## Paquetages (1/3)

- La notion de sous-programme ne suffit pas à structurer le code dans les logiciels modernes où les lignes de code se comptent en dizaines ou centaines de milliers (voire millions!). Nécessité de davantage de modularité.
- ADA introduit la notion de paquetage qui se rapproche des *modules* du langage Modula, des *unités* du Turbo-Pascal ou des *classes* des langages à objet (notion postérieure à ADA83).
- Typiquement, un paquetage contient la définition d'un type et les sous-programmes associés à ce type définissant et implémentant les opérations sur le type. Un paquetage n'est pas un fourre-tout!!!
- *Remarque : les paquetages en ADA ne sont pas de même nature que les paquetages en Java!*



## Paquetages (2/3)

- Différences entre sous-programmes et paquetages :
  - les paquetages permettent de cacher certains morceaux de code (déclarations, sous-programme) mais aussi d'en rendre certains accessibles de l'extérieur alors que les sous-programmes cachent tout, les variables locales comme les sous-sous-programmes. Un paquetage permet donc différents degrés d'encapsulation.
  - les paquetages ne sont pas exécutable en tant que tel même s'ils contiennent du code exécutable, ce sont essentiellement des déclarations (de types et de sous-programmes).
- Différences entre paquetages et classes :
  - un paquetage ne peut être hérité
- ADA95 implémente l'héritage par extension des types tagués (**`tagged`**) et permet donc de faire de la POO.

## Paquetages (3/3)

- Un paquetage est constitué *obligatoirement* d'une partie spécification où sont déclarés les types, variables, constantes, exceptions, profils des sous-programmes. Ces spécifications sont publiques par défaut mais peuvent être déclarées partiellement ou totalement privées (invisibles en dehors du paquetage).
- Si nécessaire (c'est-à-dire si des sous-programmes sont déclarés), un corps de paquetage peut être ajouté pour y écrire le corps des sous-programmes. Tout ce qui apparait dans ce corps est *privé*.
- Utiliser un paquetage signifie utiliser les spécifications publiques uniquement. Le compilateur ne se réfère alors qu'à la partie spécification pour contrôler le bon usage du paquetage par un autre programme. Cette séparation incite à **prototyper sans penser implémentation.**



## Spécification de paquetage (1/2)

```
package_specification ::= package defining_program_unit_name is
                            { basic_declarative_item }
                            [ private { basic_declarative_item } ]
                          end [ [ parent_unit_name. ] identifier ] ;
```

- Un paquetage peut être déclaré dans toute partie déclarative d'un programme (le corps d'un paquetage doit alors être écrit dans la même partie déclarative).
- Il peut aussi être écrit comme unité de bibliothèque pour compilation séparée dans un fichier de spécification avec l'extension *.ads* (ADA Spécification).

# Spécification de paquetage (2/2)

- *Exemple* :

```
package Rational_Numbers is

    type Rational is record
          Numerator   : Integer;
          Denominator : Positive;
    end record;

    function "="(X,Y : Rational) return Boolean;

    --  to construct a rational number
    function "/"  (X,Y : Integer)  return Rational;

    function "+"  (X,Y : Rational) return Rational;

    function "-"  (X,Y : Rational) return Rational;

    function "*"  (X,Y : Rational) return Rational;

    function "/"  (X,Y : Rational) return Rational;

    procedure affiche(r : Rational);

end Rational_Numbers;
```



# Corps de paquetage (1/2)

```
package_body ::= package body defining_program_unit_name is
                    declarative_part
                 [ begin
                     handled_sequence_of_statements]
                 end [ [ parent_unit_name . ] identifier ] ;
```

- Un corps de paquetage a une partie déclarative où sont déclarés les corps des sous-programmes de sa spécification. On peut ajouter une partie initialisation après le mot-clé **`begin`**. Cela permet par exemple d'initialiser des variables du paquetage dont la valeur initiale n'est connue qu'après l'exécution d'un sous-programme.

- Un corps de paquetage est écrit dans la même partie déclarative où apparait sa spécification. Si sa spécification est écrite comme unité de bibliothèque, le corps de paquetage (ou un programme principal) doit être écrit dans un fichier propre avec l'extension *.adb* (ADA Body).

# Corps de paquetage (2/2)

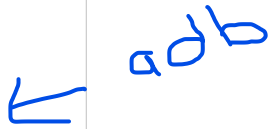
```
package body Rational_Numbers is

   function "="(X,Y : Rational) return Boolean is
         U : Rational := X;
         V : Rational := Y;
      begin
         Same_Denominator (U,V);
         return U.Numerator = V.Numerator;
      end "=";

   function "/" (X,Y : Integer) return Rational is
      begin
         if Y > 0 then
            return (Numerator => X,  Denominator => Y);
         else
            return (Numerator => -X, Denominator => -Y);
         end if;
      end "/";

   ...

end Rational_Numbers;
```






# Utilisation des paquetages (1/3)

- Les unités de bibliothèque sont invoquées par des clauses de contexte :
  - **`with`** indique que l'unité va utiliser des déclarations d'une autre unité (seules les déclarations visibles sont utilisables!)
  - **`use`** permet de ne pas préfixer les identificateurs de l'unité importée. Une clause **`use`** doit apparaitre après la clause **`with`** correspondante
  - **`use type`** permet de ne rendre visible que les fonctions-opérateurs sur un type donné

- Les clauses **`use`** et **`use type`** obéissent aux règles suivantes :
  - La clause **`use`** ne rend pas visible un identificateur si le même identificateur est déjà visible dans une déclaration et qu'il n'y a pas de surcharge possible
  - Si une clause **`use`** peut rendre visibles des identificateurs déjà rendus visibles par une clause **`use`**, et que la surcharge n'est pas possible, aucun des identificateurs en question n'est visible

## Utilisation des paquetages (2/3)

- *Exemples* :

```
with Ada.Text_IO;with Ada.Integer_Text_IO;use Ada.Text_IO;use Ada.Integer_Text_IO;

procedure UseExample is

package Truc is
   a : Integer := 1;
   b : Integer := 2;
end Truc;

package Bidule is
   a : Integer := 3;
   b : Integer := 4;
end Bidule;

use Truc;use Bidule;

a : Integer := 5;

begin
   put(a);
   -- put(b); inacceptable par le compilateur (b invisible)
end UseExample;
```



## Utilisation des paquetages (3/3)

- La clause use type appliquée à un type permet de rendre visible sans préfixe les fonctions-opérateurs déclarées dans la même spécification que le type

- *Exemple* :

```
use type Rational_Numbers.Rational;
```

- Renommage d'un paquetage :

```
package Machin renames Truc;
```

# Paquetage local (1/3)

```
with Ada.Text_IO;use Ada.Text_IO;with Ada.Integer_Text_IO;use Ada.Integer_Text_IO;

procedure Essai is

package Paquetage_Liste is

type Element;
type Liste is access all Element;
type Element is record
  suivant:Liste;
  valeur:Integer;
end record;

liste_vide : constant Liste := null;

procedure affiche(l : in Liste);

procedure addElement(l : in out Liste; i : in Integer);

procedure removeElement(l : in out Liste; i : in Integer);

end Paquetage_Liste;

...
```



# Paquetage local (2/3)

```
...

package body Paquetage_Liste is

procedure affiche(l : in Liste) is

lTemp : Liste :=l;

begin
   while lTemp /= null loop
        put(lTemp.valeur);
        new_line;
        lTemp := lTemp.suivant;
   end loop;
end affiche;

procedure addElement(l : in out Liste; i : in Integer) is
...
end addElement;

procedure removeElement(l : in out Liste; i : in Integer) is
...
end removeElement;

end Paquetage_Liste;

...
```

## Paquetage local (3/3)

```
...

use Paquetage_Liste;

e : Liste := new Element'(null,5);
f : Liste := new Element'(e,4);
g : Liste := new Element'(f,3);
h : Liste := new Element'(g,2);
i : Liste := new Element'(h,1);

begin
   affiche(i);
   addElement(i,6);
   affiche(i);
end Essai;
```



## Compilation séparée (1/3)

```
with Ada.Text_IO;with Ada.Integer_Text_IO;use Ada.Text_IO;use Ada.Integer_Text_IO;

package Paquetage_Liste is

type Element;
type Liste is access all Element;
type Element is record
  suivant:Liste;
  valeur:Integer;
end record;

liste_vide : constant Liste := null;

procedure affiche(l : in Liste);

procedure addElement(l : in out Liste; i : in Integer);

procedure removeElement(l : in out Liste; i : in Integer);

end Paquetage_Liste;
```

# Compilation séparée (2/3)

```
with Ada.Text_IO;with Ada.Integer_Text_IO;use Ada.Text_IO;use
   Ada.Integer_Text_IO;

package body Paquetage_Liste is

procedure affiche(l : in Liste) is

lTemp : Liste :=l;

begin
   while lTemp /= null loop
        put(lTemp.valeur);
        new_line;
        lTemp := lTemp.suivant;
   end loop;
end affiche;

procedure addElement(l : in out Liste; i : in Integer) is
...
end addElement;

procedure removeElement(l : in out Liste; i : in Integer) is
...
end removeElement;

end Paquetage_Liste;
```



# Compilation séparée (3/3)

```
with Ada.Text_IO;with Ada.Integer_Text_IO;use Ada.Text_IO;use
   Ada.Integer_Text_IO;

with Paquetage_Liste; use Paquetage_Liste;

procedure Test is

e : Liste := new Element'(null,5);
f : Liste := new Element'(e,4);
g : Liste := new Element'(f,3);
h : Liste := new Element'(g,2);
i : Liste := new Element'(h,1);

begin
   affiche(i);
   addElement(i,6);
   affiche(i);
end;
```

## Compilation séparée

- Il est possible à l'aide de la directive **`separate`** de séparer le corps d'un sous-programme du reste d'une unité de bibliothèque pour compilation séparée (utile quand une procédure est très grosse).

- *Exemple* :

```
package Paquetage_Liste is

...

procedure removeElement(l : in out Liste; i : in Integer) is separate;

end Paquetage_Liste;
```

```
separate(Paquetage_Liste)

procedure removeElement(l : in out Liste; i : in Integer) is
...
begin
...
end removeElement;
```



## Types privés (1/5)

- L'encapsulation dans les paquetages existe :
  - au niveau des corps de sous-programmes : ces corps, écrits dans le corps du paquetage, sont inaccessibles de l'extérieur
  - au niveau des types et variables déclarés : il faut pour cela les déclarer privés
- Les déclarations de types ou variables privés sont faites dans un bloc **`private`**
- Les types utilisés dans la partie publique mais dont on choisit de cacher la déclaration complète doivent être déclarés **`private`** dans la partie publique et définis complétement dans la partie privée
- Une constante d'un type privé est différée :
  - son identifiant et son type sont déclarés en partie publique
  - sa valeur est déclarée dans la partie privée, une fois le type complètement déclaré

# Types privés (2/5)

- Le corps d'un paquetage a accès à toutes les déclarations du paquetage, y compris celles du bloc private
- De l'extérieur du paquetage, seuls les types publics et sous-programmes publics sont accessibles. On peut cependant utiliser les types privés déclarés en partie publique, mais sans avoir accès à leur structure interne.
- Il est possible d'encapsuler des sous-programmes :
  - le sous-programme est déclaré dans la partie **`private`** de la déclaration du paquetage
  - le corps du sous-programme est défini dans le corps du paquetage
  - le sous-programme n'est accessible que dans le paquetage



# Types privés (3/5)

```
package Paquetage_Liste is

type Liste is private; -- type privé dont seule la déclaration est cachée

liste_vide : constant Liste; -- constante différée à valeur cachée

procedure affiche(l : in Liste);

procedure addElement(l : in out Liste; i : in Integer);

procedure removeElement(l : in out Liste; i : in Integer);

private

type Element; -- type complétement caché
type Liste is access all Element;
type Element is record
  suivant:Liste;
  valeur:Integer;
end record;

liste_vide : constant Liste := null;

procedure coucou; -- procédure privée

end Paquetage_Liste;
```

# Types privés (4/5)

```
package body Paquetage_Liste is

procedure affiche(l : in Liste) is

lTemp : Liste :=l;

begin
   while lTemp /= null loop
        put(lTemp.valeur);
        new_line;
        lTemp := lTemp.suivant;
   end loop;
   coucou;
end affiche;

procedure addElement(l : in out Liste; i : in Integer) is
...
end addElement;

procedure removeElement(l : in out Liste; i : in Integer) is
...
end removeElement;

procedure coucou is begin put("coucou"); end;

end Paquetage_Liste;
```



# Types privés (5/5)

```
with Ada.Text_IO;with Ada.Integer_Text_IO;
use Ada.Text_IO;use Ada.Integer_Text_IO;

with Paquetage_Liste; use Paquetage_Liste;

procedure Test is

l : Liste := liste_vide;

begin
   addElement(l,1);
          addElement(l,2);
   addElement(l,3);
   addElement(l,4);
   addElement(l,5);
   affiche(l);
   addElement(l,6);
   affiche(l);
   put_line("bonjour" & " aurevoir");
end;
```

# Types privés limités (1/2)

- La déclaration d'un type privé et donc sa structure est cachée. Il est possible d'aller plus loin dans l'encapsulation en déclarant un type privé limité :
  - l'affectation est interdite sur ce type, ainsi que l'égalité et l'inégalité
  - utiliser ce type passe **obligatoirement** par les seuls sous-programmes déclarés publics dans le paquetage
- *Exemple* :

```
package Paquetage_Liste is

type Liste is limited private; -- type limité privé

liste_vide : constant Liste; -- constante différée

...

procedure init(l : out Liste);

...

end Paquetage_Liste;
```



# Types privés limités (2/2)

```
package body Paquetage_Liste is

procedure init(l : out Liste) is
begin
   l := liste_vide;
end;

...

end Paquetage_Liste;
```

```
procedure Test is

l : Liste;

begin
   init(l);
   addElement(l,1);addElement(l,2);addElement(l,3);
   addElement(l,4);addElement(l,5);
   affiche(l);
   addElement(l,6);
   affiche(l);
end;
```

## Programmation Orientée Objet en ADA

- ADA83 implémentait déjà trois des aspects de la POO
  - la définition de types de données (classe) par un ensemble de valeurs (éventuellement composées) et un ensemble d'opérateurs et méthodes
  - l'encapsulation des données, types et sous-programmes (dans les paquetages) permettant d'assurer la *modularité* des programmes et leur *fiabilité* en cachant une partie du code
  - le polymorphisme des sous-programmes, le sous-programme a appeler effectivement étant déterminé à la compilation
- ADA95 introduit le dernier aspect de la POO
  - l'héritage qui permet la *réutilisabilité* de tout le code d'un paquetage, y compris le code caché (ce que l'importation ne permet pas)
  - l'héritage permet de rajouter des types/données/méthodes dans un paquetage enfant sans modifier le paquetage parent (qui n'a pas besoin d'être recompilé)
  - ADA95 introduit le polymorphisme dynamique (sous-programme à appeler déterminé à l'exécution) pour gérer l'héritage entre types



## Héritage en ADA

- Pour implémenter l'héritage, ADA combine :
  - l'héritage des paquetages : un paquetage enfant hérite des déclarations du paquetage parent, y compris les déclaration privées.
  - l'héritage de type : un type **`record`** (ou **`private`**) peut être défini comme héritable (mot-clé **`tagged`**).
- ADA propose en outre
  - des types abstraits ne pouvant être instanciés
  - des méthodes abstraites n'ayant pas de corps
  - des types interfaces (abstraits) qui sont des types abstraits pouvant être hérités
- De manière générale, les mécanismes objet en ADA sont très décomposés et peuvent n'être utilisés qu'en partie par le programmeur

# Paquetage enfant (1/6)

- Un paquetage enfant est déclaré en préfixant son nom par celui de son paquetage parent. La spécification (resp. le corps) du paquetage se fait dans un fichier **`nom_parent-nom_enfant.ads`** (resp. **`nom_parent.nom_enfant.adb`**)

```
package Parent.Enfant is

...

end Parent.Enfant;
```

- Le paquetage enfant hérite des déclarations du paquetage parent, *y compris les déclaration privées*. Il n'a par contre pas accès au corps du paquetage parent et ne peut utiliser les déclarations privées du parent dans sa partie publique.

- Il existe aussi des paquetage enfant privés (**`private package`**) qui peuvent servir à regrouper des sous-programmes ou des types internes au paquetage parent.



# Paquetage enfant (2/6)

- *Exemple : on veut définir une hiérarchie de classes de formes géométriques*

```
package Class_Geometry is

type Shape is limited private;

procedure constructor(s : out Shape;name : String := "");

function toString(s : Shape) return String;

private

type Shape is limited record
   name : String(1..20);
   xPos : Integer := 0;
   yPos :  Integer := 0;
end record;

end Class_Geometry;
```

# Paquetage enfant (3/6)

```
package body Class_Geometry is

procedure constructor(s : out Shape;name : String := "") is
begin
   s.name(1..name'LENGTH) := name;
end;

function toString(s : Shape) return String is
begin
   return "Shape " & s.name;
end;

end Class_Geometry;
```



# Paquetage enfant (4/6)

```
package Class_Geometry.Class_Rectangle is

type Rectangle is limited private;

procedure constructor(r : out Rectangle;name : String := "";width :
   Integer := 0;height : Integer := 0);

function toString(r : Rectangle) return String;

private

type Rectangle is limited record
   s : Shape; -- problème : pas d'héritage de type sur Shape!
   width : Integer := 0;
   height : Integer := 0;
end record;

end Class_Geometry.Class_Rectangle;
```

# Paquetage enfant (5/6)

```
package body Class_Geometry.Class_Rectangle is

procedure constructor(r : out Rectangle;name : String := "";width :
   Integer := 0;height : Integer := 0) is
s : Shape;
begin
   Class_Geometry.constructor(s,name);
   r.s := s;
   r.width := width;
   r.height := height;
end;

function toString(r : Rectangle) return String is
begin
   return "Rectangle " & r.s.name;
end;

end Class_Geometry.Class_Rectangle;
```



# Paquetage enfant (6/6)

```
with Ada.Text_IO;with Ada.Integer_Text_IO;
use Ada.Text_IO;use Ada.Integer_Text_IO;

with Class_Geometry.Class_Rectangle; use Class_Geometry.Class_Rectangle;

procedure Essai is

rect : Rectangle;

begin
   constructor(r => rect,name => "ABCD");
   put(toString(rect));
end Essai;
```

## Héritage de type (1/4)

- un type **`record`** (ou **`private`**) peut être défini comme héritable par le mot-clé **`tagged`**. Le type record défini par héritage étend le type parent en y ajoutant des champs.

- *Exemple* :

```
package Class_Geometry is

type Shape is tagged limited private;

...

end Class_Geometry;
```

- Un type qui hérite d'un type **`limited`** est **`limited`**
- Un type qui hérite d'un type **`private`** est **`private`**
- Un type qui hérite d'un autre est lui même héritable (**`tagged`**)



## Héritage de type (2/4)

```
package Class_Geometry.Class_Rectangle is

type Rectangle is new Shape with private;
-- les champs ajoutés ici sont privés
-- (mais ils pourraient être définis publics)

...
private

type Rectangle is new Shape with record
   width : Integer := 0;
   height : Integer := 0;
end record;

end Class_Geometry.Class_Rectangle;
```

# Héritage de type (3/4)

```
package body Class_Geometry.Class_Rectangle is

procedure constructor(r : out Rectangle;name : String := "";width :
   Integer := 0;height : Integer := 0) is
begin
   r.name(1..name'LENGTH) := name;
   r.width := width;
   r.height := height;
end;

function toString(r : Rectangle) return String is
begin
   return "Rectangle " & r.name;
end;

end Class_Geometry.Class_Rectangle;
```



# Héritage de type (3/4)

- On peut ne rajouter aucun champ :

```
package Class_Geometry.Class_Rectangle is

type Rectangle is new Shape with private;

...
private

type Rectangle is new Shape with null record;

end Class_Geometry.Class_Rectangle;
```

- On peut définir un enregistrement héritable sans aucun champ (il s'agit alors quasiment d'un type abstrait)

```
type Machin_Chouette is tagged null record;
```

## Héritage et conversion

- Attention : l'héritage ne remet pas en cause l'impossibilité de la conversion implicite

```
s : Shape;
r : Rectangle;

s := r; -- impossible, pas de conversion implicite!
-- en plus ici Shape est limited donc l'affectation est impossible même
-- après conversion
```

- ADA oblige toujours à recourir à une conversion explicite par une fonction de conversion : la robustesse est privilégiée par rapport à la facilité d'écriture



## Liaison statique vs dynamique

- Les méthodes en ADA sont par défaut virtuelles (**`virtual`** en C++) c'est à dire gérées par liaison statique : la méthode à utiliser est déterminée à la compilation (la liaison statique est rendue possible par le typage fort de ADA).

```
procedure toString(s : Shape) is
begin
   toString(s);
   -- la méthode toString à appeler est déterminer à la compilation
end;
```

- On peut forcer un paramètre de méthode à être géré dynamiquement en utilisant l'attribut **`CLASS` d'un type `tagged`** qui désigne l'union de tous les types dérivés

```
procedure toString2(s : Shape'CLASS) is
-- le parmètre de String2 peut être tout objet Shape ou d'un type
-- qui en hérite
begin
   toString(s);
   -- la méthode toString à appeler est déterminer à l'exécution
end;
```

## Abstraction en ADA (1/2)

- Un type abstrait est forcément **`tagged`**, et doit toujours être redéfini par dérivation avant toute utilisation. Il ne peut pas exister d'instance d'un type abstrait.
- Un sous-programme abstrait ne possède qu'une spécification et pas de corps.
- Un type qui hérite d'un type abstrait doit redéfinir obligatoirement tous les sous-programmes abstraits de son type père. Sinon, le type fils doit être déclaré abstrait.



## Abstraction en ADA (2/2)

- La procédure **`constructor`** doit être écrite dans le corps du paquetage mais pas la fonction **`toString`**
- Les types qui héritent de Shape doivent implémenter la fonction toString ou être déclarés abstraits

```
package Class_Geometry is

type Shape is abstract tagged limited private;

procedure constructor(s : out Shape;name : String := "");

function toString(s : Shape) return String is abstract;

private

type Shape is abstract tagged limited record
   name : String(1..20);
   xPos : Integer := 0;
   yPos :  Integer := 0;
end record;

end Class_Geometry;
```

# Interface

- Un type interface est un type abstrait et tagués dont tous les sous-programmes sont abstraits. Il permet de gérer l'héritage multiple.

```
type Shape is limited interface;
```

- Les interfaces ont été introduites dans un ajout à la norme en 2005 et ne sont pas encore gérées dans le compilateur GNAT 3.15 (*http://www.gnat.com/*)

- Conclusion : ADA a pu être étendu à la POO sans rien changer aux bases du langage qui contenait déjà des éléments de la POO. La POO permet une meilleure réutilisabilité du code sans altérer sa robustesse.