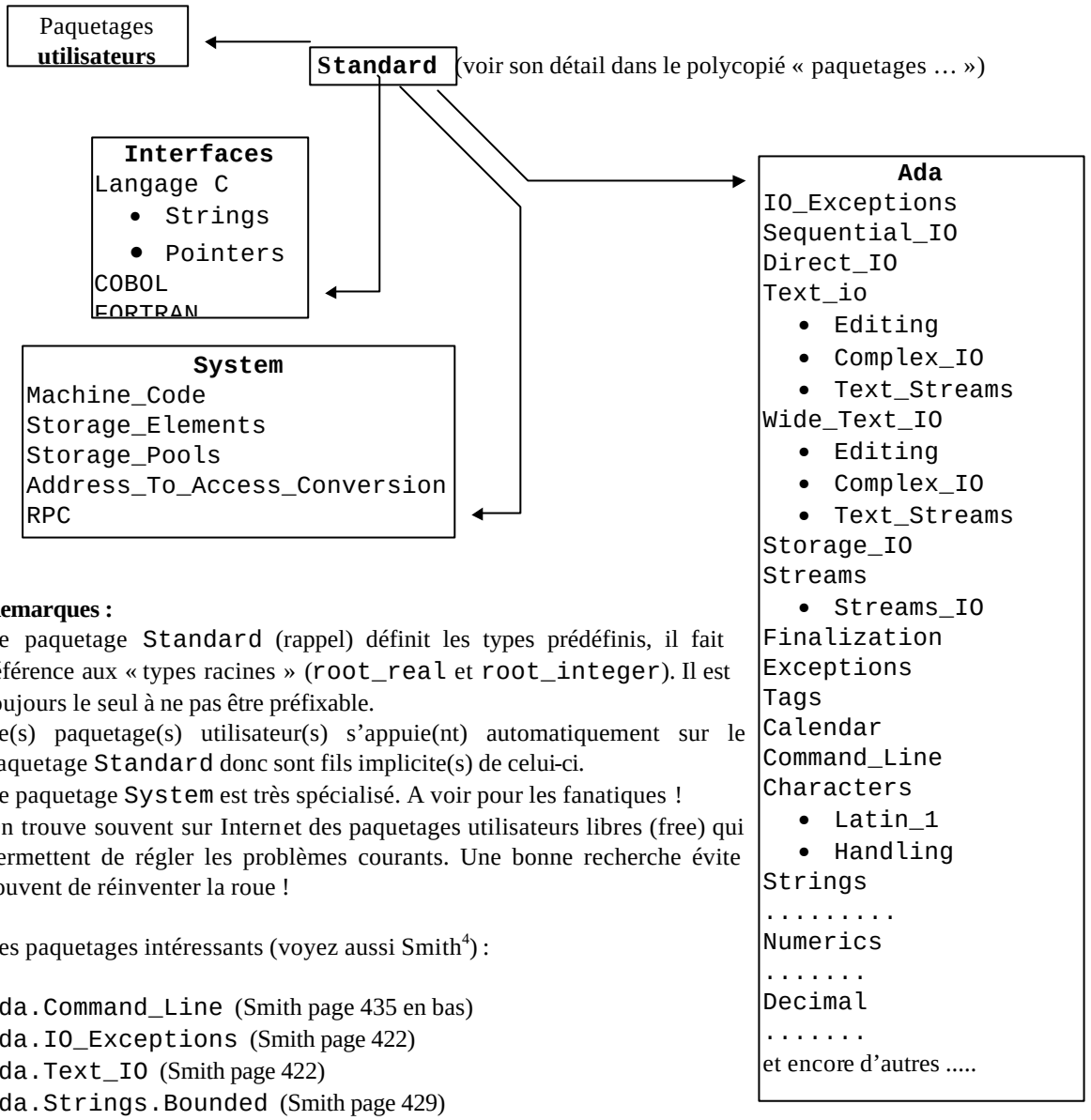
Le qualificatif de fils « privé » est obtenu en ajoutant, **avant la déclaration,** le mot réservé `private` ainsi :

```
private package P_COMPLEXE.LE_PRIVE is
......
end P_COMPLEXE.LE_PRIVE;
```

Ce fils « dit privé » `P_COMPLEXE.LE_PRIVE` a donc un père : `P_COMPLEXE` évidemment et au moins un frère (non privé) : `P_COMPLEXE.ELARGI`. Les propriétés de ce fils privé est de n'être visible que de l'intérieur du sous arbre de la hiérarchie familiale. **Il n'est donc pas connu** d'un client traditionnel de la famille `P_COMPLEXE`. De plus il **n'est plus visible** aux spécifications des frères (ceux qui sont non privés seulement). Il reste cependant visible aux corps de tous les membres de la famille. Grâce à ces propriétés la partie **visible** du fils privé peut maintenant avoir accès à la partie **privée** du père car il ne risque plus d'exporter les informations privées du père vers les utilisateurs ! Un paquetage privé permet de réaliser une Boîte à Outils interne à la famille où chaque corps (père, fils etc.) ira utiliser des sous programmes utiles.

**Les fils génériques.** Cette partie sera développée cours n°9 avec l'étude des génériques.

**Panorama de l'organisation des paquetages et des sous paquetages Ada95 :**




**Remarques :**

Le paquetage `Standard` (rappel) définit les types prédéfinis, il fait référence aux « types racines » (`root_real` et `root_integer`). Il est toujours le seul à ne pas être préfixable.

Le(s) paquetage(s) utilisateur(s) s'appuie(nt) automatiquement sur le paquetage `Standard` donc sont fils implicite(s) de celui-ci.

Le paquetage `System` est très spécialisé. A voir pour les fanatiques !

On trouve souvent sur Internet des paquetages utilisateurs libres (free) qui permettent de régler les problèmes courants. Une bonne recherche évite souvent de réinventer la roue !

Les paquetages intéressants (voyez aussi Smith[4]) :

`Ada.Command_Line` (Smith page 435 en bas)
`Ada.IO_Exceptions` (Smith page 422)
`Ada.Text_IO` (Smith page 422)
`Ada.Strings.Bounded` (Smith page 429)
`Ada.Numerics` (Smith page 434)
`Ada.Numerics.Generic_elementary_functions` (Smith page 435 haut)
`Ada.Calendar` (Smith page 436)
`Ada.Characters.Handling` (Smith page 428)
`Ada.Finalization` (Smith page 436)
`Ada.Tags` (Smith page 436)
`Interfaces.C` (Smith page 433)
`Ada.Sequential_IO` (Smith page 427)
`System` (Smith page 437)

On rappelle que tous ces paquetages doivent être **consultables** rapidement pour être utilisés avec efficacité. Donc il faut absolument en avoir une copie lisible avec soi (la plupart sont sur le polycopié « paquetages … » !).

[4] Object-Oriented Software in Ada 95 (livré avec les 5 compilateurs Aonix) en bibliothèque.

# Cours 8 Ada les exceptions (2 heures)

## Thème : les exceptions

**Avertissement** : Comme pour les paquetages nous avons déjà été amenés, au cours des TP de programmation, à « parler » des exceptions et à les utiliser de façon très implicite. C'est le moment (semaine 6 !) de structurer tout cela ! Une remarquable analyse des exceptions est faite par J.P. Rosen dans son livre (ouvrage recommandé dans le cours n°1). Voir le chapitre 20.4 «Politiques de gestion des erreurs » pages 275 à 285 (tout y est!). Les exceptions existent en Ada depuis son origine 83, de nombreux langages nés après lui (C++, Java et même Eiffel) ont adopté des fonctionnalités identiques. En Ada95 des ajouts intéressants sont venus les compléter.

**Introduction.**

Au cours du déroulement d'un programme, des « anomalies » peuvent perturber son fonctionnement : division par zéro, dépassement de bornes d'un tableau, initialisation hors des valeurs du type, fichier inexistant, etc. pour ne citer que les plus classiques. Imaginons un T.A.D. pour lequel on souhaite prendre en compte des anomalies éventuelles de fonctionnement. Une « **astuce** » pourrait consister à ajouter un paramètre booléen aux procédures pour indiquer en «sortie » leur réussite ou leur échec ! A charge pour l'utilisateur d'en prendre connaissance et d'utiliser cette information. Solution **lourde** ! Ada permet les **exceptions** : solution originale à notre problème. Parfois, suivant le problème l'« anomalie » n'a pas le sens d'erreur que les exemples suggèrent ! En Ada une **exception est associée à un événement exceptionnel** nécessitant **l'arrêt du traitement normal** et impliquant un **traitement particulier**. Notez que l'on a préféré parler d'événement exceptionnel (ou de situation anormale**) plutôt que d'erreur** (trop péjorative comme dit plus haut) même si c'est souvent le cas.

On distingue d'une part la **levée de l'exception** (ou son **déclenchement**), à cause justement de l'événement « anormal » qui s'est produit, et d'autre part le **traitement de l'exception** (que fait-on quand l'événement anormal m'est signalé ? Comment réagir ?).

De nombreuses levées d'exception peuvent être évitées par des tests «intelligents » de diagnostic (tests logiques). Par exemple utiliser `End_Of_File` pour éviter l'exception `End_Error` lorsque l'algorithme persiste à lire au delà du fichier! Dans un autre ordre d'idées un langage qui ne signalerait pas qu'une condition anormale s'est produite est un langage inachevé voire dangereux. J'ai souvenir du Turbo Pascal qui, quand un entier était saturé (32767), continuait à cumuler avec des valeurs ….. négatives ! Comme le dit Rosen « il y a pire qu'un programme qui se plante c'est un programme qui délivre des résultats faux mais vraisemblables ».

Une exception en Ada est **déclenchée** par :
- le « matériel »
- l'exécutif Ada ou l'environnement (par exemple en E/S manipulation incorrecte de fichiers),
- le programme lui-même.

Dans ce dernier cas (déclenchée par le programme lui-même) alors il s'agit :
- d'exceptions **prédéfinies** qui se déclenchent lorsque les « erreurs » se produisent.
- d'exceptions **définies dans le programme** dont le déclenchement est **explicite** (voir l'instruction **`raise`** et plus récemment (Ada95) et encore mieux `Raise_Exception`). Ces levées d'exceptions sont utilisées **lorsqu'un test logique** sur des variables ou des attributs **détecte une anomalie**.

Toute exception déclenchée lors de l'exécution d'une unité de programme est normalement **propagée** à l'unité appelante. Cette propagation **peut continuer** jusqu'à provoquer **l'arrêt** du programme.

Que faire pour éviter l'arrêt du programme (propagation en cascade). Il suffit de **traiter** l'exception c'est-à-dire mettre en œuvre une séquence d'instructions «particulières » qui se substitue à la suite d'instructions « normales » qui ne doivent plus s'effectuer. En Ada une exception peut être traitée **localement** (c'est-à-dire dans l'unité de programme où l'exécution l'a déclenchée) ou alors propagée (donc levée à nouveau) à l'unité appelante pour subir peut-être à cet endroit un traitement approprié. Dans le cas contraire (non prise en compte de l'exception) la propagation continue jusqu'à peut-être arriver au programme principal qui avorte lamentablement. Mais comme on l'a dit mieux vaut peut-être cela qu'un traitement erroné qui continue. Affaire de goût !

Mais avant d'être **levée** pour être éventuellement **traitée** une exception se **déclare** !

## Déclaration d'une exception.

Une exception est soit :
- prédéfinie (dans le paquetage `Standard`).
- « fournie » par un paquetage spécifique (`IO_EXCEPTION` par exemple).
- déclarée par l'utilisateur (avec le mot réservé **`exception`**).

On examinera les trois cas de « déclarations » et des exemples.

- **Les exceptions prédéfinies** (qui ne se déclarent donc pas !)

Elles sont déclarées dans le paquetage `STANDARD`, c'est-à-dire l'environnement dans lequel se trouve toute application. Essentiellement on a les **quatre exceptions** suivantes :

| | |
|---|---|
| `CONSTRAINT_ERROR` | Un tableau est indexé en dehors de ses bornes.<br>Un champ d'article est inexistant.<br>Un accès est nul<br>Erreur dans un calcul (débordement par exemple) |
| `PROGRAM_ERROR` | Appel de S/P non encore élaboré [1] (S/P **`separate`**).<br>Sortie de fonction sans passer par **`return`**. |
| `TASKING_ERROR` | Logique de communication entre tâches mise en cause. (voir cours `TACHES`) |
| `STORAGE_ERROR` | Place mémoire insuffisante (voir exemples avec la récursivité). |
| `NUMERIC_ERROR` | citée pour mémoire (obsolète) « confondue » avec `CONSTRAINT_ERROR` ! |

- **Les exceptions fournies par un paquetage connu** (surtout `IO_EXCEPTIONS`) voir le cours n°11 à venir (fichiers) et voyez l'annexe A.13 dans le polycopié « paquetages…. ».

Les exceptions liées aux E/S (à revoir) dont la déclaration est faite dans le paquetage `IO_EXCEPTIONS`.

| | |
|---|---|
| `STATUS_ERROR` | L'état du fichier utilisé est incorrect (travail sur fichier non ouvert par exemple). |
| `NAME_ERROR` | Le nom physique du fichier est incorrect (ou le fichier est absent). |
| `USE_ERROR` | L'utilisation du fichier est incorrecte. (ex: ouverture en accès direct d'un fichier séquentiel.) |
| `END_ERROR` | Fin de fichier "dépassée". |
| `DATA_ERROR` | Donnée incorrecte (ce que l'on lit n'est pas du type déclaré). |
| `LAYOUT_ERROR` | Chaîne trop longue pour impression |
| `MODE_ERROR` | Mode d'utilisation incorrect (écriture sur fichier ouvert en lecture). |
| `DEVICE_ERROR` | Utilisation incorrecte du matériel (périphériques par exemple). |

- **Les exceptions déclarées par l'utilisateur.**

Une déclaration d'exception peut apparaître dans n'importe quelle partie déclarative d'un bloc ou d'une unité (fonction, procédure, paquetage). Il suffit d'utiliser le mot réservé **`exception`** !

```
-- partie déclaration

EXC_MOT_TROP_LONG : exception; -- est déclarée par le programmeur

EXC_VALEUR_NON_TROUVEE : exception ;
```

Notez qu'il est recommandé de préfixer les exceptions « utilisateur » par `EXC_` ceci pour une meilleure lisibilité du programme ; de la même manière que nous avions recommandé de préfixer les types par `T_`.

[1] Voir à ce sujet les pragmas `Elaborate`, `Elaborate_All` et `Elaborate_Body`.

## Le traitement des exceptions déclenchées.

Rappel : Un bloc est constitué de deux parties : une partie déclarative et un corps :

```
{declare}  -- declare ou en-tête d'un sous-programme
{partie déclaration}
begin
   partie instructions
   ..............
{exception
   partie exception}
end;
```

C'est ici (partie exception) que l'on traite les exceptions levées dans la partie instructions

**Remarque :**

```
{declare}  -- declare ou en-tête d'un sous-programme
  {partie déclaration}
  begin
     partie instructions
     ..............
  {exception
     partie exception}
  end;
```

Si l'exception est levée avant le begin, donc à l'élaboration des déclarations, elle ne peut pas être traitée dans la fin de bloc! Voir plus loin !

Un bloc ou plus généralement une unité (fonction, procédure) peuvent contenir une **séquence de traitement des exceptions**, commençant au mot **`exception`** et se terminant au mot **`end`** du bloc ou de l'unité. La partie exception contient un ou plusieurs **récupérateurs d'exceptions** qui ont pour rôle d'effectuer un traitement lorsque certaines exceptions sont déclenchées ou propagées dans le bloc ou l'unité.

On peut écrire plusieurs traitements d'exception entre les délimiteurs **`exception`** et **`end`**.

**Exemple** (analogue à la syntaxe du **`case`** !) :

```
begin
     ..........
exception
     when CONSTRAINT_ERROR     => PUT("Erreur numérique ");
     when STORAGE_ERROR        => PUT("Mémoire insuffisante ");
     when others               => PUT("Autre erreur. ");
end;
```

**Remarques** :

- On notera que le contrôle ne peut **jamais être rendu** ensuite **au cadre où exception a été levée**.
- La suite d'instructions suivant le symbole => contient le traitement d'exception et **achève** ainsi **l'exécution du cadre**. Par conséquent pour une fonction un traitement d'exception **doit comporter** aussi une instruction **`return`** pour fournir un résultat (sinon on retourne l'exception PROGRAM_ERROR) à moins que le traitement d'exception consiste à lever à nouveau une exception **(`raise`)**.

```
function DEMAIN (AUJOURD_HUI : T_JOUR) return T_JOUR is
begin
     return T_JOUR'SUCC(AUJOURD_HUI);
exception
     when CONSTRAINT_ERROR => return T_JOUR'FIRST;
end DEMAIN;
```

Un **`if`** serait plus approprié !

Les instructions de traitement d'exception peuvent être **aussi complexes que l'on veut** (les exemples avec PUT plus haut sont caricaturaux!). Les instructions de traitement d'exception peuvent comprendre des blocs, des appels de S/P, et même des levées d'exceptions ou des traitements d'exception, etc.

## Déclenchement (ou levée) et éventuellement propagation d'une exception.

**Lever une exception.**

On dit également **déclencher** l'exception (**`raise`** en anglais) par exemple avec l'instruction :
`raise NOM_DE_L_EXCEPTION; -- l'exception est levée`
signifie que « l'algorithme » s'apercevant d'un problème déclenche l'exception. L'algorithme ne **continue pas en séquence**. Il est « interrompu » et dirigé vers **l'éventuel traitement** prévu **en fin de bloc**.

**Exemple** :

```
     -- partie déclaration
     EXC_ERREUR_ACQUISITION  : exception;
     EXC_ERREUR_CALCUL  : exception;

     begin
       ..........
       if (.....) then
          raise EXC_ERREUR_ACQUISITION;
       end if;
       ..........
       if (.....) then
          raise EXC_ERREUR_CALCUL;
       end if;
       ..........
     exception
         when  EXC_ERREUR_ACQUISITION    =>    TRAITEMENT_1;
         when  EXC_ERREUR_CALCUL         =>    TRAITEMENT_2;
         when  others                    =>    TRAITEMENT_3;
     end;
```

Schéma classique de validation ou test de cohérence. A approfondir voir plus loin.

**Exemple** (connu !) : Lecture validée (d'un entier de type `T_ENTIER`) sur le modèle de la lecture validée d'un énumératif déjà largement utilisée! Attention : le bloc est dans un **`loop`**.

```
procedure LIRE (RES : out  T_ENTIER) is
  L: T_ENTIER ;
begin
  loop
    declare
    --  pour traiter l'exception sur le champ c'est-à-dire tout de suite et
    --  non en fin de procédure il faut utiliser un bloc qui possède un end
     begin
       GET (L);
       SKIP_LINE ;
       exit;
       exception
            when DATA_ERROR =>  SKIP_LINE; PUT_LINE ("Un entier S.V.P ");
     end;
     -- fin de bloc
  end loop;
  RES := L;
-- fin du traitement avec acquisition correcte
end LIRE;
```

Notez ces `SKIP_LINE` (voir cours E/S)

On verra dans le cours fichier la traditionnelle procédure `OUVRIR` d'un fichier.

**Remarque**: Quelquefois il est pratique de traiter l'exception puis de **propager** la même exception.

```
     ..........
     when others   =>   PUT("Autre erreur  "); raise;
     end;
```

voir complément à ce sujet page 8.

**Propagation d'une exception.**

Le déclenchement d'une exception entraîne **l'abandon de l'exécution normale** de la séquence restante du bloc de l'unité où a lieu l'exception (bis !) .

- Si le bloc **comporte un traitement** pour cette exception alors il y a exécution de l'action associée à ce traitement, l'unité en cours se terminant après le traitement de l'exception. Ceci correspond à l'exemple donné (dans ce cas il n'y a **pas de propagation**). Le **contrôle** est ensuite **rendu à l'unité appelante**.

- Si le bloc **ne comporte pas** de récupérateur pour cette exception alors **l'unité** (où a lieu l'exception) **se termine** à l'instruction qui l'a déclenchée. Cette **exception est propagée** à l'unité appelante. Cette propagation **continue tant que** l'unité qui la reçoit **n'a pas de récupérateur** pour cette exception. Dans les cas extrêmes le programme « **avorte** » au programme principal avec l'exception `PROGRAM_ERROR`.

**Portée des exceptions, problèmes divers.**

- Une exception peut être rendue visible par la notation pointée. Dans l'exemple de la pile (voir TD associé) sans la clause **`use`** `PILE` il faut écrire:

```
when PILE.EXC_PILE_PLEINE      => ..... ;
when PILE.EXC_PILE_VIDE        => ..... ;
```

- Une exception peut être renommée :

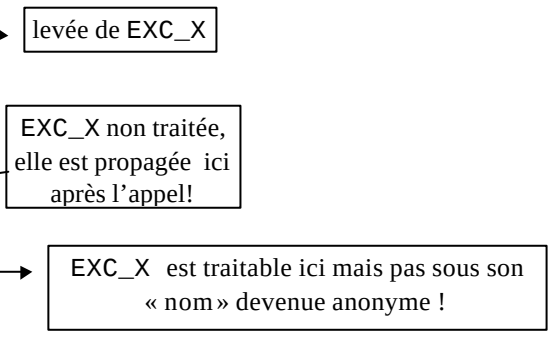
```
EXC_DEBORDEMENT_HAUT    :  exception renames PILE.EXC_PILE_PLEINE;
EXC_DEBORDEMENT_BAS     :  exception renames PILE.EXC_PILE_VIDE;
```

L'exemple le plus classique de renommage est celui du paquetage `ADA.TEXT_IO` qui renomme les exceptions de `IO_EXCEPTIONS` en utilisant … les mêmes identificateurs (voir annexe A.13. polycopié « paquetages... »)

- Une exception utilisateur, donc déclarée, peut être **propagée hors de sa portée**. Mais alors elle ne pourra être traitée que par **`others`** anonymement car son identificateur est inconnu sauf marquage spécial (à voir).

**Exemple** :

```
declare
     procedure PROC is
       EXC_X : exception;
     begin
          raise EXC_X;
     end PROC;
begin -- du bloc
     PROC;
.....
     exception
       when others  = > .....
end;
```




`PROC` **déclare** et **lève** `EXC_X` mais ne la traite pas. Donc `EXC_X` est propagée au bloc utilisant `PROC` où elle peut être traitée (mais pas sous son nom elle est devenue anonyme! Il faut utiliser **`others`**). Voir cependant plus loin `RAISE_EXCEPTION`.

- Une exception peut être levée à l'élaboration d'une déclaration (danger ! voir page 3).

Ainsi : `N : POSITIVE := 0; --lèvera CONSTRAINT_ERROR.`

**Retenir : Une exception levée dans l'élaboration d'une déclaration n'est pas traitée par le traitement d'exception du cadre contenant la déclaration, mais est immédiatement propagée au niveau supérieur (donc prudence !).**

Cette propriété peut être utilisée pour « faire la chasse aux codes morts » voir la démonstration en cours.

Cas d'une procédure **récursive** contenant une déclaration d'exception. Contrairement aux variables déclarées dans une procédure **on n'obtient pas de nouvelle exception** pour chaque appel récursif. **Chaque appel récursif se référera à la même exception.**

Exemple :

```
procedure F(N : INTEGER) is
EXC_X :  exception;
begin
     if N = 0  then
          raise EXC_X;
     end if;
     F(N-1);
exception
     when EXC_X =>  PUT ("Je la tiens "); PUT(N);
end F;
```

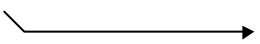
Attention exemple pédagogique uniquement !

Avec F(4) on obtient les appels récursifs F(3), F(2), F(1), F(0) (cf. TD récursivité!).

Quand F est appelée avec un paramètre nul, F lève l'exception qui imprime le message `Je la tiens` et édite la valeur 0. **L'exception ne sera plus traitée et sortira du cadre du premier appel.**

Si nous écrivons:

```
exception
     when EXC_X  => PUT("Je la tiens "); PUT (N);
     raise;
end;
```



En plus

Dans ce cas F(0) lève tout d'abord l'exception qui est ensuite levée anonymement à chaque appel récursif et le message apparaîtra alors cinq fois.

Utilisation des exceptions dans l'écriture d'une **fonction de validation** d'un « objet » quelconque :

```
function EST_VALIDE (OBJET : in T_OBJET) return boolean is
...
begin
...
     if (test_spécifique_de_non_validité_de_l'objet)
     then return FALSE ;
     end if ;
...
     -- on répète la séquence ci-dessus avec d'autres tests spécifiques
...
     ACTION(OBJET) ; -- qui déclenche peut-être une exception
...
     -- on répète la séquence ci-dessus avec d'autres actions
...
     return TRUE ; -- tout c'est bien passé !

     exception when others => return FALSE ;

end EST_VALIDE ;
```

voir aussi les remarques de Barnes page 575.

## Compléments à privilégier.

La fonction de validation d'un objet (voir son schéma général ci-dessus) peut être complétée par une **meilleure connaissance** des anomalies déclenchées. En effet il est toujours frustrant de traiter des exceptions par :

```
exception when others => ... ;
```

La « gestion » des exceptions traitées (par exemple mise en place d'une trace des anomalies dans la phase de mise au point d'un logiciel appelée « debuggage » en jargon informatique) n'est guère facile car les exceptions (si elles ont bien un identificateur « local ») ne peuvent être repérées beaucoup plus loin (hors du bloc de déclaration) dans le déroulement du programme. Il peut être intéressant de **mieux documenter** une exception en la levant et de récupérer cette documentation le moment opportun i.e. dans le traitement.

Le paquetage `ADA.EXCEPTIONS` (voir dans polycopié « paquetages en 11.4.1 ») propose pour une meilleure gestion des exceptions notamment les entités suivantes à détailler :

- des fonctions :
  `EXCEPTION_NAME`, `EXCEPTION_MESSAGE` et `EXCEPTION_INFORMATION`.
- des procédures :
  `RAISE_EXCEPTION, SAVE_OCCURRENCE`
- deux types :
  `EXCEPTION_ID, EXCEPTION_OCCURRENCE`

**Capture et édition de l'identité d'une exception.**

`EXCEPTION_ID` de type **`private`** permet de manipuler « l'identité » d'une exception. Pour fixer les idées on peut comprendre par identité au moins le nom complet de l'exception (**préfixe compris** c'est-à-dire l'unité dans laquelle elle est déclarée). L'association entre : une exception et son identité se fait grâce à l'attribut `Identity`. Par exemple avec les déclarations préalables suivantes :

```
UNE_EXCEP : exception ; -- l'identificateur de l'exception
IDENT : EXCEPTION_ID ;  -- pour capturer son identité
```

Et l'instruction : `IDENT := UNE_EXCEP'IDENTITY ;` a capturé l'identité de l'exception.

La fonction `EXCEPTION_NAME` utilisée avec un paramètre effectif instance de type `EXCEPTION_ID` retourne un `String` identifiant **complètement** l'exception associée. Ainsi :

```
PUT_LINE (EXCEPTION_NAME (IDENT)) ;
```

affiche la chaîne `"XXXX.UNE_EXCEP"` où `XXXX` représente l'identificateur du sous programme dans lequel l'exception `UNE_EXCEP` est déclarée.

Comment peut-on associer **encore plus de renseignements** à une exception au moment où elle estdéclenchée ?

Pour ce faire nous disposons de la procédure `RAISE_EXCEPTION` qui lève, à la fois, l'exception et qui la documente. Remarquez et notez que cette procédure utilise en paramètre **l'identité** de l'exception et non pas l'exception elle même (c'est cette dernière **qui est cependant levée**).

Exemples :

```
RAISE_EXCEPTION (UNE_EXCEP'Identity, "Erreur de calcul") ;
……..
RAISE_EXCEPTION (UNE_EXCEP'Identity, "débordement de tableau") ;
```

Cette technique (plus élaborée) se substitue alors au schéma traditionnel :

```
raise UNE_EXCEP ;
……
raise UNE_EXCEP ;
```

La même exception (UNE_EXCEP), levée deux fois avec RAISE_EXCEPTION, est alors **différenciable ultérieurement** dans un traite exception comme on le verra ci-dessous.

**Documentation (traçage)**

La documentation de l'exception représentée par une chaîne de caractères est soit, une documentation dite « courte » on l'appelle MESSAGE, soit une documentation dite « longue » on l'appelle INFORMATION.

Voyons maintenant la prise en compte (donc le traitement plus élaboré) des exceptions levées avec RAISE_EXCEPTION.

Comme avec le **raise**, le traitement se fait dans un récupérateur d'exception (en fin de bloc) mais utilise le type EXCEPTION_OCCURRENCE. Ce type joue le rôle d'un marqueur pour l'exception et il est **limité privé**. L'association entre l'exception et son marqueur se fait dans le traite exception. Exemple :

```
     MARQUEUR : EXCEPTION_OCCURRENCE ;
.....
.....
     exception
          when ....
          .....
          when MARQUEUR : others => ..... ;
     end;
```

Déclaration non indispensable !
Bizarre en Ada !

On notera que la variable MARQUEUR **n'a même pas à être déclarée** ! C'est à ma connaissance, un cas unique en Ada !

MARQUEUR contient : non seulement l'identité de l'exception levée et récupérée anonymement avec **others** mais aussi éventuellement la documentation associée si elle existe (voir RAISE_EXCEPTION ci-dessus). Dans le traite exception (après le =>) il est possible de connaître, au choix, respectivement :

l'identité, le message ou l'information (identité + message) associés à l'exception.

On utilise pour cela respectivement les fonctions :

EXCEPTION_NAME, EXCEPTION_MESSAGE et EXCEPTION_INFORMATION qui retournent un String mais utilisent en paramètre le marqueur d'exception et non pas l'exception elle-même.

```
PUT_LINE (EXCEPTION_NAME (MARQUEUR)) ; -- l'identité

PUT_LINE (EXCEPTION_MESSAGE (MARQUEUR)) ; -- le message

PUT_LINE (EXCEPTION_INFORMATION (MARQUEUR)) ; -- les deux !
```

Le paquetage ADA.EXCEPTIONS propose encore beaucoup des fonctionnalités et de types à découvrir. Il est recommandé d'étudier un listing de ce paquetage. Voir aussi dans Barnes page 308 la possibilité de stocker (avec SAVE_OCCURRENCE) dans un tableau des marqueurs pour un traitement retardé. Attention c'est coûteux !

**Remarques** : (à propos du **raise** qui permet de propager la même exception).

Le **raise** s'utilise seul (on l'a vu) mais **seulement dans un traite exception** (ce qui est évident).
Voici un exemple très significatif de l'intérêt de ce **raise** (cité par J.P. Rosen dans ses fiches pratiques [2]).

[2] On trouvera ces fiches pratiques (très instructives) dans son site http://perso.wanadoo.fr/adalog voyez le lien Articles techniques. Voyez aussi le lien sur le catalogue des principales ressources disponibles via Internet, c'est fou ce que l'on peut trouver sur Ada.

Problème : Vous souhaitez, dans un traite exception (donc en fin de bloc), faire, pour toutes les exceptions traitées, un traitement commun, et ensuite, faire, pour chaque exception, un traitement particulier.

Une solution **lourde** mais correcte pourrait être :

```
Exception

   when Constraint_Error => Traitement_Commun ;
                            Traitement_spécial_n°1 ;

   when Data_Error =>       Traitement_Commun ;
                            Traitement_spécial_n°2 ;

   when Exc_Valeur_Nulle => Traitement_Commun ;
                            Traitement_spécial_n°3 ;

   when others =>           Traitement_Commun ;
                            Traitement_spécial_n°4 ;

 end ... ;
```

Une solution plus élégante :

```
exception
   when others => Traitement_Commun ;
   begin
       raise ;  -- notez bien sa place et le deuxième bloc imbriqué !
   exception
      when Constraint_Error => Traitement_spécial_n°1 ;
      when Data_Error =>       Traitement_spécial_n°2 ;
      when Exc_Valeur_Nulle => Traitement_spécial_n°3 ;
      when others =>           Traitement_spécial_n°4 ;
   end ;
 end ... ;
```

Cette dernière solution plus modulaire est **bien plus facile à maintenir**.

**Complément** : une exception levée dans le bloc d'initialisation d'un paquetage ne peut être récupéré dans le corps de l'unité élaborant ce paquetage. Il faut prévoir le traite exception en fin du bloc d'initialisation du paquetage ou transformer le bloc d'initialisation en une vraie procédure appelée au début de l'unité élaborant le paquetage.

Celles et ceux que la gestion des erreurs intéressent et qui est d'ailleurs un point capital en informatique sécurisée trouveront dans le livre de Rosen (voir au début de ce chapitre) une étude très intéressante recouvrant :

- Politique de correction locale
- Politique de code de retour
- Politique de déroutement
- Politique du contrat
- Politique par exceptions simples
- Politique Oméga
- Politique de gestion centralisée

Très complet !

# Cours 9 Ada la généricité (2 heures)

## Thème : la généricité.

**Introduction (attention : dure, dure !).**

La généricité permet d'élargir le contexte d'utilisation d'une unité de programme. Elle permet de **définir** des **familles paramétrées d'unités** de programme. Les unités d'une même famille ne différant que par un certain nombre de caractéristiques décrites à l'aide de **paramètres formels génériques**. La **création** d'une unité de programme (vraie ou concrète) à partir d'une unité générique (abstraite ou formelle) est faite par **instanciation**. On associe les paramètres effectifs aux paramètres formels génériques (comme pour les associations paramètre effectif ⇔ paramètre formel des sous programmes).

**Exemple simple** (en cours nous prendrons d'autres exemples pour mieux concrétiser ce concept) **:**
Soit la procédure ci-dessous échangeant deux objets de type entier :

```
procedure  ECHANGE_ENTIER (PREMIER, SECOND : in out T_ENTIER) is
     TAMPON  :  T_ENTIER;
begin
     TAMPON    :=  PREMIER;
     PREMIER   :=  SECOND;
     SECOND    :=  TAMPON;
end   ECHANGE_ENTIER;
```

Pour échanger deux réels il faut **réécrire** une nouvelle procédure avec un type `FLOAT` par exemple remplaçant le type `T_ENTIER` (vive le copier-coller pour les bœufs !). La généricité va permettre d'écrire un «moule » (un modèle) à partir duquel on peut créer des procédures spécifiques à chaque type, **seul le type** en question **diffère**. Le type en question constitue ce que l'on appelle un type **générique**. Les paramètres génériques sont déclarés, d'abord, à la suite du mot clé **`generic`.** La déclaration s'arrête à la rencontre d'un des 3 mots réservés : **`procedure`**, **`function`** ou **`package`**.

```
generic
  type T_ELEMENT  is  private;
procedure ECHANGE_TOUT (PREMIER, SECOND : in  out  T_ELEMENT);
...
procedure ECHANGE_TOUT (PREMIER, SECOND  : in  out  T_ELEMENT) is
TAMPON  :  T_ELEMENT;
begin
       TAMPON    :=  PREMIER;
       PREMIER   :=  SECOND;
       SECOND    :=  TAMPON;
end  ECHANGE_TOUT;
```

Ici, un seul paramètre générique !

Notez les deux parties distinctes : spécifications puis réalisation

Grâce à la procédure générique `ECHANGE_TOUT` il est possible de créer en les **spécialisant par un type précis**, de nouvelles procédures. **L'opération de création est ici l'instanciation** de procédures (notez le **`new`**).

```
procedure  ECHANGE_CARAC is new ECHANGE_TOUT (T_ELEMENT => CHARACTER);
procedure  ECHANGE_ENTIER is new ECHANGE_TOUT (T_ELEMENT => T_ENTIER);
procedure  ECHANGE_REEL is new ECHANGE_TOUT (T_ELEMENT => FLOAT);
```

L'association par nom est facultative mais est **très lisible et fortement conseillée** (revoir l'agrégation) !

**Mise en œuvre de la généricité.**

La généricité s'applique à deux unités de programmes :

- les sous-programmes (procédure ou fonction). Moins usité.
- les paquetages. Surtout eux !

Une unité générique comporte (comme toute unité de programme) une partie spécifications et une partie réalisation (avec corps) qui peuvent être **compilés séparément**. L'utilisation d'une instanciation par exemple d'un paquetage générique est **possible dès l'opération d'instanciation**. Les unités génériques et les instanciations de générique peuvent être des unités de bibliothèque. Si le paquetage générique `P_PILE` (voir TD correspondant) est introduit dans la bibliothèque de programme (grâce à une compilation par exemple), on peut alors compiler une instanciation isolée par :

```
with P_PILE;  ... – pas de use (absurde!)
package  PILE_BOOLEENNE is new  P_PILE (200, BOOLEAN);
```

ou bien encore (Voir cours E/S)

```
with  ADA.TEXT_IO; ...
package  ENTIER_IO  is new  ADA.TEXT_IO.INTEGER_IO (T_ENTIER);
```

L'emboîtement d'unités génériques est possible mais ne doit pas être récursif (voir TD récursivité).
**Exemple** d'emboîtements :

```
generic
     type  T_OBJET  is private;
procedure  DECAL_GAUCHE  (A, B, C  :  in  out  T_OBJET)  is
     procedure  TROC is new  ECHANGE_TOUT (T_ELEMENT => T_OBJET);
begin
     TROC  (A,B);
     TROC  (B,C);
end  DECAL_GAUCHE;
```

procedure Decal_Entier is new DECAL_GAUCHE (T_ENTIER) ;
permet de créer une vraie procédure Decal_Entier

Remarque : le type effectif (sur l'exemple `T_ENTIER`) associé à `T_OBJET` au moment de l'instanciation de `DECAL_GAUCHE` servira aussi à l'instanciation interne de `TROC`. Instanciation en cascade

**Spécification générique.**

Une spécification générique commence par le mot clé **`generic`** suivi de la liste des paramètres formels génériques et se termine comme une unité non générique (c'est-à-dire par le **`end`** de l'unité). Mais la liste des paramètres, elle, s'arrête (on l'a dit) à l'un des mots réservés : **`package`**, **`procedure`** ou **`function`**.

Soit le schéma formel :

```
generic
--   liste des paramètres génériques (voir plus loin une étude détaillée)
package NOM_DU_PAQUETAGE  is
         --   les objets exportés
end  NOM_DU_PAQUETAGE;  -- fin de la spécification générique
```

Exemple :

Début de la liste des paramètres génériques

4 paramètres génériques

```
generic
     type  PIXEL       is range    <>;
     type  ABSCISSE    is range    <>;
     type  ORDONNEE    is range    <>;
     type  IMAGE       is array (ABSCISSE,ORDONNEE) of PIXEL;

package  LOGICIEL_IMAGE  is
```

Le mot package marque la fin de la liste des paramètres génériques

```
     --   partie visible et partie privée du paquetage
end  LOGICIEL_IMAGE;
```

Les paramètres formels de la généricité sont classés en :
- paramètres vus comme des valeurs ou des variables.
- paramètres sous forme de types.
- paramètres sous-programmes.
- paramètres paquetages.

## • Les paramètres génériques « valeurs ou variables ».

1) Variables mais vues comme des "constantes" ou des données.
Ces objets sont considérés comme constants pour l'unité de programme générique et peuvent même avoir une valeur par défaut. Leur mode de passage est **in** (évidemment) sans qu'il soit besoin de le préciser.

**Exemple** :
```
generic
     TAILLE             :  NATURAL := 200;
     NOMBRE             :  in POSITIVE  := POSITIVE'LAST;
     NOMBRE_MAX     :  NATURAL;
package P_....... is
.....
end P_....... ;
```

2) Variables « vraies » en paramètre (**in out** obligatoires ; pas de **out**)

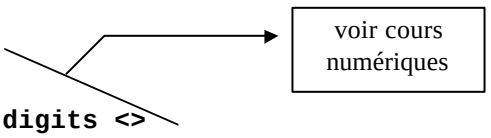
**Exemple** :
```
generic
     SCORE : in out NATURAL ;
package P_JEU is
.....
end P_JEU ;
```

Le paramètre effectif "associé" au paramètre formel SCORE (au moment de l'instanciation) sera tenu à jour tout au long de la portée du paquetage P_JEU instancié.

## • Les paramètres génériques sous forme de types.

La possibilité de passer des types en paramètre est **l'un des traits les plus remarquables de la généricité**. Dans une unité générique, les types passés en paramètres sont évidemment formels. Les types génériques constituent donc des **classes de type**.

On distingue **dix** catégories :
- Un type sur lequel on ne possède aucune information est déclaré **limited private**
- Un type sur lequel on exige au moins l'affectation et la comparaison sera noté **private**
- Un type discret sera noté **(<>)** (énumératifs presque essentiellement)
- Un type entier sera noté **range <>**
- Un type entier modulaire sera noté **mod** <>
- Un type réel point flottant sera noté **digits <>**
- Un type réel point fixe binaire sera noté **delta <>**
- Un type réel point fixe décimal sera noté **delta <> digits <>**
- Un type tableau sera noté **array**
- Un type accès (pointeur) sera introduit par **access** (voir le cours correspondant n°12).




Exemples :
```
generic
     type  T_CIRCULAIRE is  mod <>;
     type  T_INDICE     is  (<>);
     type  T_EURO  is delta <> digits <> ;
```

```
     type  T_LONGUEUR   is  range  <>;
     type  T_ANGLE      is  digits <>;
     type  T_ENTIER     is  range  <>;
     type  T_ACC_INFO   is  access  INFO;
```
.....

Les types génériques tableaux peuvent être **contraints** ou **non contraints**.

**Tableaux génériques <u>non contraints</u> :**

```
generic
     type  T_INDEX   is  (<>);   -- type discret
     type  T_TABLE   is  array  (T_INDEX range  <>) of T_ENTIER;
```
.....

Le type des éléments du tableau peut lui même être générique :

```
generic
     type  T_INDEX      is  (<>);
     type  T_ELEMENT    is private;
     type  T_TABLEAU    is array  (T_INDEX range <>)  of  T_ELEMENT;
```
.....

**Tableaux génériques <u>contraints</u> :**
Les indices sont indiqués par une marque de type discret : sans le "`range <>`" évidemment.

```
generic
     type  T_INDEX      is (<>);
     type  T_ELEMENT    is private;
     type  T_TABLE      is array (T_INDEX)  of INTEGER;
     type  T_TABLEAU    is array  (T_INDEX) of  T_ELEMENT;
```
.....

**Type générique formel et type générique effectif (association à l'instanciation) :**

Exemple simple : (on travaille ici avec une fonction générique et pas un paquetage générique)

```
generic
    type  T_DISCRET  is  (<>);
function  SUIVANT  (X : T_DISCRET)  return  T_DISCRET; -- spécifications
```
.....
puis plus loin la réalisation :

```
function  SUIVANT  (X : T_DISCRET)  return  T_DISCRET is -- définition
begin
     if  X = T_DISCRET'LAST
     then
          return  T_DISCRET'FIRST;
     else
          return  T_DISCRET'SUCC(X);
     end if;
end  SUIVANT;
```

Le paramètre formel `T_DISCRET` exige que le type effectif soit un type discret (à cause de: `is (<>)`). Les attributs `FIRST` et `LAST` peuvent être utilisés dans le corps car le type discret les possède. Nous pouvons instancier et écrire :

```
function  DEMAIN is new  SUIVANT(T_JOUR);
```
et `DEMAIN(DIMANCHE)` donne `LUNDI`.

Un type générique effectif peut aussi être un **sous-type** :

```
function  JOUR_OUVRE_SUIV  is new SUIVANT(T_JOUR_OUVRABLE);
```

avec `subtype T_JOUR_OUVRABLE is T_JOUR range LUNDI..VENDREDI;`

On obtient `JOUR_OUVRE_SUIV(VENDREDI)` donne `LUNDI`. C'est beau non ? !

**Autre exemple :**

Un paramètre générique formel peut dépendre d'un type paramètre formel précédent. C'est souvent le cas pour les tableaux. (ci dessous `T_VEC` dépend de `T_INDICE`) :

```
generic
     type  T_INDICE  is  (<>);
     type  T_FLOTTANT is  digits  <>;
     type  T_VEC  is array  (T_INDICE range  <>)  of  T_FLOTTANT;
function  SOMME  (A : T_VEC)  return  T_FLOTTANT;
.................

function  SOMME  (A : T_VEC)  return  T_FLOTTANT  is
RESULTAT  :  T_FLOTTANT  :=  0.0;
begin
     for  IND  in  A'RANGE  loop
          RESULTAT  :=  RESULTAT  + A(IND);
     end  loop;
     return  RESULTAT;
end  SOMME;
```

L'instanciation et l'utilisation de la fonction `SOMME` s'effectuent de la façon suivante :

```
procedure  TEST  is

     type  T_JOUR is  (LUNDI,….,DIMANCHE);
     type  T_REEL  is digits 15 ;
     type  T_VECTEUR  is array  (T_JOUR range <> ) of T_REEL;

     function  SOMME_VECTEUR  is new  SOMME(T_JOUR,T_REEL,T_VECTEUR);
     subtype  S_VECTEUR  is  T_VECTEUR (LUNDI..VENDREDI);

     TABLE     :  S_VECTEUR;
     S         :  T_REEL;
begin
     ............
     S := SOMME_VECTEUR(TABLE);
     ............
end  TEST;
```

ici on contraint enfin le tableau !

**Remarque :**

Attention à la correspondance entre type générique formel et type générique effectif. Si on avait écrit :

```
type T_VECTEUR is array (CHARACTER range  <>)  of  FLOAT;
```

l'instanciation s'écrirait :

```
function  SOMME_VECTEUR  is new SOMME(CHARACTER,FLOAT,T_VECTEUR);
```

- **Paramètres génériques sous-programmes (fonctions et procédures).**

Il est possible de « passer » des sous-programmes comme paramètres de la généricité. Ils seront décrits dans la liste des paramètres de l'unité générique et ils seront introduits par le mot clé **`with`**.

**Avant Ada 95 les S/P ne pouvaient être paramètres que des unités génériques[1].** Voyons cela.
Soit la fonction `INTEGRALE` générique (notez le **`with`** ! dans la « liste » des paramètres) :

```
generic
     with  function G (X : FLOAT)  return FLOAT;
function  INTEGRALE (A,B : FLOAT) return  FLOAT;
```

un paramètre générique « fonction »

Notez bien les deux mots function (pas la même chose)

qui évalue une fonction formelle comme :

$$\int_a^b G(x)\,dx$$

Pour intégrer une fonction particulière par exemple :

$$\int_0^{\pi} e^t * \sin(t)dt$$

On écrit d'abord la **fonction effective** F qui servira à instancier la fonction générique `INTEGRALE` :

```
function  F (T : FLOAT) return FLOAT  is
begin
     return  EXP(T) * SIN(T);
end  F;
```

puis on effectue **l'instanciation** de la fonction `INTEGRALE` avec le paramètre effectif `F` ainsi :

```
function INTEGRER_F is new INTEGRALE (F);
```

et nous obtenons le résultat en **utilisant** la fonction instanciée `INTEGRER_F` par exemple en écrivant :

```
X  :=  INTEGRER_F(0.0,PI);
```

La spécification du S/P formel peut dépendre de types formels précédents. Reprenons notre exemple :

```
generic
     type  T_FLOTTANT is digits <>; -- un type réel
     with  function G (X : T_FLOTTANT)  return  T_FLOTTANT;
function  INTEGRALE  (A,B : T_FLOTTANT)  return  T_FLOTTANT;
```

La fonction d'intégration s'applique à tout type point-flottant (cette classe de type réel sera vue plus tard).

```
type T_REEL is digits 8;
function INTEGRER_F is new INTEGRALE(T_REEL,F);
-- deux paramètres d'instanciation cette fois
```

**Les S/P « passés » en tant que paramètres génériques (fonctions ou procédures) permettent de spécifier les opérations disponibles sur les types formels génériques. Voyons un exemple :**

On veut généraliser la fonction générique `SOMME` vue en amont en passant l'opérateur + en paramètre générique et en rendant le type des composants plus « ouvert ». Soit l'identificateur `OPERER` pour cette nouvelle écriture.

```
generic
     type  T_INDICE  is  (<>);
     type  T_ITEM    is  private;
     type  T_VEC     is array  (T_INDICE  range  <> ) of T_ITEM;
     with  function  "+"  (X,Y : T_ITEM)  return  T_ITEM;
function OPERER (A : T_VEC) return  T_ITEM;
```

[1] Depuis les choses se sont améliorées ! la nouvelle norme permet de définir des « pointeurs » sur S/P. A revoir !

L'opérateur "+" a été ajouté comme **paramètre** car T_ITEM est déclaré **private** (type à affectation). Peut-on maintenant appliquer la fonction générique à **n'importe quelle opération binaire sur n'importe quel type ?**

Les instanciation s'écriraient :

```
function ADDI_R is new OPERER (T_JOUR,FLOAT,T_VECTEUR,"+");

function PROD_R is new OPERER (T_JOUR,FLOAT,T_VECTEUR,"*");
```

on souhaite des produits au lieu d'additions

De même peut-on définir les fonctions ADDI_E et PROD_E en remplaçant FLOAT par INTEGER ?

Ceci ne va pas **toujours sans problème** ! Voyons la définition de la fonction OPERER. Si cette fonction reprend l'esprit de la fonction SOMME page 5 soit :

```
function  OPERER  (A : T_VEC)  return  T_ITEM is
RESULTAT  :  T_ITEM  :=  0.0;
begin
     for  IND  in  A'RANGE  loop
          RESULTAT  :=  RESULTAT + A(IND);
     end  loop;
     return  RESULTAT;
end  OPERER;
```

Ici avec 0.0 il y a un gros problème suivant le type numérique !

Revoyons la liste des paramètres formels pour mémoire :

```
generic
     type  T_INDICE  is  (<>);
     type  T_ITEM    is  private;
     type  T_VEC     is array  (T_INDICE  range  <> ) of T_ITEM;
     with  function  "+"  (X,Y : T_ITEM)  return  T_ITEM;
function OPERER (A : T_VEC) return  T_ITEM;
```

T_INDICE est discret donc l'écriture A'RANGE est sans ambiguïté. L'opération symbolisée formellement par le + est une opération binaire entre deux T_ITEM elle sera (suivant l'instanciation) soit un « vrai » + ou alors un * (voir les deux instanciations proposées). Là encore pas de problème. Par contre l'affectation avec la valeur 0.0 est « osée ». D'abord on imagine ce que donnerait la fonction instanciée PROD avec une telle initialisation ! Il est clair qu'il faut initialiser RESULTAT avec la valeur « élément neutre » pour l'opération donc 0 ou 0.0 (suivant le type) pour l'addition et 1 ou 1.0 pour le produit. Donc même s'il n'y a pas deux opérateurs distincts on est obligé de connaître la valeur d'initialisation **suivant le type des éléments** du vecteur. Rappelons que **private** ne définit rien de précis (type à affectation : affectation et égalité seulement) !

Pour résoudre notre problème il faut absolument **ajouter un cinquième paramètre** générique qu'il faudra instancier correctement suivant le type des composants du vecteur et suivant le signe de l'opération. (page suivante).

**Remarque** encore : Dans la partie générique nous avons déclaré la fonction

```
with function "+" (X,Y : T_ITEM)  return  T_ITEM;
```

Dans ce cas le paramètre effectif "+" ne peut être omis au cours de l'opération d'instanciation bien que l'on instancie "+" avec "+". Le "*" ne peut être omis non plus mais cela est évident ! En écrivant maintenant (notez le **is** <> en plus) :

```
with function  "+" (X,Y: T_ITEM)  return T_ITEM is <>;
```

en plus!

alors le paramètre effectif "+" peut être omis au cours de l'opération d'instanciation si l'on souhaite faire des additions (mais il faudra mettre le "*" évidemment si l'on souhaite faire des produits) !

D'où les écritures :

```
generic
     type  T_INDICE  is  (<>);
     type  T_ITEM    is  private;
     type  T_VEC     is array  (T_INDICE  range  <> ) of T_ITEM;
     ELEM_NEUTRE : T_ITEM;
     with  function  "+"  (X,Y : T_ITEM)  return  T_ITEM is <>;
function OPERER (A : T_VEC) return  T_ITEM;
.................

function  OPERER  (A : T_VEC)  return  T_ITEM is
RESULTAT  :  T_ITEM  :=  ELEM_NEUTRE;
begin
     for  IND  in  A'RANGE  loop
          RESULTAT  :=  RESULTAT + A(IND);
     end  loop;
     return  RESULTAT;
end  OPERER;
```

Un paramètre de plus !

Ici c'est correct car le **private** admet l'affectation

Et quelques instanciations :

```
function ADD_V_ENTIER is new OPERER (T_JOUR,T_ENTIER,0,T_V……); -- + omis !
function ADD_V_REEL is new OPERER (T_JOUR,FLOAT,0.0,T_V……); -- + omis !

function PROD_V_REEL is new OPERER (T_JOUR,FLOAT,1.0,T_V……,"*");
function PROD_V_ENTIER is new OPERER (T_JOUR,T_ENTIER ,1,T_V……,"*");
```

**Remarque :** pour chaque instanciation le vecteur marqué ci-dessus T_V..... devra être déclaré comme compatible avec les paramètres effectifs qui le précèdent !

**Un autre exemple** de réalisation de corps (qui, celui-là, ne pose pas de problème!) :
La fonction INTEGRALE calcule l'intégrale d'une fonction entre deux valeurs A et B (admise plus haut). Voici une **réalisation possible** du corps. La fonction à intégrer G constitue un paramètre générique. Une **amélioration intéressante** de cette fonction sera vue en TP numérique.

```
generic
     type T_FLOTTANT is digits <>;
     with function  G (X : T_FLOTTANT) return T_FLOTTANT;
function  INTEGRALE (A,B : in T_FLOTTANT) return T_FLOTTANT;
……..
function  INTEGRALE (A,B : in T_FLOTTANT) return T_FLOTTANT is
     SOMME,DELTAX : T_FLOTTANT;
     NB_PAS : POSITIVE := 100;
begin
     SOMME := 0.0;
     DELTAX := (B-A)/T_FLOTTANT(NB_PAS);
     for IND  in  1..NB_PAS-1  loop
            SOMME := SOMME + G(A + T_FLOTTANT(IND) * DELTAX);
     end loop;
     return SOMME + DELTAX * (G(A) + G(B)) / 2.0 ;
end  INTEGRER;
```

**Remarque :**
Ada permet aussi les paramètres **génériques non contraints** ainsi :

```
type T_NON_CONTRAINT (<>) is private;
```

A revoir au cours n°11 avec les fichiers séquentiels !

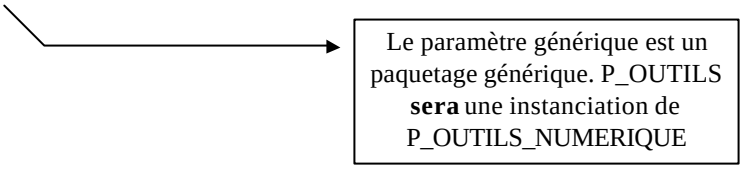
## Les paramètres génériques sous forme de paquetage (mais générique !).

Ada permet aussi un paquetage générique comme paramètre d'un autre paquetage générique. Ainsi :

```
generic
     type T_NUMERIQUE is private;
     ELEM_NEUTRE : T_NUMERIQUE;
     ..........
package P_OUTILS_NUMERIQUE is
....... contient quelques outils simples
end P_OUTILS_NUMERIQUE;
..............
```

on a définit ainsi un premier paquetage générique `P_OUTILS_NUMERIQUE` qui permet des opérations sur un type numérique « formel». On désire maintenant écrire un paquetage `P_MATRICE` (lui aussi générique) mais plus ambitieux qui utiliserait le premier paquetage `P_OUTILS_NUMERIQUE` déjà réalisé.

```
generic
.......
     with package P_OUTILS is new P_OUTILS_NUMERIQUE(<>);
.........
..........
package P_MATRICE is
........
end P_MATRICE;
```




avec `package P_OUTILS_FLOAT is new P_OUTILS_NUMERIQUE(FLOAT,0.0,....);`
on instancie d'abord le premier paquetage avec des `FLOAT`. Puis avec :

`package P_MATRICE_FLOAT is new P_MATRICE (.....,P_OUTILS_FLOAT,....);`
on instancie le deuxième paquetage. Notez qu'il faut instancier le deuxième avec, comme paramètre effectif, une instance du premier paquetage.

De même :
avec `package P_OUTILS_ENTIER is new P_OUTILS_NUMERIQUE(T_ENTIER,0,....);`
on peut instancier avec des entiers :
`package P_MATRICE_ENTIER is new P_MATRICE (.....,P_OUTILS_ENTIER,....);`

On a ainsi encore élargi la puissance de Ada pour la **réutilisation** par le biais de la généricité.

**Remarques :**

- On verra de belles applications de la généricité par la suite cours sur les E/S (cours n°11). et plus loin dans ce cours n°9 avec les sous paquetages génériques de `ADA.TEXT_IO` : `INTEGER_IO`, `FLOAT_IO`, `FIXED_IO`, `ENUMERATION_IO`, `MODULAR_IO`, `DECIMAL_IO`.

- Les paquetages de Ada qui sont très utiles (car génériques) sont à découvrir (polycopié ….) notamment: `Generic_Bounded_Length` sous paquetage de `Ada.Strings.Bounded` (vu en TD 12A et B) ainsi que pour générer des nombres aléatoires : `Ada.Numerics.Discrete_Random`. Et enfin pour tout ce qui est calculs numériques : `Ada.Numerics.Generic_Elementary_Functions` permettant les fonctions mathématiques classiques (cosinus, sinus, tangente, arc tangente etc.).

- Lancez vous ! Osez la généricité c'est-à-dire pensez (**dès la conception**) à la généricité vous récupérerez plus tard votre investissement !

## Les fils génériques.

La notion de hiérarchie de paquetages (vue au cours n°7) peut évidemment se conjuguer avec le concept de généricité. Puisque la généricité permet la construction de modules instanciables il était intéressant de pouvoir lui associer cette autre propriété de construction hiérarchique pour une puissance d'utilisation renforcée.

Tout paquetage père (même non générique) peut avoir des fils génériques.

- Si le père n'est pas générique alors un fils générique est instancié traditionnellement aux endroits où il est visible (avec **with**).
- Si le père est générique alors tous ses fils seront **obligatoirement génériques** et notés comme tels (même avec une liste vide de paramètres s'il ne faut pas en rajouter : vu en TD-TP). L'instanciation d'un fils peut avoir lieu à l'intérieur de la hiérarchie familiale sans problème. Si l'instanciation **est externe** (par un utilisateur par exemple) **alors ce ne peut être que d'une instance de son père** (voir en TD-TP).

Exemple (reprenons le paquetage P_COMPLEXE cours n°7) et **rendu générique** avec un type réel flottant :

```
generic
     type T_FLOTTANT is digits <> ;
package  P_COMPLEXE  is
 type  T_NOMBRE_COMPLEXE  is private;
 function  "+" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
 function  "-" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
 function  "*" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
 function  "/" (C1,C2 : T_NOMBRE_COMPLEXE) return T_NOMBRE_COMPLEXE;
 procedure LIRE (C : out T_ NOMBRE_COMPLEXE);
 procedure ECRIRE (C : in T_ NOMBRE_COMPLEXE);
private
     type  T_NOMBRE_COMPLEXE  is
     record
          PARTIE_REELLE  : T_FLOTTANT;
          PARTIE_IMAGINAIRE  : T_FLOTTANT;
     end record;
end  P_COMPLEXE;
```

l'utilisation avec le type prédéfini FLOAT est facile (rappel) :

```
package P_COMPLEXE_FLOAT is new P_COMPLEXE(FLOAT) ;
```

et c'est tout.

Si nous souhaitons améliorer, pour une utilisation plus pointue, le paquetage P_COMPLEXE **on écrit un fils** :

```
generic
```

Liste générique vide dans ce cas ! Rien à rajouter!

```
package P_COMPLEXE.ELARGI is
function Conjuguée (X : T_Complexe) return T_Complexe;
function Argument  (X : T_Complexe) return T_FLOTTANT;
function Module    (X : T_Complexe) return T_FLOTTANT;
end P_COMPLEXE.ELARGI;
```

Si l'on souhaite utiliser une instance de ce paquetage fils P_COMPLEXE.ELARGI il faut absolument **d'abord** instancier le père P_COMPLEXE par exemple avec le type FLOAT :

```
package P_COMPLEXE_FLOAT is new P_COMPLEXE(FLOAT) ;
```

**puis** instancier le fils générique **qui se nomme maintenant** P_COMPLEXE_FLOAT.ELARGI (notez cela et comprenez le) par exemple ainsi :

```
package P_ELARGI_FLOAT is new P_COMPLEXE_FLOAT.ELARGI ;
```

Notez l'instanciation sans paramètre effectif puisqu'il n'y a pas de nouveau paramètre formel. Si la liste n'avait pas été vide on aurait écrit :

```
package P_ELARGI_FLOAT is new P_COMPLEXE_FLOAT.ELARGI(param_effectifs) ;
```

## Application de la généricité.

**Retour sur `ADA.TEXT_IO`** (voir cours 5 bis et ce n'est pas fini!)

Des rudiments d'entrées-sorties simples (puisés dans le paquetage `ADA.TEXT_IO`) ont déjà été évoqués dans le cours n° 5 bis (pour ne plus utiliser `P_E_SORTIE`). Nous allons compléter nos connaissances sur ce paquetage en étudiant les sous paquetages qui composent l'essentiel de son corps. Ces 6 paquetages : `INTEGER_IO`, `FLOAT_IO`, `FIXED_IO`, `ENUMERATION_IO`, `DECIMAL_IO` et `MODULAR_IO` ont la propriété d'être génériques. Cependant nous ne nous intéresserons ici qu'aux sous-programmes qui permettent les échanges clavier écran seulement. Les E/S sur fichiers «textes » puis dans d'autres paquetages autres que `ADA.TEXT_IO` sur fichiers « séquentiels » et enfin sur fichiers « directs » sont étudiés dans un autre support. Il est conseillé à ceux qui le peuvent d'éditer l'annexe A.10.1 du manuel de référence pour plus de lisibilité.

**Le sous-paquetage `INTEGER_IO`.**

Après les sous-programmes non génériques on trouve (polycopié « paquetages… ») dans le paquetage `ADA.TEXT_IO` les lignes suivantes :

```
generic
     type NUM is range <>;
package INTEGER_IO is
DEFAULT_WIDTH : FIELD := NUM'WIDTH;
DEFAULT_BASE : NUMBER_BASE := 10;
.....
procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
.....
procedure PUT(ITEM  : in NUM;
              WIDTH : in FIELD := DEFAULT_WIDTH;
              BASE  : in NUMBER_BASE := DEFAULT_BASE);
procedure GET(FROM  : in STRING; ITEM : out NUM; LAST : out POSITIVE);
procedure PUT(TO    : out STRING;
              ITEM  : in NUM; BASE : in NUMBER_BASE := DEFAULT_BASE);
end INTEGER_IO;
```

Les deux lignes marquées `...` concernent les GET et PUT sur fichiers textes.

Les deux premières procédures ci-dessus permettent : des saisies clavier (avec `GET`) ou des affichages écran (avec `PUT`) d'instances de type générique `NUM` (« type entier » à cause du **`range <>`**). Les **deux dernières** (**intéressantes**) permettent d'extraire l'entier `ITEM` d'un `STRING` (avec `GET`) ou de convertir en caractères dans un `STRING` l'entier `ITEM` (avec `PUT`) elles ne sont pas utilisées en général et **c'est bien dommage** !

Aussi après avoir instancié un paquetage « vrai » avec un type entier (`T_COMPTEUR` par exemple) on peut mettre en œuvre les E/S sur ce type.Exemple après :

```
package E_S_COMPTEUR is new ADA.TEXT_IO.INTEGER_IO(T_COMPTEUR);
```

les opérations :

`E_S_COMPTEUR.GET` et `E_S_COMPTEUR.PUT` sont disponibles.

**Remarques :**

- On peut utiliser `GET` et `PUT` sans préfixer si l'on a utilisé la clause **`use`** `E_S_COMPTEUR`.
- La procédure `GET` (clavier) n'est pas validée pour autant et il est préférable de l'inclure dans une boucle **`loop`** avec traitement d'exception et nous voilà ramené à notre vieille connaissance de lecture validée d'un type discret qui est « béton ». Vue cours n°1 (paquetage `P_E_SORTIE`), cours n°3 (instruction bloc), cours n°5 (procédure `LIRE`), cours n°5 bis (E/S simples), cours n°8 (exception) etc.
- `PUT` est intéressant quand on utilise son « formatage ».
- `GET` et `PUT` dans un `String` sont très intéressantes (à voir).

**Le paquetage ENUMERATION_IO.**

Dans le même ordre d'idée on trouve aussi dans le paquetage ADA.TEXT_IO (à la fin) le sous-paquetage ENUMERATION_IO. Soit :

```
generic
     type ENUM is (<>);
package ENUMERATION_IO is
DEFAULT_WIDTH : FIELD := ENUM'WIDTH;
DEFAULT_SETTING : TYPE_SET := UPPER_CASE;
.....
procedure GET(ITEM : out ENUM);
.....
procedure PUT(ITEM  : in ENUM;
              WIDTH : in FIELD := DEFAULT_WIDTH;
              SET   : in TYPE_SET := DEFAULT_SETTING);
procedure GET(FROM  : in STRING; ITEM : out ENUM; LAST : out POSITIVE);
procedure PUT(TO    : out STRING;
              ITEM  : in ENUM; SET : in TYPE_SET := DEFAULT_SETTING);
end ENUMERATION_IO;
```

**Remarques :**

- Il faut instancier avec un type énumératif (par exemple T_JOUR).
- GET ne nous dispense pas de la validation et on peut lui préférer notre vieille connaissance de lecture de type discret (car les énumératifs sont discrets)!
- Seules en fait, comme précédemment, les deux dernières procédures en relation avec les STRING sont intéressantes. Ainsi que le premier PUT pour son formatage.

**Le paquetage FLOAT_IO.**

Ce paquetage générique permet des entrées-sorties sur tout type réel (virgule flottante ou digits) construit ou prédéfini (comme FLOAT). Ces types numériques seront étudiés prochainement (après le cours n° 11).
FORE signifie partie entière du nombre. AFT signifie partie décimale (après le point décimal). EXP (non nul) implique la notation scientifique normalisée. EXP (nul) implique l'écriture « normale » non scientifique.

```
generic
     type NUM is digits <>;
package FLOAT_IO is
DEFAULT_FORE : FIELD := 2;
DEFAULT_AFT  : FIELD := NUM'DIGITS - 1;
DEFAULT_EXP  : FIELD := 3;
.....
procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
.....
procedure PUT(ITEM : in NUM;
              FORE : in FIELD := DEFAULT_FORE;
              AFT  : in FIELD := DEFAULT_AFT;
              EXP  : in FIELD := DEFAULT_EXP);
procedure GET(FROM : in STRING; ITEM : out NUM; LAST : out POSITIVE);
procedure PUT(TO   : out STRING;
              ITEM : in NUM;
              AFT  : in FIELD := DEFAULT_AFT;
              EXP  : in FIELD := DEFAULT_EXP);
end FLOAT_IO;
```

**Le paquetage FIXED_IO.**

Ce paquetage générique permet des entrées-sorties sur tout type réel (virgule fixe ou **delta**) construit ou prédéfini (comme DURATION). Ces types numériques seront étudiés plus tard en même temps que les réels à virgule flottante (**digits**).
FORE signifie partie entière du nombre. AFT signifie partie fractionnaire (après le point décimal). EXP (non nul) implique notation scientifique normalisée. EXP (nul) implique l'écriture « normale » non scientifique.

```
generic
     type NUM is delta <>;
package FLOAT_IO is
DEFAULT_FORE : FIELD := NUM'FORE;
DEFAULT_AFT  : FIELD := NUM'AFT;
DEFAULT_EXP  : FIELD := 0;
.....
procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
.....
procedure PUT(ITEM : in NUM;
              FORE : in FIELD := DEFAULT_FORE;
              AFT  : in FIELD := DEFAULT_AFT;
              EXP  : in FIELD := DEFAULT_EXP);
procedure GET(FROM : in STRING; ITEM : out NUM; LAST : out POSITIVE);
procedure PUT(TO   : out STRING;
              ITEM : in NUM;
              AFT  : in FIELD := DEFAULT_AFT;
              EXP  : in FIELD := DEFAULT_EXP);
end FLOAT_IO;
```

**Le paquetage MODULAR_IO.**

```
generic
     type NUM is mod <>;
package MODULAR_IO is
DEFAULT_WIDTH : FIELD := NUM'WIDTH;
DEFAULT_BASE : NUMBER_BASE := 10;
.....
procedure GET(ITEM : out NUM; WIDTH : in FIELD := 0);
.....
procedure PUT(ITEM  : in NUM;
              WIDTH : in FIELD := DEFAULT_WIDTH;
              BASE  : in NUMBER_BASE := DEFAULT_BASE);
procedure GET(FROM  : in STRING; ITEM : out NUM; LAST : out POSITIVE);
procedure PUT(TO    : out STRING;
              ITEM  : in NUM; BASE : in NUMBER_BASE := DEFAULT_BASE);
end MODULAR_IO;
```

totalement identique à `INTEGER_IO` mais, bien sûr, il traite des entiers modulaires.

Le paquetage `DECIMAL_IO` (identique à `FIXED_IO`) sera étudié avec un cours associé il utilise le type dit décimal ou delta-digits.

**Remarque** : On a recommandé l'utilisation de la lecture `Get` et l'écriture `Put` d'un numérique par le biais d'un `String`. L'intérêt ne paraît pas toujours évident. Voici peut être un exemple convaincant :

```
function Jour_Courant return String is
   Top : Time := Clock;
   La_Date : String (1..10) := "  /  /    " ;
begin
   Put (La_Date(1..2), Day(Top)) ;
   Put (La_Date(4..5), Month(Top)) ;
   Put (La_Date(7..10), Year(Top)) ;
   return La_Date;
end Jour_Courant;
```

Bien sûr il faut avoir évoqué le paquetage `Calendar` (pour `Time`, `Clock`, `Day`, `Month` et `Year`) et il faut avoir instancié un paquetage vrai de `Integer_IO` pour faire les `Put` !

Pour le `Get` imaginez des lectures de plusieurs valeurs numériques dans une ligne de texte.

# Je retiens n°3

**Quelques termes, informations ou conseils à retenir après le cours n° 9 Ada (semaines 5, 6).**

Cette fiche fait suite aux fiches « je retiens n°1 et n°2 » vues précédemment.

- Le type article est, comme le type tableau, un type **composite** (ou composé). Par contre les composants d'un type article ne sont pas forcément du même type alors qu'ils le sont forcément pour les tableaux.
- Les composants d'un tableau sont repérés et sélectionnables grâce à un indice de type discret. Les composants d'un type article sont repérables et sélectionnables grâce à des identificateurs de champ.
- Si les tableaux pouvaient être anonymes ou « muets » (mais ce n'est pas recommandé !) les articles ne le peuvent pas.
- Un tableau ne peut pas être anonyme s'il est composant d'article.
- Comme pour les tableaux le mécanisme de passage de paramètres d'un type article n'est pas défini par la norme.
- Si un type article n'a pas de discriminant avec une «expression par défaut » alors tous les objets (ou instances) de ce type doivent être contraints.
- Le discriminant d'un article non contraint ne peut être modifié que par affectation globale de tout l'article.
- Toute partie variante d'article doit apparaître comme dernier composant d'article (avec **`case`**).
- Les discriminants servent souvent pour gérer des bornes variables de tableaux ou pour gérer des parties variantes.
- Les variables déclarées dans la partie spécifications d'un paquetage sont des variables globales très persistantes!
- Une unité de bibliothèque doit être compilée après toutes les unités évoquées avec **`with`** (évident !).
- Un corps (**`body`**) de paquetage doit être compilé seulement après la spécification correspondante !
- Une spécification de paquetage et son corps forment un tout logique indissociable.
- Ne pas confondre les attributs `LENGTH` et `LAST`. Pour que leur valeur soit identique il faut que le type d'indice du tableau soit `NATURAL` et que l'attribut `FIRST` donne 1. Pas forcément courant!
- Les sous programmes (procédure et fonction) génériques ont toujours une spécification et un corps **séparés**.
- Soignez les modes des paramètres formels : **`in`**, **`out`** ou **`in out`**. Chacun exprime une « sémantique » importante il faut la respecter.
- Prescrire les variables globales systématiquement. Sinon danger!
- Pour les procédures avec paramètres formels **`out`** et pour les fonctions il est recommandé de créer une variable locale jouant le rôle du résultat dans le corps du sous programme. Le transfert final s'effectue à la fin seulement.
- `PUT_LINE` et `GET_LINE` (de `Text_Io`) sont uniquement réservés aux objets de type `STRING`.
- Les sous-programmes génériques ne peuvent pas être surchargés. Mais leurs instances peuvent l'être.
- On ne peut utiliser **`use`** avec un paquetage générique. Un peu de réflexion nous démontre pourquoi !
- Le mode **`out`** n'existe pas pour les paramètres génériques. Seulement **`in`** ou **`in out`**.
- Attention aux variables non initialisées.
- Pour éviter les `CONSTRAINT_ERROR` réfléchir aux contraintes d'indice sur les tableaux, ainsi qu'aux valeurs extrêmes des objets discrets (entiers et énumératifs).
- Les exceptions paraissent simples à comprendre .. Attention piège ! Ne pas en abuser.
- Il est préférable de gérer ses propres exceptions. Evitez les prédéfinies. Recourir au if quand c'est possible plutôt qu'aux levées d'exceptions.
- Le traite exception mérite une attention soignée. Attention aux paramètres **`out`** ou **`in out`** quand une exception est traitée. Même problème dans une fonction.
- Editez les fichiers chaine.doc, rationnel.doc, pragma.doc.
- Etudiez avec AdaGide (ou le polycopié « paquetages ») : `Ada.Exceptions`, `Ada.Strings.Bounded`, `Ada.Io_Exception`.