Chapitre 9

ADA 95

Ada 95

- Le processus de révision de Ada 83 a conduit à Ada 95 dont les aspects importants sont:
  - Programmation Orientée Objet (OOP)
  - Bibliothèques hiérarchiques
  - Objets Protégés
- La compatibilité vers le haut a été maintenue.

PROGRAMMATION PAR EXTENSION

- Idée
  - déclarer un nouveau type qui affine un type parent en
    - héritant
    - modifiant
    - ajoutant

    les composantes et les opérations de ce type parent
- Extension:
  - type dérivé d'Ada 95 (***tagged*** *type*) peut être étendu par dérivation (seulement les ***record*** et types ***private***)

PROGRAMMATION PAR EXTENSION (Exemple)

```
type Rectangle is tagged
record
   Length     : Float;
   Width      : Float;
end record;

type Cuboid is new Rectangle with
record
   Height: Float;
end record;
```

PROGRAMMATION PAR EXTENSION (Exemple -- Suite)

```
type Shape is tagged private;

type Shape is tagged
record
   ....
end record;
```

ou

```
type Shape is new Rectangle with null record;
```

9-5

- *with null rec..* signifie que l'on n'ajoute aucune nouvelle composante.

PROGRAMMATION PAR EXTENSION

- Les types dérivés héritent des opérations du type parent -- en Ada 95, on les appelle les opérations primitives
- Les sous-programmes d'usager sont classifiés parmi les opérations primitives s'ils sont déclarés dans le même paquetage que le type et qu'ils ont un paramètre ou le résultat de ce type.

9-6

PROGRAMMATION PAR EXTENSION

- On pourrait avoir déclaré

```
function Size ( R: in Rectangle) return Float is
begin
   return R.Length * R.Width;
end Size;
```

- Le type CUBOID hérite de cette fonction, mais il serait préférable d'écrire une nouvelle fonction:

```
function Size ( C: in Cuboid) return Float is
begin
   return Size (Rectangle(C)) * C.Height;
end Size;
```

9-7

OOP: Exemple (Ada 83)

```
with Ada.Calendar;
package Alert_System is
   type Priority is ( Low, Medium, High);
   type Device is ( Teletype, Console, Big_Screen);
   type Alert ( P : Priority) is
   record
      Time_Of_Arrival : Ada.Calendar.Time;
      Messager : Text;
      case P is
        when Low => null;
        when Medium | High =>
          Action_Officer : Person;
          case P is
            when Low | Medium => null;
            when High =>
              Ring_Alarm_At: Ada.Calendar.Time;
          end case;
      end case;
   end record
   procedure  Display ( A : in Alert; On : in Device);
   procedure Handle ( A : in out Alert);
   procedure Log ( A : Alert);
   procedure  Set_Alarm ( A : in Alert);
end Alert_System;
```

9-8

OOP: Exemple (Ada 83 - Suite)

```
procedure Handle( A : in out Alert) is
begin
  A.Time_Of_Arrival := Calendar.Clock;
  Log (A);
  Display ( A, Teletype);
  case A.P is
    when Low  => null;
    when Medium | High =>
      A.Action_Oficer := Assign_Volunteer;
      Display ( A, Console);
      case A.P is
        when Low | Medium => null;
        when High =>
          Display ( A, Big_Screen);
          Set_Alarm ( A);
      end case;
  end case;
end Handle;
```


OOP: Exemple (Ada 95)

```
with Ada.Calendar;
package New_Alert_System is
   type Device is ( Teletype, Console, Big_Screen);
   type Alert  is tagged
   record
      Time_Of_Arrival : Ada.Calendar.Time;
      Messager : Text;
   end record;
   procedure Display ( A  : in Alert;
                       On : in Device);
   procedure Handle  ( A  : in out Alert);
   procedure Log     ( A  : in Alert);
   procedure Set_Alarm ( A : in Alert);
```

OOP: Exemple (Ada 95- Suite)

```
type Low_Alert is new Alert with null record;
type Medium_Alert is new Alert with
record
   Action_Officer : Person;
end record;
-- maintenant, annuler l'opération héritée
procedure Handle ( Ma : in out Medium_Alert);
type High_Alert is new Medium_Alert with
record
   Ring_Alarm_At : Ada.Calendar.Time;
end record;
procedure Handle ( Ha : in out High_Alert);
end New_Alert_System;
```


OOP: Exemple (Ada 95- Suite)

```
package body New_Alert_System is
   procedure Handle ( A : in out Alert ) is
   begin
      A.Time_Of_Arrival := Ada.Calendar.Clock;
      Log (A);
      Display ( A, Teletype);
   end Handle;

   procedure Handle ( Ma : in out Medium_Alert ) is
   begin
      Handle ( Alert(Ma));
      Ma.Action_Officer := Assign_Volunteer;
      Display ( Ma, Console);
   end Handle;
```

OOP: Exemple (Ada 95- Suite)

```
procedure Handle ( Ha : in out High_Alert ) is
begin
   Handle ( Medium_Alert(Ha));
   Display ( Ha, Big_Screen);
   Set_Alarm (Ha);
end Handle;

procedure Display ( A     : in Alert:
                    On    : in Device)     is separate;
procedure Log (     A : in Alert)          is separate;
procedure Set_Alarm ( A : in Alert )       is separate;

end New_Alert_System;
```


OOP: Exemple (Ada 95- Suite)

- Nouveau modèle ALERT

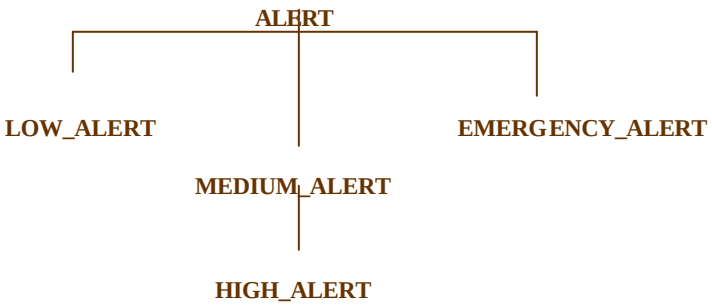
```
-- EMERGENCY_ALERT peut être ajouté sans nouvelle
   compilation

 with New_Alert_System;
 package Emergency_Alert_System is
    type Emergency_Alert is new
       New_Alert_System.Alert with private;
    procedure Handle ( Ea : in out Emergency_Alert);
    procedure Display (Ea : in Emergency_Alert;
                       On : in New_Alert_System.Device);
    procedure Log ( Ea : in Emergency_Alert);
 private
    ....
 end  Emergency_Alert_System;
```

OOP: Classe

- À chaque type étiqueté *[**tagged**] T*, il y a un type associé *T'CLASS*. Il est l'union de tous les types dévirés dans l'arbre.
- Les valeurs de *T'CLASS* sont les valeurs de *T* et de tous ses types dérivés.
- De plus, une valeur d'un type dérivé de *T* peut être converti implicitement au type *T'CLASS*
- *T'CLASS* est un type non-contraint

OOP: Classe

- Exemple: Arbre de type de ALERT



OOP: Classe

- On peut déclarer un type ***access*** à un type classe => désigne n'importe quelle valeur du type classe.
- Exemple sans type ***access***:

```
procedure Process_Alerts ( A : in Alert'Class) is
   ....
begin
   ...
   Handle (A); -- Appeler selon la valeur de l'étiquette
   ....
end Process_Alerts;
```

9-17

OOP: Classe

- On peut déclarer un pointeur à T'CLASS

```
type Alert_Ptr is access Alert'Class;

procedure Process_Alert is
   Next : Alert_Ptr;
begin
   ....
   Next : = ...    -- obtenir la prochaine alerte
   ....
   Handle (Next.all); -- Appeler la routine HANDLE appropriée
   ...
end Process_Alert;
```

9-18

OOP : TYPES ET SOUS-PROGRAMMES ABSTRAITS

- Sous-Programme Abstrait: Sorte de porteur pour les opérations à fournir.
- Un type ayant une ou plusieurs opérations abstraites est un type abstrait.
- En dérivant d'un type abstrait, on fournit les sous-programmes actuels de tous les sous-programmes abstraits.

OOP : TYPES ET SOUS-PROGRAMMES ABSTRAITS (Exemple)

- Exemple:

```
package Base_Alert_System is
   type Alert is tagged null record;
   procedure Handle ( A : in out Alert) is abstract;
end Base_Alert_System;
```

- *tagged null record* pour

```
record;
        null;
end record;
```

OOP : TYPES ET SOUS-PROGRAMMES ABSTRAITS (Exemple)

- Développement d'une infrastructure ALERT

```
with Ada.Calendar;
package Normal_Alert_System is
   type Device is ( Teletype, Console, Big_Screen);
   type Low_Alert is new Base_Alert_System.Alert with
   record
      Time_Of_Arrival : Ada.Calendar.Time;
      Message : Text;
   end record;
   -- Fournir le sous-programme actuel  pour le sous-
    programme abstrait
   procedure Handle ( La : in out Low_Alert);

   procedure Display      ( La : in Low_Alert; On : in Device);
   procedure Log          ( La : in  Low_Alert);
   procedure Set_Alarm ( La : in Low_Alert);
```


OOP : TYPES ET SOUS-PROGRAMMES ABSTRAITS (Exemple)

```
   type Medium_Alert is new Low_Alert with
   record
      Action_Officer : Person;
   end record;

   procedure Handle ( Ma : in out Medium_Alert);

   type High_Alert is new Medium_Alert with
   record
      Ring_Alarm_At : Ada.Calendar.Time;
   end record;

   procedure Handle (Ma: in out High_Alert);
end Normal_Alert_System;
```

OOP : TYPES ET SOUS-PROGRAMMES ABSTRAITS (Exemple)

```
package body Normal_Alert_System  is
   procedure Handle ( La : in out Low_Alert ) is
   begin
      A.Time_Of_Arrival := Ada.Calendar.Clock;
      Log (La);
      Display ( La, Teletype);
   end Handle;

   procedure Handle ( Ma : in out  Medium_Alert ) is
   begin
      Handle ( Low_Alert(Ma));
      Ma.Action_Officer := Assign_Volunteer;
      Display ( Ma, Console);
   end Handle;
   .....
   end Normal_Alert_System;
```


OOP : types Controlled et Limited_Controlled

- Deux types abstraits étiquetés sont déclarés dans le paquetage *Ada.Finalization* avec leurs opérations*:*
  - Controlled : type non-limité (:=, =, /=)
  - Limited_Controlled : type limité (=, /=)
- Opérations fournies sont appelées automatiquement lorsque ces objets sont créés ou cessent d 'exister. Le type *Controlled* et ses descendants possèdent de plus une opération qui est appelée lors de l 'affectation.
- C++ : constructeur, destructeur, et surcharge de l 'opérateur =.

OOP : types Controlled

- Un type dérivé des types *Controlled* ou *Limited_Controlled* est un *type contrôlé* ou *type contrôlé limité*.
- Un objet de ce type est un *objet contrôlé* ou *objet contrôlé limité*.
- Opérations définies (ne sont pas des procédures abstraites, elles effectuent un ***return***):
  - *procedure Initialize (Objet : in out Controlled);*
  - *procedure Adjust (Objet : in out Controlled);*
  - *procedure Finalize (Objet : in out Controlled);*

9-25

OOP : Ada.Finilazition
Opération :*Initialize*

- La procédure *Initialize* est appelée:
  - Lors de l 'élaboration d 'une déclaration d 'un type contrôlé, si la déclaration ne spécifie pas de valeur initiale,
  - Lors de l 'élaboration d 'une déclaration d 'un type composite qui directement ou indirectement contient une composante contrôlée, si
    - aucune valeur n 'a été spécifiée pour la déclaration de l 'objet,
    - aucune valeur de défaut n 'a été spécifiée pour la ou les composantes contrôlées,
  - Lors de l 'évaluation d 'un allocateur (***new***) créant un objet contrôlé, si l 'allocateur ne spécifie pas de valeur initiale,
  - Lors de l 'évaluation d 'un allocateur (***new***) pour un objet composite qui directement ou indirectement contient une composante contrôlée si
    - aucune valeur n 'a été spécifiée pour la déclaration de l 'objet,
    - aucune valeur de défaut n 'a été spécifiée pour la ou les composantes contrôlées,
- Les procédures *Initialize* pour les composantes contrôlées sont appelées avant la procédure *Initialize* de l 'objet contenant les composantes.

9-26

OOP : Ada.Finilazition
Opération : *Adjust*

- La procédure *Adjust* est appelée juste après l 'affectation d 'une valeur à un *objet contrôlé* dans les circonstances suivantes:
  - Durant l 'exécution d 'un énoncé d 'affectation, une valeur est affectée à la variable cible,
  - Lors de la création d 'un objet avec une valeur initiale, la valeur est affectée au nouvel objet,
    - valeur initiale : déclaration objet
    - valeur initiale : appel à un allocateur
    - valeur initiale : dans la déclaration du type
  - Lors de l 'évaluation des résultats d 'un agrégat dans la création d 'un nouvel objet. Les valeurs de l 'agrégat sont affectées aux composantes de l 'objet,
  - Sauf dans le cas d 'une fonction retournant une référence (pointeur), l 'exécution d 'un ***return*** dans le corps de la fonction crée un nouvel objet pour contenir le résultat de la fonction.
- Les procédures *Adjust* sont appelées pour les composantes les plus internes avant les composantes externes.

OOP : Ada.Finilazition
Opération : *Adjust*

- Si l 'objet auquel on affecte une valeur est un *objet contrôlé* la procédure *Adjust* est appelée pour cet objet.

OOP : Ada.Finilazition
Opération : *Finalize*

- Les procédures *Finalize* sont appelées dans l 'ordre inverse des procédures *Initialize* lorsque les objets cessent d 'exister:
  - Un objet cesse d 'exister lorsque le programme quitte la construction dans lequel l 'objet est déclaré par exemple:
    - pour un objet déclaré dans un sous -programme, lorsque l 'on quitte le sous -programme, l 'objet cesse d 'exister,
    - pour un objet déclaré dans un bloc, lorsque l 'on quitte le bloc, l 'objet cesse d 'exister,
  - Un objet alloué dynamiquement cesse d 'existe
    - Lors d 'un appel avec une instance de la procédure générique *Ada.Unchecked_Deallocation* avec un pointeur à l 'objet. (en C++, **delete**).
    - Si un compilateur fournit un ramasse-miette (garbage collector), un objet cesse d 'exister lorsqu 'il est réclamé par le ramasse-miette.
    - Lors du départ d 'une construction contenant un type accès, toutes variables allouées dynamiquement par l 'allocateur pour ce type cessent d 'exister.

OOP : Ada.Finilazition
Opération : *Finalize*

- Les procédures *Initialize* sont appelées juste avant l 'affectation d 'une valeur à une variable cible dans un énoncé d 'affection (fait partie du processus de « finalisation »).
- Les objets créés temporairement pendant l 'exécution d 'une expression sont aussi "finalisés ".
- Dans certaines affectations : A(I .. J) := A (K .. L); le compilateur se crée des objets temporaires:

```
declare
   Objet_Temporaire : T := A ( K .. L);
begin
   A ( I .. J ) := Objet_Temporaire;
end;
```

OOP : Ada.Finilazition
Opération : *Finalize*

- Les étapes suivantes sont effectuées:
  - L 'objet temporaire est créé.
  - L 'objet temporaire est initialisé en affectant la valeur de droite de l 'énoncé d 'affectation du programmeur. *Adjust* est appelée pour chaque composante contrôlée de l 'objet temporaire et si l 'objet affecté est contrôlé, *Adjust* est appelé pour l 'objet lui-même.
  - Le côté gauche de l 'énoncé d 'affectation du programmeur est « finalisé ».
  - La valeur de l 'objet temporaire est affecté au côté gauche de l 'énoncé d 'affectation du programmeur. *Adjust* est appelée pour chaque composante contrôlée de la variable cible et si cette dernière est de type contrôlé, *Adjust* est appelée pour cette variable.
  - L 'objet temporaire est « finalisé ».

Classe en C++ (Spécification)

```
// Déclaration de la classe Temps.
// Fonctions membres définies dans temps6.cpp
#ifndef TEMPS6_H
#define TEMPS6_H
 ¡¥¥¥¥¥¥¥¥¥
ÕÖ×class Temps {
 £¤¦¦¦¦¦¦¦¦
  §public:
  § í Temps( int = 0, int = 0, int = 0 );  // constructeur par défaut
  §
  §// fonctions d'ajustement (set)
  § í Temps &ajusterTemps( int, int, int ); // ajuste heure, minute, seconde
  § í Temps &ajusterHeure( int );    // ajuste heure
  § í Temps &ajusterMinute( int );   // ajuste minute
  § í Temps &ajusterSeconde( int );  // ajuste seconde
  §
  §// fonctions de lecture (get) (normalement déclarées const)
  § í int lectureHeure() const;     // renvoie heure
  § í int lectureMinute() const;    // renvoie minute
  § í int lectureSeconde() const;   // renvoie seconde
  §
  §// fonctions d'affichage (normalement déclarées const)
  § í void afficherMilitaire() const; // affiche temps militaire
  § í void afficherStandard() const;  // affiche temps standard
  §private:
  § í int heure;              // 0 - 23
  § í int minute;             // 0 - 59
  § í int seconde;            // 0 - 59
  ©};

#endif
```

Classe en C++ (Corps)

```
#include "temps6.h"
#include <iostream.h>

// Fonction constructeur initialisant les données private.
// Appelle la fonction membre ajusterTemps pour ajuster
// les variables. Les valeurs par défaut sont de 0.
Temps::Temps( int hr, int min, int sec )
   : { ajusterTemps( hr, min, sec ); }

// Ajuste les valeurs de heure, de minute et de seconde.
Temps &Temps::ajusterTemps( int h, int m, int s )
{
   ajusterHeure( h );
   ajusterMinute( m );
   ajusterSeconde( s );
   return *this;   // permet les appels en cascade
}

// Ajuste la valeur de l'heure
Temps &Temps::ajusterHeure( int h )
{
   heure = ( h >= 0 && h < 24 ) ? h : 0;

   return *this;   // permet les appels en cascade
}
```


Classe en C++ (Corps)

```
// Ajuste la valeur de la minute
Temps &Temps::ajusterMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0;

   return *this;   // permet les appels en cascade
}

// Ajuste la valeur de la seconde
Temps &Temps::ajusterSeconde( int s )
{
   seconde = ( s >= 0 && s < 60 ) ? s : 0;

   return *this;   // permet les appels en cascade
}

// Lit la valeur de l'heure
int Temps::lectureHeure() const {
return heure; }

// Lit la valeur de la minute
int Temps::lectureMinute() const {
return minute; }
```

Classe en C++ (Corps)

```
// Lit la valeur de la seconde
int Temps::lectureSeconde() const {
  return seconde; }

// Affiche le temps en format militaire: HH:MM:SS
void Temps::afficherMilitaire() const
{
  cout << ( heure < 10 ? "0" : "" ) << heure << ":"
       << ( minute < 10 ? "0" : "" ) << minute
       << ":" << ( seconde < 10 ? "0" : "" ) << seconde;
}

// Affiche le temps en format standard: HH:MM:SS AM (ou PM)
void Temps::afficherStandard() const
{
  cout << ( ( heure == 0 || heure == 12 ) ? 12 : heure % 12 )
       << ":" << ( minute < 10 ? "0" : "" ) << minute
       << ":" << ( seconde < 10 ? "0" : "" ) << seconde
       << ( heure < 12 ? " AM" : " PM" );
}
```

9-35

Classe en C++ (Test)

```
#include <iostream.h>
#include "temps6.h"

int main()
{
  Temps t;

  t.ajusterHeure( 18 ).ajusterMinute( 30 ).ajusterSeconde( 22 );
  cout << "Temps militaire: ";
  t.afficherMilitaire();
  cout << "\nTemps standard: ";
  t.afficherStandard();

  cout << "\n\nNouveau temps standard: ";
  t.ajusterTemps( 20, 20, 20 ).afficherStandard();
  cout << endl;

  return 0;
}
```

9-36

Package Temps en Ada 95 (Spécification)

```
with Ada.Finalization;
¡¥¥¥¥¥¥¥¥¥
ØÓipackage Temps is
£¤¦¦¦¦¦¦¦¦¦
 §-- Appels de fonctions membres en cascade.
 §-- Déclaration de la classe Objet.
 §-- Fonctions membres définies dans Temps6.adb
 §
 § ï type Hh is mod 24;
 § ï type Mm is mod 60;
 § ï type Ss is mod 60;
 §
 § ï type Objet       is tagged private;
 § ï type Référence   is access all Objet;
 § ï type Vue         is access constant Objet;
 § í Nil : constant Objet;
 § ¬¹¹¹¹¹¹¹¹¹
 §ÛÜÝfunction constructeurObjet (
 § §                    Objet_Ptr : in     Temps.Référence;
 § §                    Heure     : in     Hh              := 0;
 § §                    Minute    : in     Mm              := 0;
 § §                    Seconde   : in     Ss              := 0  )
 § § return Temps.Référence;
 § ª¹¹¹¹¹¹¹¹¹
 § ¬¹¹¹¹¹¹¹¹¹
 §ÛÜÝfunction ajusterTemps (
 § §                    Objet_Ptr : in     Temps.Référence;
 § §                    Heure     : in     Hh;
 § §                    Minute    : in     Mm;
 § §                    Seconde   : in     Ss             )
 § § return Temps.Référence; -- ajuste Heure, Minute, Seconde
 § ª¹¹¹¹¹¹¹¹¹
```


Package Temps en Ada 95 (Spécification)

```
§ ¬¹¹¹¹¹¹¹¹¹
 §ÛÜÝfunction ajusterHeure (Objet_Ptr : in     Temps.Référence;
 § §                        Heure     : in     Hh              )
 § § return Temps.Référence;    -- ajuste Heure
 § ª¹¹¹¹¹¹¹¹¹
 §
 § ¬¹¹¹¹¹¹¹¹¹
 §ÛÜÝfunction ajusterMinute (Objet_Ptr : in     Temps.Référence;
 § §                         Minute    : in     Mm              )
 § § return Temps.Référence;    -- ajuste Minute
 § ª¹¹¹¹¹¹¹¹¹
 §
 § ¬¹¹¹¹¹¹¹¹¹
 §ÛÜÝfunction ajusterSeconde (Objet_Ptr : in     Temps.Référence;
 § §                          Seconde   : in     Ss              )
 § § return Temps.Référence;  -- ajuste Seconde
 § ª¹¹¹¹¹¹¹¹¹
 §
 §
 §-- fonctions de lecture (get) (normalement déclarées const)
 § ¬¹¹¹¹¹¹¹¹¹
 §ÛÜÝfunction lireHeure (Objet : in     Temps.Objet )
 § § return Hh;     -- renvoie Heure
 § ª¹¹¹¹¹¹¹¹¹
 §
 § ¬¹¹¹¹¹¹¹¹¹
 §ÛÜÝfunction lireMinute (Objet : in     Temps.Objet )
 § § return Mm;    -- renvoie Minute
 § ª¹¹¹¹¹¹¹¹¹
 §
```

Package Temps en Ada 95 (Spécification)

```
§
§ ¬111111111
§ÛÜÝfunction lireSeconde (Objet : in     Temps.Objet )
§ § return Ss;  -- renvoie Seconde
§ a111111111
§
§ ¬111111111
§ÛÜÝprocedure afficherMilitaire (Objet_Ptr : in Temps.Référence );
§ §  -- affiche Temps militaire
§ a111111111
§ ¬111111111
§ÛÜÝprocedure afficherStandard (Objet_Ptr : in Temps.Référence );
§ §  -- affiche Temps standard
§ a111111111
§
§private
§pragma Inline ( ajusterHeure);
§pragma Inline ( ajusterMinute);
§pragma Inline ( ajusterSeconde);
§pragma Inline ( lireHeure);
§pragma Inline ( lireMinute);
§pragma Inline ( lireSeconde);
§ ï type Objet is new Ada.Finalization.Controlled with
§       record
§          Heure   : Hh := 0;
§          Minute  : Mm := 0;
§          Seconde : Ss := 0;
§       end record;
§
§ ¬111111111
§ÛÜÝprocedure Initialize (Objet : in out Temps.Objet );
§ a111111111
```


Package Temps en Ada 95 (Spécification)

```
§ ¬111111111
§ÛÜÝprocedure Adjust (Objet : in out Temps.Objet );
§ a111111111
§ ¬111111111
§ÛÜÝprocedure Finalize (Objet : in out Temps.Objet );
§ a111111111
§ í Nil : constant Objet := (Ada.Finalization.Controlled
§                               with
§                               Heure   => 0,
§                               Minute  => 0,
§                               Seconde => 0);
§
©end Temps;
```

Package Temps en Ada 95 (Corps)

```
with Ada.Text_IO; use Ada.Text_IO;
package body Temps is

   function ajusterTemps( Objet_Ptr       : in Temps.Référence;
                          Heure           : in Hh;
                          Minute          : in Mm;
                          Seconde         : in Ss ) return Temps.Référence is
      Tmp_Ptr : Temps.Référence := Objet_Ptr;
   begin
      Tmp_Ptr := AjusterHeure ( Objet_Ptr, Heure);
      Tmp_Ptr := AjusterMinute ( Objet_Ptr, Minute);
      Tmp_Ptr := AjusterSeconde ( Objet_Ptr, Seconde );
      return Tmp_Ptr ;
   end Ajustertemps; -- ajuste heure, minute, seconde

   function Ajusterheure( Objet_Ptr : in Temps.Référence;
                          Heure : in Hh ) return Temps.Référence is
   begin
      Objet_Ptr.Heure := Heure;
      return Objet_Ptr;
   end Ajusterheure;    -- ajuste heure

   function Ajusterminute( Objet_Ptr      : in Temps.Référence;
                           Minute         : in Mm ) return Temps.Référence is
   begin
      Objet_Ptr.Minute := Minute;
      return Objet_Ptr;
   end AjusterMinute;   -- ajuste minute
```


Package Temps en Ada 95 (Corps)

```
   function Ajusterseconde( Objet_Ptr     : in Temps.Référence;
                            Seconde : in Ss ) return Temps.Référence  is
   begin
      Objet_Ptr.Seconde := Seconde;
      return Objet_Ptr;
   end AjusterSeconde;  -- ajuste seconde

-- fonctions de lecture (get) (normalement déclarées const)

   function Lireheure(Objet : in Temps.Objet) return Hh is
   begin
      return Objet.Heure;
   end LireHeure;    -- renvoie heure

   function LireMinute(Objet : in Temps.Objet) return Mm  is
   begin
      return Objet.Minute;
   end LireMinute;   -- renvoie minute

   function LireSeconde(Objet : in Temps.Objet) return Ss  is
   begin
      return Objet.Seconde;
   end LireSeconde;  -- renvoie seconde

-- fonctions d'affichage (normalement déclarées const)

   package hh_IO is new Ada.Text_IO.Modular_IO ( HH );
      use HH_IO;
```

Package Temps en Ada 95 (Corps)

```
§ØÓìpackage mm_IO is new Ada.Text_IO.Modular_IO ( mm );
§ £¦¦¦¦¦¦¦¦¦
§    use mm_IO;
§ ¡¥¥¥¥¥¥¥¥¥
§ØÓìpackage ss_IO is new Ada.Text_IO.Modular_IO ( ss );
§ £¦¦¦¦¦¦¦¦¦
§    use ss_IO;
§ ¬111111111
§ÞßàprocedureÛPut ( Minute : in mm ) is
§ ªË11111111
§  §begin
§  ¨13´if Minute < 10 then
§  § 6¨11 Put ("0");
§  § 6¾11 Put ( Minute, 1);
§  § ¶´else
§  § ¾11 Put (Minute, 2);
§  § È´end if;
§  §
§  ©end Put;
§ ¬111111111
§ÞßàprocedureÛPut ( Seconde : in ss ) is
§ ªË11111111
§  §begin
§  ¨13´if Seconde < 10 then
§  § 6¨11 Put ("0");
§  § 6¾11 Put ( Seconde, 1);
§  § ¶´else
§  § ¾11 Put (Seconde, 2);
§  § È´end if;
§  §
§  ©end Put;
```


Package Temps en Ada 95 (Spécification)

```
¬111111111
§ÞßàprocedureÛAffichermilitaire(Objet_Ptr : in Temps.Référence) is
§ ªË11111111
§  §begin
§  ¨13´if Objet_Ptr.Heure < 10 then
§  § 6¨11 Put ("0");
§  § 6¾11 Put ( Objet_Ptr.Heure, 1);
§  § ¶´else
§  § ¾11 Put (Objet_Ptr.Heure, 2);
§  § È´end if;
§  ¨11 Put(":");
§  ¨11 Temps.Put ( Objet_Ptr.Minute);
§  ¨11 Put(":");
§  ¨11 Temps.Put ( Objet_Ptr.Seconde);
§  ©end AfficherMilitaire; -- affiche temps militaire
§ ¬111111111
§ÞßàprocedureÛAfficherStandard(Objet_Ptr : in Temps.Référence) is
§ ªË11111111
§  §begin
§  ¨13´if Objet_Ptr.Heure = 0 or Objet_Ptr.Heure = 12 then
§  § 6¾11 Put ( "12");
§  § ¶´else
§  § ¾11 Put ( Objet_Ptr.Heure mod 12 );
§  § È´end if;
§  ¨11 Put(":");
§  ¨11 Temps.Put ( Objet_Ptr.Minute);
§  ¨11 Put (":");
§  ¨11 Temps.Put ( Objet_Ptr.Seconde ); Put (" ");
§  ¨13´if Objet_Ptr.Heure < 12 then
§  § 6¾11 Put ("AM");
§  § ¶´else
§  § ¾11 Put ("PM");
§  § È´end if;
§  ©end AfficherStandard; -- affiche temps standard
```

Package Temps en Ada 95 (Spécification)

```
§ ¬ııııııııı
§Þßàfunction    ConstructeurObjet( Objet_Ptr : in Temps.Référence;
§ §                                Heure          : in hh := 0;
§ §                                Minute         : in mm := 0;
§ §                                Seconde        : in ss := 0) return
 Temps.Référence is
§ ªË¹ııııııı
§   §begin
§Â¹Á¹¹ return AjusterTemps ( Objet_Ptr, Heure, Minute, Seconde) ;
§   ©end ConstructeurObjet;  -- constructeur par défaut
§
§ ¬ııııııııı
§ÞßàprocedureInitialize ( Objet : in out Temps.Objet) is
§ ªË¹ııııııı
§   §begin
§   ¨¹¹ Put_Line (" Le constructeur Initialize est appelé");
§   ©end Initialize;
§ ¬ııııııııı
§ÞßàprocedureAdjust ( Objet : in out Temps.Objet) is
§ ªË¹ııııııı
§   §begin
§   ¨¹¹ Put_Line ("Appel de Adjust ");
§   ©end Adjust;
§ ¬ıııııııı¹
§ÞßàprocedureFinalize ( Objet : in out Temps.Objet) is
§ ªË¹ııııııı
§   §begin
§   ¨¹¹ New_Line;
§   ¨¹¹ Put_Line (" Le destructeur Finalize est appelé");
§   ©end Finalize;
§
©end Temps;
```

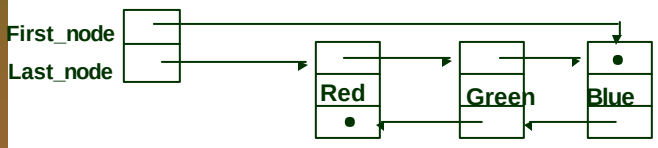

Package Temps en Ada 95 (Test)

```
  with Temps;
   with Ada.Text_IO;
 ¬ııııııııı
ÞßàprocedureMain is
 ªË¹ııııııı
  § ¡¥¥¥¥¥¥¥¥
  §ØÓipackage Txt_IO renames Ada.Text_IO;
  § £¦¦¦¦¦¦¦¦
  § í T        : aliased Temps.Objet;
  § í T_Ptr    : Temps.Référence := T'unchecked_access;
  § í Un_Temps : Temps.Référence := Temps.AjusterTemps( new Temps.Objet, 21 ,4, 5);
  §
  §begin
  ¨¹¹ Txt_IO.New_Line;
  ¨¹¹ Txt_IO.Put ("Temps initial militaire : ");
  ¨¹¹ Temps.AfficherMilitaire(T_Ptr); Txt_IO.New_Line;
  §
  ¨¹¹ T_Ptr := Temps.AjusterHeure ( Temps.AjusterMinute (
  §                                 Temps.AjusterSeconde ( T_Ptr, 22), 30), 18 );
  ¨¹¹ Txt_IO.Put ("Temps militaire : ");
  ¨¹¹ Temps.AfficherMilitaire(T_Ptr); Txt_IO.New_Line;
  §
  ¨¹¹ Txt_IO.Put ("Temps standard : ");
  ¨¹¹ Temps.AfficherStandard(T_Ptr); Txt_IO.New_Line(2);
  §
  ¨¹¹ Txt_IO.Put ("Nouveau temps standard : ");
  ¨¹¹ Temps.AfficherStandard(Temps.AjusterTemps (T_Ptr, 20, 20, 20));
   Txt_IO.New_Line;
  ¨¹¹ T := Un_Temps.all;
  ¨¹¹ Temps.AfficherStandard(Un_Temps); Txt_IO.New_Line;
  §
  ¨¹¹ Temps.AfficherStandard(T_Ptr); Txt_IO.New_Line;
  §
  §
  ©end Main;
```

OOP - Classe - Une liste Conteneur

- Méthodes -- Classe Liste

Méthode	Responsabilité
Initialize Finalize	Initialiser le conteneur Terminer d'utiliser l'objet conteneur
Adjust	Faciliter une copie (deep copy)
=	Comparaison d'égalité pour une liste



OOP - Classe - Une liste Conteneur

- Méthodes -- Classe Liste_iter

Méthode	Responsabilité
Initialize Finalize Deliver	Initialiser l'itérateur Terminer d'utiliser l'objet itérateur Livrer l'objet contenu à la position indiquée par l'itérateur
First Last	Établir la position courante de l'itérateur au premier objet de la liste Établir la position courante de l'itérateur au dernier objet de la liste
Insert	Insérer dans la liste un objet avant la position courante de l'itérateur
Delete	Éliminer et détruire dans la liste l'objet qui est spécifié par la position courante de l'itérateur
Is_End Next	Retourner la valeur True si l'itérateur est rendu à la fin de la liste Aller au prochain élément dans le conteneur qui devient la position courante
Prev	Aller à l'élément précédent dans le conteneur qui devient la position courante

Liste liées générique

```
with Ada.Finalization;
use Ada.Finalization;
generic
   type Item_Type is private;
package Lists is
   type List_Type is limited private;
private
   type Item_Record;
   type Item_Access is access Item_Record;

   type Item_Record is
   record
      Item   : Item_Type;
      Next   : Item_Access;
      Pred   : Item_Access;
   end record;

   type List_Header is
   record
      First  : Item_Access;
      Last   : Item_Access;
      Count  : Natural := 0;
   end record;

   type List_Access is access all List_Header;
   type List_Type is new Limited_Controlled with
   record
      Header : List_Access := new List_Header;
   end record;
   procedure Finalize ( Object : in out List_Type);
end Lists;
```


Class_list -- générique

```
with Ada.Finalization, Unchecked_Deallocation;
use  Ada.Finalization;
generic
   type T is private;                -- Any type
package Class_List is
   type List is new Controlled with private;
   procedure Initialize       ( The : in out List );
   procedure Initialize       ( The : in out List; Data:in T );
   procedure Finalize         ( The : in out List );
   procedure Adjust           ( The : in out List );
   function   "="  ( F : in List; S : in List ) return Boolean;
private
   procedure  Release_Storage( The:in out List );
   type  Node;                    -- Tentative declaration
   type P_Node is access all  Node;  -- Pointer to Node
   type  Node is record
      Prev    : P_Node;                    -- Previous Node
      Item    : T;                         -- The physical item
      Next    : P_Node;                    -- Next Node
   end record;
   procedure  Dispose_Node is new Unchecked_Deallocation( Node, P_Node );
   type  List is new Controlled with record
      First_Node  : aliased  P_Node := null;   -- First item in list
      Last_Node   : aliased  P_Node := null;   -- First item in list
   end record;
end Class_List;
```

Class_list -- générique

```
package body Class_List is

   procedure Initialize ( The:in out List ) is
   begin
      The.First_Node := null;   -- Empty list
      The.Last_Node  := null;   -- Empty list
   end Initialize ;

   procedure Initialize ( The:in out List; Data:in T ) is
   begin
      The.First_Node := new Node'(null, Data, null);
      The.Last_Node  := The.First_Node;
   end Initialize ;

   procedure Finalize ( The:in out List ) is
   begin
      if The.First_Node /= null then
         Release_Storage( The );
         The.First_Node := null;
      end if;
   end Finalize ;
```


Class_list -- générique

```
procedure Adjust ( The:in out List ) is
   Cur              : P_Node := The.First_Node;  -- Original list
   Lst              : P_Node := null;            -- Last created node
   Prv: P_Node := null;          -- Previously created node
   Fst              : P_Node := null;            -- The first node
begin
   while Cur /= null loop
      Lst := new Node'( Prv, Cur.Item,  null );
      if Fst =  null then Fst := Lst; end if;
      if Prv /= null then Prv.Next := Lst; end if;
      Prv := Lst;
      Cur := Cur.Next;              -- Next node
   end loop;
   The.First_Node := Fst;           -- Update
   The.Last_Node  := Lst;
end Adjust ;
function "=" ( F:in List; S:in List ) return Boolean is
   F_Node         : P_Node := F.First_Node;  -- First list
   S_Node         : P_Node := S.First_Node;  -- Second list
begin
   while F_Node /= null and S_Node /= null loop
      if F_Node.Item /= S_Node.Item then
         return False;              -- Different items
      end if;
      F_Node := F_Node.Next; S_Node := S_Node.Next;
   end loop;
   return F_Node = S_Node;          -- Both null if equal
end "=";
```

Class_list -- générique

```
procedure Release_Storage( The:in out List ) is
   Cur : P_Node := The.First_Node;   -- Pointer to curr node
   Tmp : P_Node;                     -- Node to dispose
begin
   while Cur /= null loop            -- For each item in list
      Tmp := Cur;                    -- Item to dispose
      Cur := Cur.Next;               -- Next node
      Dispose_Node( Tmp );           -- Dispose of item
   end loop;
end Release_Storage;

end Class_List;
```

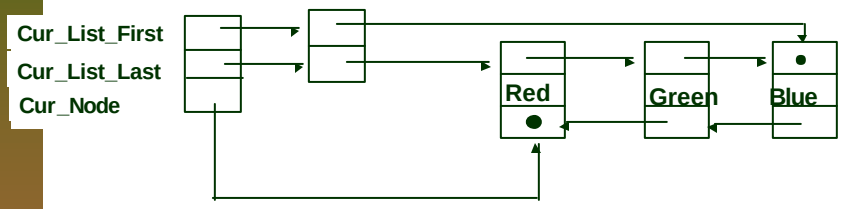

OOP -- Affectation d ’objets contrôlés

Affectation d'objets contrôlés	Actions effectuées

a := b;	anon := b;
	Adjust (anon);
	Finalize (a) ;
	a := anon;
	Adjust (a);
	Finalize (anon);

Actions à l'affectation	Commentaires
anon := b;	Créer une copie temporaire anon
Adjust (anon);	Ajustements requis après avoir copié la mémoire de l'objet source b dans anon
Finalize (a) ;	"Finaliser" la cible de l'affectation
a := anon;	Effectuer l'affectation des composantes de anon dans a
Adjust (a);	Ajustements requis après avoir copié la mémoire de l'objet source anon dans a
Finalize (anon);	"Finaliser" l'objet anon

OOP - Classe - List_Iterator



9-55

Class_list.Iterator -- générique

```
generic
package Class_List.Iterator is

   type List_Iter is limited private;

   procedure Initialize     ( The:in out List_Iter );
   procedure Finalize       ( The:in out List_Iter );
   procedure First          ( The:in out List_Iter; L:in out List);
   procedure Last           ( The:in out List_Iter; L:in out List);

   function    Deliver      ( The:in List_Iter) return T;
   procedure Insert         ( The:in out List_Iter; Data:in T );
   procedure Delete         ( The:in out List_Iter );
   function    Is_End       ( The:in List_Iter ) return Boolean;
   procedure Next           ( The:in out List_Iter );
   procedure Prev           ( The:in out List_Iter );
private
   type P_P_Node is access all P_Node;
   type List_Iter is record
      Cur_List_First        : P_P_Node := null;          -- First in chain
      Cur_List_Last         : P_P_Node := null;          -- Last in chain
      Cur_Node              : P_Node   := null;          -- Current item
   end record;
end Class_List.Iterator;
```

9-56

Class_list.Iterator -- générique

```
package body Class_List.Iterator is

   procedure Initialize ( The:in out List_Iter ) is
   begin
      The.Cur_Node      := null;  -- Iterator not setup
      The.Cur_List_First := null;
      The.Cur_List_Last  := null;
   end Initialize ;

   procedure Finalize ( The:in out List_Iter ) is
   begin
      null;
   end Finalize ;

   procedure First( The:in out List_Iter; L:in out List ) is
   begin
      The.Cur_Node     := L.First_Node;      -- Set to first
      The.Cur_List_First:= L.First_Node'Unchecked_Access;
      The.Cur_List_Last := L.Last_Node'Unchecked_Access;
   end First;
```


Class_list.Iterator -- générique

```
procedure Last( The:in out List_Iter; L:in out List ) is
begin
   The.Cur_Node        := L.Last_Node;      -- Set to last
   The.Cur_List_First  := L.First_Node'Unchecked_Access;
   The.Cur_List_Last   := L.Last_Node'Unchecked_Access;
end Last;

function Deliver( The:in List_Iter ) return T is
begin
   return The.Cur_Node.Item;  -- The current item
end Deliver;

function  Is_End( The:in List_Iter ) return Boolean is
begin
   return The.Cur_Node = null;            -- True if end
end Is_End;

procedure Next ( The:in out List_Iter ) is
begin
   if The.Cur_Node /= null then            --
      The.Cur_Node  := The.Cur_Node.Next;     -- Next
   end if;
end Next;
```

Class_list.Iterator -- générique

```
procedure Insert( The:in out List_Iter; Data:in T ) is
   Tmp   : P_Node;
   Cur   : P_Node      := The.Cur_Node;   -- Current element
   First : P_P_Node    := The.Cur_List_First;
   Last  : P_P_Node    := The.Cur_List_Last;
begin
   if Cur = null then            -- Empty or last item
      if First.all = null then   --  Empty list
         Tmp := new Node'( null, Data, null );
         First.all := Tmp;
         Last.all  := Tmp;
         The.Cur_Node := Tmp;
      else                       --  Last
         Tmp := new Node'( Last.all, Data, null );
         Last.all.Next := Tmp;
         Last.all      := Tmp;
         The.Cur_Node := Tmp;
      end if;
   else
      Tmp := new Node'( Cur.Prev, Data, Cur );
      if Cur.Prev = null then      -- First item
         First.all := Tmp;
      else
         Cur.Prev.Next := Tmp;
      end if;
      Cur.Prev := Tmp;
   end if;
end Insert;
```


Class_list.Iterator -- générique

```
procedure Delete( The:in out List_Iter) is
   Cur   : P_Node       := The.Cur_Node;   -- Current element
   First : P_P_Node     := The.Cur_List_First;
   Last  : P_P_Node     := The.Cur_List_Last;
begin
   if Cur /= null then                    -- Something to delete
      if Cur.Prev /= null then            -- Fix forward pointer;
         Cur.Prev.Next := Cur.Next;       --  Not first in chain
      else
         First.all := Cur.Next;           --  First in chain
         if First.all = null then
            Last.all := null;             --   Empty list
         end if;
      end if;
      if Cur.Next /= null then            -- Fix backward pointer;
         Cur.Next.Prev := Cur.Prev;       --  Not last in chain
      else
         Last.all := Cur.Prev;            --  Last in chain
         if Last.all = null then
            First.all := null;            --   Empty list
         end if;
      end if;
```

Class_list.Iterator -- générique

```
if Cur.Next /= null then              -- Fix current pointer
   The.Cur_Node := Cur.Next;          --  next
elsif Cur.Prev /= null then
   The.Cur_Node := Cur.Prev;          --  previous
else
   The.Cur_Node := null;              --  none empty list
end if;
Dispose_Node( Cur );                  -- Release storage
end if;
end Delete;

procedure Prev( The:in out List_Iter ) is
begin
  if The.Cur_Node /= null then              --
     The.Cur_Node  := The.Cur_Node.Prev;     -- Previous
  end if;
end Prev;

end Class_List.Iterator;

with Class_List;
pragma Elaborate_All( Class_List );
package Class_List_Nat is new Class_List(Natural);
```


Class_list.Iterator -- générique

```
with Class_List_Nat, Class_List.Iterator;
pragma Elaborate_All( Class_List_Nat, Class_List.Iterator );
package Class_List_Nat_Iterator is new Class_List_Nat.Iterator;


with Simple_Io, Class_List_Nat, Class_List_Nat_Iterator;
use  Simple_Io, Class_List_Nat, Class_List_Nat_Iterator;
package Pack_Procedures is
   procedure Ex1;
   procedure Ex2;
   procedure Ex3;
   procedure Ex4;
   procedure Ex5;
   procedure Ex6;
end Pack_Procedures;
```

OOP - Classe

- Exemple : Compte de banque

```
package Class_Account is

   type Account is private;
   subtype Money  is Float;
   subtype Pmoney is Float range 0.0 .. Float'Last;

   procedure Statement  ( The : in Account );
   procedure Deposit    ( The : in out Account; Amount : in Pmoney );
   procedure Withdraw   ( The : in out Account; Amount : in Pmoney;
                          Get : out Pmoney );
   function  Balance    ( The : in Account ) return Money;

private
   type Account is record
      Balance_Of : Money := 0.00;      -- Amount in account
   end record;
end Class_Account;
```


OOP - Classe -- Héritage

- Exemple : Compte de banque avec extension
  - Classe de base
  - Classe dérivée

```
package Class_Account is

   type     Account      is tagged private;
   subtype Money         is Float;
   subtype Pmoney        is Float range 0.0 .. Float'Last;

   procedure Statement   ( The : in Account );
   procedure Deposit     ( The : in out Account; Amount : in Pmoney );
   procedure Withdraw    ( The : in out Account; Amount : in Pmoney;
                           Get : out Pmoney );
   function  Balance     ( The : in Account ) return Money;

private
   type Account is tagged record
      Balance_Of : Money := 0.00;      -- Amount in account
   end record;
end Class_Account;
```

OOP - Classe abstraite -- Héritage

- Exemple : Classe abstraite d 'un compte de banque avec extension

```
package Class_Abstract_Account is

  type Abstract_Account is abstract tagged null record;
  subtype Money  is Float;
  subtype Pmoney is Float range 0.0 .. Float'Last;

  procedure Statement  ( The      : in Abstract_Account ) is abstract;
  procedure Deposit    ( The      : in out Abstract_Account;
                         Amount   : in Pmoney ) is abstract;
  procedure Withdraw   ( The      : in out Abstract_Account;
                         Amount   : in Pmoney;
                         Get      : out Pmoney ) is abstract;
  function  Balance    ( The      : in Abstract_Account )
                       return Money is abstract;
end Class_Abstract_Account;
```

9-65

OOP - Classe dérivée

- Exemple : Utilisation de la classe abstraite pour créer une classe dérivée

```
with Class_Abstract_Account;
use  Class_Abstract_Account;
package Class_Account is

  type      Account     is new Abstract_Account with private;
  subtype   Money       is Class_Abstract_Account.Money;
  subtype   Pmoney      is Class_Abstract_Account.Pmoney;

  procedure Statement  ( The : in Account );
  procedure Deposit    ( The : in out Account; Amount : in Pmoney );
  procedure Withdraw   ( The : in out Account; Amount : in Pmoney;
                         Get : out Pmoney );
  function  Balance    ( The : in Account ) return Money;
private
  type Account is new Abstract_Account with record
    Balance_Of : Money := 0.00;      -- Amount in account
  end record;
end Class_Account;
```

9-66

OOP - Classe dérivée

- Exemple : Utilisation d 'une classe dérivée comme classe de base pour dériver une autre classe

```
with Simple_Io; use Simple_Io;
package body Class_Account is

  procedure Statement( The:in Account ) is
  begin
    Put("Mini statement: The amount on deposit is $" );
    Put( The.Balance_Of, Aft=>2, Exp=>0 );
    New_Line(2);
  end Statement;

  procedure Deposit( The:in out Account; Amount:in
    Pmoney ) is
  begin
    The.Balance_Of := The.Balance_Of + Amount;
  end Deposit;
```

9-67

OOP - Classe dérivée

- Exemple : Utilisation d 'une classe dérivée comme classe de base pour dériver une autre classe

```
with Class_Account;
use Class_Account;
package Class_Account_Ltd is
  type Account_Ltd is new Account with private;
  procedure Withdraw ( The : in out Account_Ltd;
                       Amount : in Pmoney; Get : out Pmoney );
  procedure Reset ( The : in out Account_Ltd);
private
  Withdrawals_In_A_Week : Natural := 3;
  type Account_Ltd is new Account with
  record
    Withdrawals : Natural := Withdrawals_In_A_Week;
  end record;
end Class_Account_Ltd;
```

9-68

OOP - Classe dérivée

```
package body Class_Account_Ltd is

  procedure Withdraw ( The : in out Account_Ltd;
      Amount : in Pmoney; Get : out Pmoney ) is
  begin
    if The.Withdrawals > 0 then              -- Not limit
      The.Withdrawals := The.Withdrawals - 1;
      Withdraw( Account(The), Amount, Get );  -- In Account
    else
      Get := 0.00;                           -- Sorry
    end if;
  end Withdraw;

  procedure Reset( The:in out Account_Ltd ) is
  begin
    The.Withdrawals := Withdrawals_In_A_Week;
  end Reset;

end Class_Account_Ltd;
```


Conteneur d ’objets différents

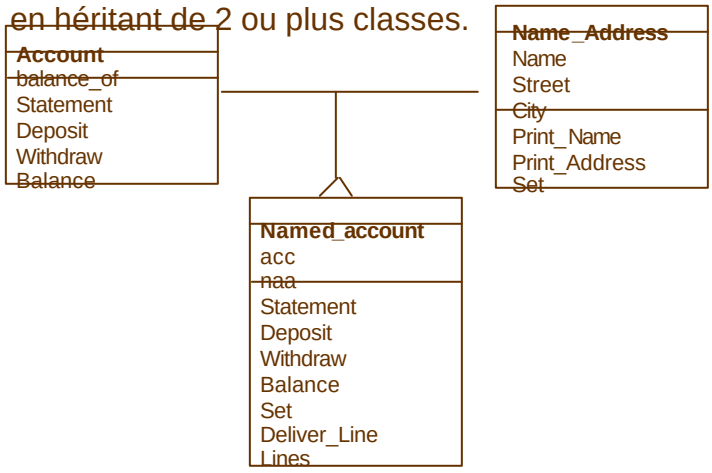
```
procedure Ex4 is
  Max_Accounts : constant := 5;
  type P_Account is access all Account'Class;

  type      Bank_Index      is range 1 .. Max_Accounts;
  subtype Bank_Range        is Bank_Index;
  type      Bank_Array      is array ( Bank_Range ) of P_Account;

  procedure Print_Statement(  No : in Bank_Range;
                              A  : in Account'Class ) is
  begin
    Put("Account number #");
    Put( Integer'Image(Integer(No)) ); New_Line;
    Statement( A );
  end Print_Statement;

  Piggy       : Bank_Array;
  Obtained    : Money;

begin
```

Conteneur d ’objets différents

```
begin
   Piggy(1) := new Account;
   Piggy(2) := new Account_Ltd;
   Piggy(3) := new Interest_Account;
   Piggy(4) := new Interest_Account;
   Piggy(5) := new Account;

   Deposit( Piggy(1).all, 100.00 );       -- deposit $100 #1
   Deposit( Piggy(3).all, 100.00 );       -- deposit $100 #3

   Withdraw( Piggy(2).all, 50.00, Obtained ); -- withdraw $50 #2
   Withdraw( Piggy(4).all, 50.00, Obtained ); -- withdraw $50 #4

   for I in Bank_Range loop
      Print_Statement( I, Piggy(I).all );
   end loop;
   null;
end Ex4;
```


OOP - Héritage Multiple

- L ’héritage multiple permet de créer un nouvelle classe en héritant de 2 ou plus classes.



OOP - Héritage Multiple

- Ada 95 ne supporte pas directement l 'héritage multiple on contourne

```
with Class_Account, Class_Name_Address;
use  Class_Account, Class_Name_Address;
package Class_Name_Account is
   type Named_Account is tagged private;
   subtype Pmoney is Class_Account.Pmoney;
   subtype Money    is Class_Account.Money;
   procedure Set           ( The : out Named_Account; Str : in String);
   procedure Deliver_Line  ( The  : out Named_Account;
                             Line : in Positive ) return String;
   function Lines ( The : in Named_Account ) return Positive;
   procedure Statement ( The : in Named_Account );
   -- … deposit, withdraw, balance
private
   type Named_Account is tagged record
      Acc : Account;                  -- An account object
      Naa : Name_Address;             -- A Name and address object
   end record;
end Class_Name_Account;
```


OOP - Héritage Multiple

```
with Simple_Io; use Simple_Io;
package body Class_Named_Account is

   procedure Set ( The : out Named_Account; Str : in String ) is
   begin
      Set ( The.Naa, Str ) ;          -- Inherited from Account
   end Set ;
   .
   .
   procedure Statement ( The : in Named_Account ) is
   begin
      Put ( « Statement for : »);
      Put ( Deliver_Line ( The.Naa, 1 )); New_Line;
      Statement ( The.Acc );
   end Statement ;

   -- etc..
```

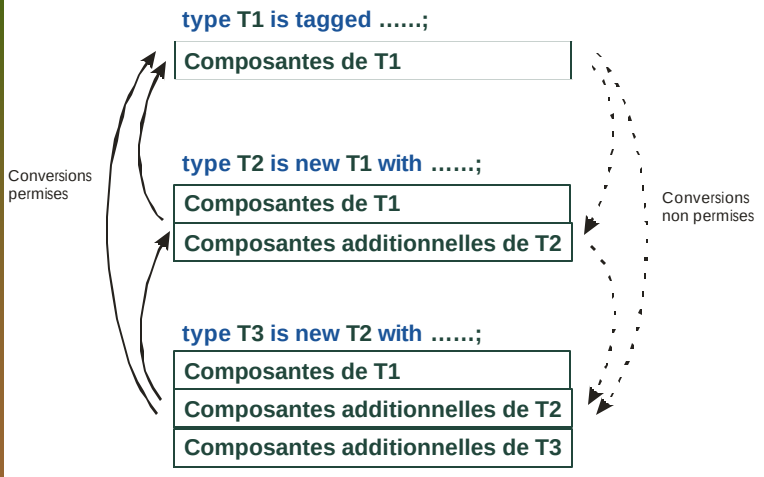
OOP : SOMMAIRE

- Ada 95 introduit:
  - les types étiquetés *[**tagged** type]*
    - seulement pour les articles *[**record**]*
    - les valeurs des types étiquetés contiennent une étiquette *[tag]*
    - l'étiquette contient le type
    - un type étiqueté peut être étendu par dérivation avec des composantes additionnelles
    - par dérivation, le type hérite des opérations primitives
  - les sous-programmes et types abstraits (dénoté par *is abstract*)
    - un sous-programme abstrait n'a pas de corps
    - un type abstrait est une fondation pour construire des types spécifiques avec un protocole commun

OOP : SOMMAIRE (Suite)

  - la notion de Classe
    - *T'CLASS* désigne une classe dont la racine est T
    - la conversion implicite à des valeurs de *T'CLASS* est permise
    - les objets et les paramètres de *T'CLASS* sont non-contraints
    - un pointeur (***type access***) peut désigner n'importe quelle valeur de *T'CLASS*
    - pour l'appel à un sous-programme, la sélection est effectuée à l'exécution selon l'étiquette -- appelé lien tardif *[late binding]* ou polymorphisme
- Avantages:
  - les extensions de type évitent de recompiler et retester un système stable en opération

OOP : Conversion de type



BIBLIOTHÈQUES HIÉRARCHIQUES

- Les paquetages s'avèrent un outil puissant, toutefois, pour les programmes de grande envergure , il faut considérer deux autres aspects:
  - le contrôle de la visibilité des types privés
  - la possibilité d'étendre le paquetage sans recompilation
- Ada 95 ajoute les paquetages enfants et les sous-programmes enfants.

BIBLIOTHÈQUES HIÉRARCHIQUES

- Considérons l'exemple:

```
package Nombre_Complexe is
  type Complexe is private;
  function "+" ( X, Y : Complexe) return Complexe;
  -- Même Chose Pour "-", "*", "/"
    function Cartésien_A_Complexe ( Reel, Imag : in Float) return
 Complexe;
  function Partie_Réelle ( X : Complexe) return Float;
  function Partie_Imag     ( X : Complexe) return Float;
private
  ......
end  Nombre_Complexe;
```

9-79

BIBLIOTHÈQUES HIÉRARCHIQUES

- Si nous voulons ajouter une vue sur l'espace complexe, on peut créer un **paquetage.enfant**:

```
package Nombre_Complexe.Polaire is
   procedure Polaire_A_Complexe ( R, Theta : Float) return
    Complexe;
   function "abs" ( X : Complexe) return Float;
   function Arg ( X : Complexe) return Float;
end Nombre_Complexe.Polaire;
```

- A l'intérieur de paquetage, on peut référencer le type privé *COMPLEXE*.

9-80

BIBLIOTHÈQUES HIÉRARCHIQUES

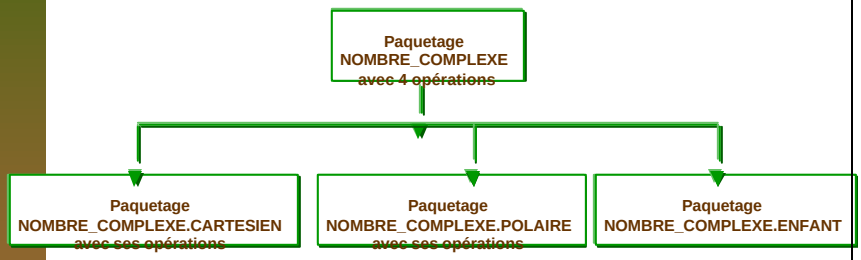
- Accès au paquetage

```
with Nombre_Complexe.Polaire;
package Client is
   ....

   Nombre_Complexe.Partie_Réelle ....;
   Nombre_Complexe.Polaire.Arg ......;
```

- La clause ***with*** pour le parent n'est plus nécessaire si la clause ***with*** *enfant* est utilisée.
- La clause ***use*** si requise pour le parent est toutefois nécessaire.

BIBLIOTHÈQUES HIÉRARCHIQUES

- **Meilleure structure (construction)**



BIBLIOTHÈQUES HIÉRARCHIQUES PROGRAMMATION PAR EXTENSION

```
package Xtk is
  type Widget is tagged private;
  type Widget_Access is access Widget'Class;
  ...
private
     type Widget is tagged
     record
                Parent : Widget_Access;
                .....
     end record;
end Xtk;
-- Maintenant l'étendre à WIDGET
package Xtk.Color is
  type Colored_Widget is new Widget with private;
  .....
private
  type Colored_Widget is new Windget with
  record
    Color : ....;
  end record;
end Xtk.Color;
```

9-83

UNITÉ ENFANT PRIVÉE

- Dans les gros systèmes, il peut être nécessaire de décomposer un système pour faciliter l'implémentation. Ada 83 fournit les sous-unités pour décomposer le corps d'un système.
  - la recompilation d'une sous-unité n'affecte pas les sous-unités de même niveau,
  - si un changement est effectué à un niveau supérieur, toutes les sous-unités doivent être recompilées.
- Pour résoudre ce problème, Ada 95 ajoute une **unité.enfant** qui est totalement privée pour le parent.

9-84

UNITÉ ENFANT PRIVÉE

```
package Os is
   -- le paquetage parent définit des types
   -- utilisés par tout le système
   ...
private
   type File_Descriptor is new Integer;
end Os;

package Os.Exceptions is
   --Exception utilisée par tout le système
   File_Descriptor_Error,
   File_Name_Error,
   Permission_Error : exception;
end Os.Exceptions;
```

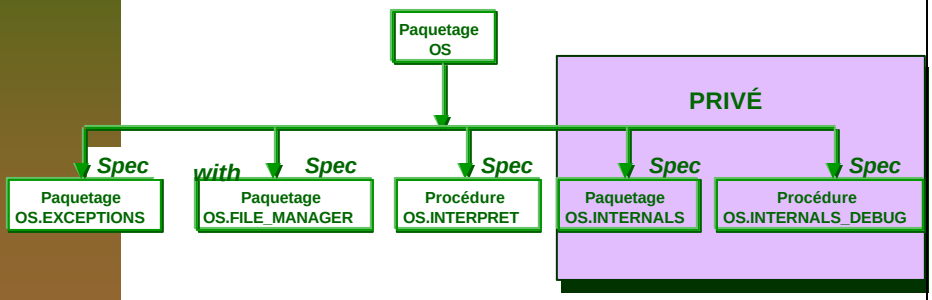
9-85

UNITÉ ENFANT PRIVÉE

```
with Os.Exceptions;
package Os.File_Manager is
   type File_Mode is ( Read_Only, Write_Only, Read_Write);
   function Open ( File_Name : String; Mode : File_Mode)
      return Os.File_Descriptor;
   procedure Close ( File : in Os.File_Descriptor);
end Os.File_Manager;
procedure Os.Interpret ( Command : String);
private package Os.Internals is
   ....
end Os.Internals;
private package Os.Internals_Debug is
   ...
end Os.Internals_Debug;
```

9-86

UNITÉ ENFANT PRIVÉE



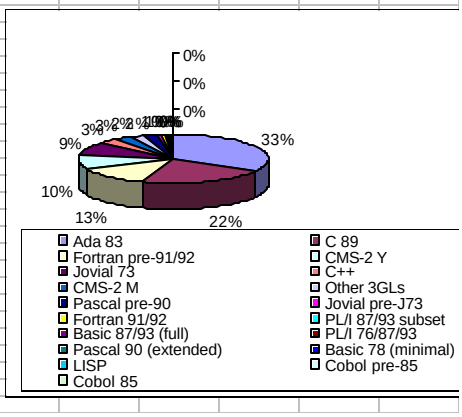
UNITÉ ENFANT PRIVÉE

- Une unité.enfant peut être déclarée n'importe où dans l'arbre hiérarchique. Les règles de visibilité pour les **unités.enfants privés** sont les mêmes que pour les unités.enfants normales sauf que:
  - l'**unité.enfant** privée est visible seulement à l'intérieur du sous-arbre où la racine est son parent; et seulement pour les parties corps (non-visible pour la partie spécification)
  - la partie privée de l'**unité.enfant** peut accéder la partie privée du parent (ne peut exporter à un client une partie privée du parent)

UNITÉ ENFANT PRIVÉE

- Visibilité -- Sommaire
  - une spécification n'a pas besoin d'importer (***with***) son parent; mais elle peut importer un enfant (sibling) -- ne peut importer un enfant privé
  - un corps ***(body)*** n'a pas besoin d'importer (***with***) son parent; il peut importer un enfant (sibling) privé ou pas; il peut importer ses propres enfants
  - une **unité.enfant** privée n'est jamais visible en dehors de l'arbre
  - la partie privée et le corps d'un enfant peut accéder la partie privée de son parent (et grand-parent...)
  - la partie visible d'un enfant privé peut aussi accéder la partie privée de son parent (et grand-parent...)
  - une clause ***with*** pour une enfant implique une clause ***with*** pour ses ancêtres
  - une clause ***use*** pour une unité rend l'unité.enfant accessible par nom_simple

77 -- 450 Langages
95 -- 37 Langages

Total SLOC by General Purpose 3GL for Weapon Systems

Ada 83	49.70
C 89	32.50
Fortran pre-91/92	18.55
CMS-2 Y	14.32
Jovial 73	12.68
C++	5.15
CMS-2 M	4.23
Other 3GLs	3.38
Pascal pre-90	3.62
Jovial pre-J73	1.12
Fortran 91/92	1.00
PL/I 87/93 subset	0.64
Basic 87/93 (full)	0.48
PL/I 76/87/93	0.36
Pascal 90 (extended)	0.29
Basic 78 (minimal)	0.17
LISP	0.10
Cobol pre-85	0.09
Cobol 85	0.00
	148.38