Université de Monastir
Institut Supérieur d'Informatique et de Mathématiques

Cours: Programmation Temps Réel et Concurrente

Filière: MP2-GL

Chapitre 3: Les sous-programmes

Réalisé par:
Dr. Sakka Rouis Taoufik

1

I. Introduction

En Ada on peut distingue deux types de sous-programmes: Les procédures et les fonctions

Si un sous-programme A a besoin d'un sous-programme B, alors il est nécessaire que B soit déclaré avant A, sinon le compilateur ne pourra le voir.

2

II. Procédure

Syntaxe: Procédure non parametrée

```
procedure Nom_De_Votre_Procedure is
   --Partie pour déclarer les variables

begin

   --Partie exécutée par le programme
end Nom_De_Votre_Procedure ;
```

Syntaxe: Procédure parametrée

```
procedure Nom_De_Votre_Procedure ( liste de paramètres)  is
   --Partie pour déclarer les variables
begin

   --Partie exécutée par le programme
end Nom_De_Votre_Procedure ;
```


II. Procédure

Exemple 1: Procédure non parametrée

```
with Ada.Text_IO, Ada.Integer_Text_IO ;
use Ada.Text_IO, Ada.Integer_Text_IO ;
procedure Figure is
   Procedure Affich_Ligne is
   begin
      for j in 1..5 loop
         put('#') ;
      end loop ;
      new_line ;
   end Affich_Ligne ;
   nb_lignes : natural ;

begin
   Put("Combien de lignes voulez-vous dessiner ?") ;
   get(nb_lignes ) ; skip_line ;
   for i in 1..nb_lignes  loop
      Affich_Ligne;
   end loop ;
end Figure ;
```

II. Procédure

Exemple 2: Procédure paramétrée

```
with Ada.Text_IO, Ada.Integer_Text_IO ;
use Ada.Text_IO, Ada.Integer_Text_IO ;
procedure Figure is
   Procedure Affich_Ligne (nb : natural) is
   begin
      for j in 1..nb loop
         put('#') ;
      end loop ;   new_line ;
   end Affich_Ligne ;
   nb_lignes : natural ;
begin
   Put("Combien de lignes voulez-vous dessiner ?") ;
   get (nb_lignes ) ; skip_line ;
   Put("Combien de colonnes voulez-vous dessiner ? ") ;
   get(nb_colonnes ) ; skip_line ;
   for i in 1..nb_lignes  loop
      Affich_Ligne(nb_colonnes);
   end loop ;
 end Figure ;
```


II. Procédure

Exemple 2: Procédure avec plusieurs paramètres

```
Procedure Affich_Rect (nb_lignes : natural ; nb_colonnes : natural) is
begin
   for i in 1..nb_lignes  loop
      Affich_Ligne(nb_colonnes);
   end loop ;
end Affich_Rect ;
```

Remarque: en Ada chaque paramètre peut être de type **In** (par défuat), **out** ou **In Out**

Exemples:

```
procedure f1 (n: in out natural) is ….
procedure f2 (n1: natural; n2: in out natural ; n3: out natural ) is ….
```

III. Fonction

Exemple 1:

```
function A_Rect (larg : natural ; long : natural) return natural is
   A : natural ;
begin
   A:= larg * long ;
   return A ;
end A_Rect ;
```

Exemple 2:

```
fonction exemple (a,b: natural; x,y: float:=1.0)  return float   is
begin
…
Return …

end exemple;
```


III. La généricité

Ada est un des premiers langages qui implémente la généricité.

Toute unité de programme (fonction, procédure, paquetage) peut être générique et une unité générique peut elle-même inclure une autre unité générique.

Une unité générique est décrite au moyen de types génériques qui sont spécifiés lors de l'instanciation en une unité effective. Et elle est utilisable comme si elle avait été programmée "normalement".

III. La généricité

Les paramètres génériques formels peuvent être :

- une variable (paramètre en mode **in, ou** ou **in out**) ou une constante (paramètre en mode **in**) ;
- un type pour lequel il est possible de spécifier une forme : type énuméré, entier, décimal, tableau, avec ou sans discriminant, et même **private** ou **limited private** (Limited ➔ l'utilisation de l'affectation et des opérateurs d'égalité et d'inégalité sur des objets de ce type **est interdite a ce niveaux)**. Ces hypothèses permettent de déterminer les opérations qui pourront être utilisées à l'intérieur du type générique ainsi que la conformité de l'instanciation ;
- une procédure ou une fonction dont le prototype est précisé ;
- un paquetage

III. La généricité

Création d'un type générique

Nous allons donc créer un type générique appelé T_Entier. Attention, ce type n'existera pas réellement, il ne servira qu'à la réalisation d'une procédure générique (et une seule). Pour cela nous allons devoir ouvrir un bloc GENERIC dans la partie réservée aux déclarations :

```
generic
   type T_Entier is range <> ;
```

Notez bien que La combinaison de RANGE et du diamant indique que les informations nécessaires ne seront transmises que plus tard. Plus précisément indique que le type attendu est un type entier (Integer, Natural, Positive, Long_Long_integer…) et pas flottant ou discret ou que-sais-je encore !

Notez également qu'il n'y a pas d'instruction « END GENERIC »

III. La généricité

Création d'une procédure générique

Comme nous l'avions dit précédemment, la déclaration du type T_Entier doit être **immédiatement** suivie de la spécification de la procédure générique, ce qui nous donnera le code suivant :

Exemple 1:

```
generic
  type T is private;
  procedure echange  (x, y : in out T);
  procedure echange  (x, y : in out T) is
            Z: T:= x;
   Begin
            x:= y; y:= z;
   End echange;

  Procedure echange_integer is new echange (integer);

  Procedure echange_float is new echange (float);
```


III. La généricité

Exemple 2:

```
generic
  type Element is private; -- private ➔ type pour lequel "=" et ":=" sont definis
  type Tab is array (Positive range <>) of Element;
  with function "<" (E1, E2 : in Element) return Boolean;
 procedure Trier (T : in out Tab);

procedure Trier (T : in out Tab) is
begin
    -- corps de la procedure s'appuyant sur  "=", ":=" et "<"
            …
            …
 end Trier;
 -- Instanciation pour le tri d'un tableau d'entiers
 type Tab_Entiers is array (Positive range <>) of Integer;
procedure Trier_Entiers is new Trier (Integer, Tab_Entiers, "<");
```

NB. La procédure Trier possède **trois paramètres génériques formels liés entre eux**

III. La généricité

Paramètre de type programme

```
package P_Point is

   type T_Point is private ;

   procedure set_x(P : out T_Point ; x : in float) ;
   procedure set_y(P : out T_Point ; y : in float) ;
   function get_x(P : in T_Point) return float ;
   function get_y(P : in T_Point) return float ;
   procedure put(P : in T_Point) ;
   procedure copy(From : in T_Point ; To : out T_Point) ;

private
   type T_Point is record
      x,y : Float ;
   end record ;
 …  --implementation des codes des methods du package
end P_Point ;
```

13

III. La généricité

```
generic
   type T_Element is limited private ;
   with procedure copier_A_vers_B (a : in T_Element ; b : out T_Element) ;
   procedure Generic_Swap (a,b : in out T_Element) ;


procedure Generic_Swap (a,b : in out T_Element) is
   c : T_Element ;
begin
   copier_A_vers_B (b,c) ;
   copier_A_vers_B (a,b) ;
   copier_A_vers_B (c,a) ;
end generic_swap ;

procedure swap is new generic_swap (T_Point, copy) ;
     --   OU BIEN
procedure swap is new generic_swap(T_Element       => T_Point,
                                   Copier_A_vers_B => copy) ;
```

14

IV. Exercices d'application

Exercice 1 :
Écrire en Ada un algorithme d'une **procédure** qui permet de calculer et de retourner la valeur absolue et le carré d'un réel passé en paramètre.

Exercice 2 :
Écrire en Ada un algorithme d'une fonction Triangle qui permet de vérifier si les 3 nombres a, b et c peuvent être les mesures des côtés d'un triangle rectangle.
Remarque: D'après le théorème de Pythagore, si a, b et c sont les mesures des côtés d'un rectangle, alors $a^2= b^2+c^2$ ou $b^2=a^2+c^2$ ou $c^2 =a^2+b^2$

Exercice 3:
Ecrire un programme en Ada qui lit deux nombre naturel non nul *m et n* et qui détermine s'ils sont amis. Deux nombres entiers n et m sont qualifiés d'amis, si la somme des diviseurs de n est égale à m et la somme des diviseurs de m est égale à n (on ne compte pas comme diviseur le nombre lui-même et 1). Proposer une solution modulaire.

Chapitre 3: Les sous-programmes

IV. Exercices d'application

Exercice 4 :
Réaliser en Ada un algorithme d'une fonction qui recherche le premier nombre entier naturel dont le carré se termine par n fois le même chiffre.
Exemple : pour n = 2, le résultat est 10 car 100 se termine par 2 fois le même chiffre.

Exercice 5:
L'algorithme de recherche séquentielle (ou linéaire) consiste à examiner la table éléments par éléments et voir si **info** appartient ou non à la table **T**. Si le résultat est positif (**info appartient à T**) alors cet algorithme retourne l'indice de la première occurrence de l'**info**, sinon il retourne -1.

Réaliser en Ada une fonction générique pour la fonction RechercheSequentielle
Proposer une instance de cette fonction pour les vaeurs de types natural et une autre insatance pour les réels.