# Introduction à ADA95

## Programmation Objet avec ADA-95

# Introduction

**On Résume d'abord la P.O.O en ADA puis on reprend pour donner plus de détails.**

**Caractéristiques de la P.O.O. :**
- Encapsulation (packages ADA),
- Généricité (généricité ADA),
- Héritage (types dérivés, types taggés d'ADA95),
- Polymorphisme (Procédures et fonctions ADA, "dynamique dispatching" d'ADA95)

**Caractéristiques des objets en POO :**
- Etat de l'objet et son comportement (compétences)
- Les types de base (entiers, réels,...) ont des compétences prédéfinies.

# Introduction à la P.O.O en ADA95

En ADA, la programmation O.O se fait à laide des types **taggés** et le **pointeur général d'accès**.

- Les types "taggés" implantent une extension de type : une sorte d'héritage.
- Un type **taggé** correspond à une généralisation de *Record* (enregistrement) à champs variables. Ce type permet l'ajout de nouveaux variants sans devoir recompiler le paquetage dans lequel le type initial a été déclaré.

Le type **accès général** est une généralisation de type accès. Il permet la création d'un pointeur vers une variable statique. En ADA83, on pouvait seulement créer un accès vers un bloc dynamique (par new).

- La combinaison de GENERAL ACCESS TYPES et de TAGGED TYPES donne un style intéressant et puissant de programmation objets.

# Types taggés

Avant de présenter les types "taggés" d'ADA, on prende l'exemple de la gestion des salaires dans une entreprise. On donne une version ADA-83 que l'on transforme ensuite avec les capacités P.O.O. d'ADA-95.
Le type employé ci-dessous (ADA83-95) est un record à champ variable.

**Version ADA-83 (utilisable en ADA-95) :**

```
TYPE catégorie IS (inconnue, technicien, commercial, conseil);
TYPE employé(type_paie : catégorie := inconnue) IS RECORD
    ID : integer;
    Nom : string(1..20);
    Periode : Dates.Date;          -- vient du package Dates supposé disponible

    CASE type_paie IS
        WHEN technicien =>
            salaire_mensuel : float;
        WHEN commercial =>
            salaire_hebdo : float;
            taux_commission : float;
            total_ventes : float;
        WHEN conseil =>
            salaire_horaire : float;
            heures_travailles : float;
        WHEN inconnue => NULL;
    END CASE;
END RECORD;
```

Si l'on veut ajouter d'autres variants ensuite, par exemple, une nouvelle catégorie "directeur", il faudra tout recompiler (les définitions, les unités utilisatrices du type modifié, ...).

La solution ADA95 pour cette modification est d'utiliser le type taggé. En utilisant le mot **tagged**, on précise que le type (record) ainsi créée peut éventuellement recevoir d'autres champs.

Chaque objet d'un type taggé aura une étiquette (tag) gérée par le compilateur. Ce *tag* peut être considéré comme un discriminent caché. Il ne sera pas directement référencable en tant que champ de l'enregistrement (mais on peut le lire par l'attribut **tag**, au même titre que s'il avait été donné explicitement dans la déclaration.

## Exemple de type taggé

Version ADA-95 :

```
TYPE personne is tagged record
     Nom : string(1..20);
     Genre : tgenre;                -- voir ci-dessous
     Date_naiss : Date;
END RECORD;
type tgenre is (masc, fem, neutre);
```

On utilise le type personne dans un paquetage où on définit ses compétences.
On découvre plus tard la nécessité d'avoir un type *employé*. Un *employé* **est une** personne qui aura besoin de représenter le nom de sa compagnie ainsi que la date de son embauche.

## Programmation par extension : dérivation d'un type taggé

On aura :

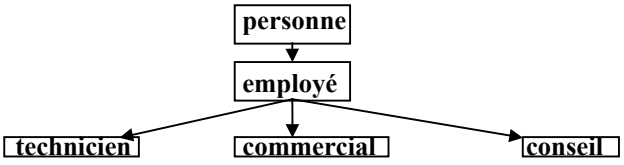
```
Type employé is NEW personne With Record
     id_compagnie : IDType;       -- voir plus bas
     date_embauche : Date;
END RECORD;
```

Le type *employé* reflète une relation est un. Chaque employé a maintenant 5 champs dont trois hérités de *personne*, sans modifier le type personne et les paquetages qui l'utilisent. Cette technique est appelée la *programmation par extension*.

Dans la compagnie en question, le service comptabilité déclare :

```
Type technicien is NEW employé With Record
     salaire_mensuel : float;
END RECORD;

Type commercial is NEW employé With Record
     salaire_hebdo : float;
     taux_commission : float;
     total_ventes : float;
END RECORD;

Type conseil is NEW employé With Record
     salaire_horaire : float;
     heures_travailles : float;
END RECORD;
```

**Remarque** : le type énuméré **catégorie** n'est plus utilisé. Le type employé n'est pas modifié pour obtenir les autres catégories de personnel.
La hiérarchie des types déclarés :



## Conversion de types taggés

ADA permet la conversion d'un type situé (dans la hiérarchie) plus bas vers plus haut. Dans une conversion dans ce sens, on ne manquera pas de champ (mais l'inverse n'est pas vrai).
Si **P** est une Personne, **E** un Employé et **T** un Technicien, on peut écrire :

```
P: Personne;
E : Employé:
T : Technicien;

T :=    (   Nom =>              "Nancy",
            Genre =>        fem,
            Date_naiss =>       Date_of(21, oct, 1950),
            ID_Compagnie => 2345,
            Date_embauche =>    Date_of(1,juin, 1990),
            Salaire_mensuel =>  5000.00
        );

P := Personne(T);
```

La conversion abandonne les 3 derniers champs en trop pour une Personne.

Comment peut on convertir dans le sens inverse? La conversion du haut de la hiérarchie vers le bas nécessite souvent des champs supplémentaires.

Par exemple, si l'on a (**P** : Personne) :

```
P := (   Nom =>          "Nancy",
         Genre =>    fem,
         Date_naiss =>    Date_of(21, oct, 1950));
```

On peut fabriquer **E** : Employé par :

```
E := (P WITH  ID => 2345, Date_embauche => Date_of(1,juin, 1990));
```

On n'a pas redonné les champs déjà dans P.
Le texte suivant WITH est l'agrégat d'extension. Notons qu'en général, les programmes clients ne procèdent pas à ce genre de conversions car souvent, de tels type sont déclarés Private.

Aussi,

Dans une conversion, s'il n'y a pas de champs supplémentaires à ajouter après "with", on écrit :
Obj_de_type1 := (Obj_de_type2 **with null record);**

## Opérations sur les types taggés

**Opération primitive** : Opération directement visible.
Toute opération primitive sur un type taggé T est héritée par tous les types dérivés de T.
Par fois, on ne veut pas cet héritage. On utilisera un paquetage interne pour "cacher" ces primitives (voir plus bas).

# Exemple-1 : gestion des salaires d'une entreprise

Soit le paquetage "Personne" (root de la hiérarchie). "Personne" sera un type tagged Private, c-à-d, il a toutes les caractéristiques d'un type privé (caché avec ":=", "=","/=") et en plus, extensible par dérivation.

```
PACKAGE personnes IS
    Type GENRE is (fem, masc, neutre);
    Subtype DOMAIN_NOM is positive range 1..20;
    Subtype TYPE_NOM is string(domain_nom);

    Type Personne is TAGGED PRIVATE;

-- Sélecteurs des champs
    Function NOM_DE(qui : Personne) return TYPE_NOM;
    Function GENRE_DE(qui : Personne) return GENRE;
    -- Et autres sélecteurs de champs

    Procedure PUT(Item : Personne);      -- Affichage

    PACKAGE constructeurs IS
        -- Pour éviter l'héritage d'une primitive, on la définit dans un Package Interne.
        -- Il ne sera plus considéré comme une primitive et ne sera pas directement hérité.

        Function MakePersonne(Nom : string; G : Genre; D_naiss : Date) return Personne;
    End constructeurs;

PRIVATE
        Type Personne is tagged Record
            Taille_nom : DOMAIN_NOM := 1;
            Nom : type_nom := (others => ' ');
            G : Genre;
            Date_naiss : Date;
        End record;
End Personnes;
```

## A propos du paquetage interne

Le but du paquetage **Personnes** est de fournir la possibilité de dérivation de types. L'héritage des parties telles que les sélecteurs est souhaitable.
Mais si l'on laisse un **employé** hériter MakePersonne, cela sera une erreur car par un appel à MakePersonne, un employé sera construite seulement avec 3 attributs (nom, genre et date). Pour éviter cette situation, il faut un constructeur propre à employé. Il faut donc interdire l'héritage de MakePersonne et proposer MakeEmployé qui initialisera bien un employé (utilisera MakePersonne pour achever une partie de cette initialisation).
On a séparé le constructeur de Personne par le paquetage interne..
"MakePersonne" reste cependant indirectement accessible (par Personnes.Constructeurs.MakePersonn).
On aura utilisé le fait qu'en ADA95, MakePersonne n'est pas une primitive et donc n'est pas (directement) héritée.

## Dérivation de nouveaux types taggés

Le paquetage *personnel* fournira le type *employé*.

```
With Personnes; Use personnes;
With Dates; Use Dates;                          -- Package à écrire en ADA95

Package Personnel IS
-- dérivation privée : les champs ajoutés sont Privés => sont inaccessibles

    Type employé is NEW Personne With Private;

    Type IDType is NEW positive range 1111..9999;

    Function D_embauche_de(qui : Employé) return Date;      --Sélecteur
    Function Id_de(qui : Employé) return IDType;            --Sélecteur
    Procedure PUT(Item : Employé);                          -- Affichage

    PACKAGE constructeurs IS        -- Paquetage interne
        Function MakeEmploye(Nom : string; G : Genre; D_naiss : Date;
                             D_embauche : Date; ID : IDType) return Employé;
    End constructeurs;

PRIVATE
    Type Employé is NEW Personne With
        Record
            id_compagnie : IDType;
            date_embauche : Date;
        End record;
End Personnels;
```

La déclaration :

```
    Type employé is NEW Personne With Private;
```

indique que l'extension sera privée. Employé sera privé, comme Personne.
Employé hérite des opérations primitives (NOM_DE, GENRE_DE, ...) de Personne.
La procédure Put de personne affiche les attributs d'une Personne. Si Put est héritée par Employé, elle affichera seulement les champs de Personne. D'où la procédure PUT dans Employé. Cette procédure PUT surcharge (override) le Put de Personne.

**Remarque** : on a surchargé la procédure Put sans utiliser un paquetage interne car leurs paramètres sont similaires (contrairement au constructeurs) ainsi que leur rôle.

**La définition de Salaires :** définit les 3 autres catégories.

```
With Personnes; Use personnes;
With Personnel; Use personnel;
With Dates; Use Dates;

Package Salaires IS
    Subtype Taux_commission is float range 0.00..0.50;

    Type Technicien is NEW Employé With Private;
    Type Commercial is NEW Employé With Private;
    Type Conseil is NEW Employé With Private;

-- Les sélecteurs à faire en exercice

    Procedure PUT(Item : technicien);
    Procedure PUT(Item : commercial);
    Procedure PUT(Item : conseil);

    PACKAGE constructeurs IS
        Function MakeTechnicien(Nom : string; G : Genre; D_naiss : Date;
                                D_embauche : Date; ID : IDType;
                                Salaire_mensuel : float) return Technicien;

        Function MakeCommercial(Nom : string; G : Genre; D_naiss : Date;
                                D_embauche : Date; ID : IDType;
                                Salaire_hebdo : float;
                                commission : Taux_commission) return Commercial;

        Function MakeConseil( Nom : string; G : Genre; D_naiss : Date;
                                D_embauche : Date; ID : IDType;
                                Tarif_horaire : float) return Conseil;
    End constructeurs;

PRIVATE
        Type Technicien is NEW Employé With Record
            Salaire_mensuel : float;
        End record;

        Type Commercial is NEW Employé With Record
            Salaire_hebdo : float;
            commission : Taux_commission;       -- et les ventes ?
        End record;

        Type Conseil is NEW Employé With Record
            Tarif_horaire : float;                  -- et le nombre d'heures ?
        End record;
End Salaires;
```

Avant de définir les body, voyons comment organise-t-on l'entreprise et sa paie.

## Exemple d'utilisation des types taggés

```
With Ada.text_io; Use Ada.text_io;              -- nouveau dans ADA95
With Personnes; Use personnes;
With Personnels; Use personnels;
With Dates; Use Dates;                          -- Package à écrire en ADA95
With Salaires; Use Salaires;
Procedure Paiement IS
    George : Personne;
    Marie : Employé;
    Martha : Technicien;
    Virginia : Commercial;
    Herman : Conseil;
BEGIN
    George := Personnes.constructeurs.MakePersonne(
              Nom => "George",
              G => masc,
              D_naiss => Date_of(2,nov,1971));

    Marie := Personnel.constructeurs.MakeEmploye(
              Nom => " Marie ",
              G => fem,
              D_naiss => Date_of(21,oct,1950),
              D_embauche => Date_of(1,jun,1989),
              ID => 1234);

    Martha := Salaires.constructeurs.MakeTechnicien(
              Nom => "Martha",
              G => fem,
              D_naiss => Date_of(8,juilt,1947),
              D_embauche => Date_of(6,juin,1985),
              ID => 2222,
              Salaire_mensuel => 5000.00);

    Virginia := Salaires.constructeurs.MakeCommercial(
              Nom => " Virginia",
              G => fem,
              D_naiss => Date_of(1,fev,1955),
              D_embauche => Date_of(1,jan,1990),
              ID => 3456,
              Salaire_hebdo => 2500.00,
              commission => 0.25);

    Herman := Salaires.constructeurs.MakeConseil(
              Nom => " Herman ",
              G => masc,
              D_naiss => Date_of(13,may,1975),
              D_embauche => Date_of(1,jul,1991),
              ID => 1557,
              Tarif_horaire => 7.50);
    Put(Item => George);
    Put(Item => Marie);
    Put(Item =>Martha);
    Put(Item => Virginia);
    Put(Item => Herman);
End Paiement;
```

## Trace d'exécution

A faire en exercice.

## Les corps des paquetages

```
With Ada.text_io;
With Ada.Integer_text_io;          -- nouveau dans ADA95
With Dates.IO;                     -- A écrire

With Salaires; Use Salaires;

--   Il n'y a pas de difficulté particulière. Pour les paquetages internes, leur Body se définit
--   de la même manière à l'intérieur du Body du paquetage qui les contient :

Package Body salaires is
    Package Body constructeurs Is
        Function Maketechnicien(...) IS .....
        .....
    End constructeurs;
```

**Remarques :**

* Les procédures PUT sont écrites et surchargées une par classe. Aucune généralisation (comme on le verra plus avec le **dispatching** et les variables **class-wide**).
* Dans la fonction MakeEmploye du paquetage "Personnels", il y a une conversion vers le haut :
  *Employé -> Personne* :

```
    Function MakeEmploye(Non, Genre, D_naiss, D_emb, Id ... ) IS    -- mettre les types
    Begin
        Return (
                Personnes.Constructeurs.MakePersonne
                (   Nom => N,          -- N est le paramètre nom
                    G => G,
                    Date_naiss => D_naiss
                )
                WITH
                    D_embauche => D_emb,
                    Id_compagnie => Id);
    End MakeEmploye;
```

**Personnes.Constructeurs.MakePersonne** n'a que 3 paramètres. Pour utiliser le constructeur de "Personne", il suffit donc de fournir les 3 paramètres (Nom, Genre, Date_naiss). La partie agrégat d'extension (with ...) fournit les 2 autres valeurs nécessaires à un "Employé". C'est une manière d'utiliser le constructeur de "Personne" pour une partie de la construction. Comme en C++, on appelle le constructeur de la classe mère avant d'alimenter les champs supplémentaires.
Dans une conversions vers le haut, il y a suffisamment de champs.
On réutilisera également les PUT de cette manière (et éventuellement d'autres sous-programmes).

# Les variables d'un type taggé

Dans l'exemple ci-dessus, on a déclaré des variables spécifiques à chaque sous-classe. C'est comme les variables des records à champ variable que l'on contraint ainsi.

On peut avoir des variables taggées non-contraintes; c-à-d, des variables d'un même type (e.g. Personne) que l'on utilise avec des conversions éventuelles.
De même, On peut également prendre un tableau de variables d'un type taggé avec sa hiérarchie (comme en C++, avec liaison dynamique). Par exemple un tableau de "Personne" dont certains éléments seront des Technicien, Employé, Conseil, Commercial ?
**Pour cela, on utilisera le type general-access.**

# Types d'accès général (General Access)

Le type accès classique en ADA83 peut recevoir une valeur de 2 manières : par un NEW ou par la copie d'un autre pointeur. On ne peut pas manipuler l'adresse d'une variable ou d'une constante.
Il existe deux sortes de types accès :
- *Pool-specific access type* : les pointeurs classique en ADA83;
- *General-access type* : pour désigner l'adresse d'une variable, d'une constante ou d'autres valeur dynamiquement créés. Ce type rempli également le rôle des pointeurs ordinaires.

Un pointeur sur procédure est différent (voir le chapitre plus loin).

**Exemple :**

```
Type int_ptr   IS access Integer;                  -- Accès Ordinaire, Pool-specific (pris dans la pool)
Type int_ptr   IS access ALL Integer;        -- Accès général, pointe tout Integer (et cst int)
Type int_ptr   IS access CONSTANT Integer;   -- Accès général (read-only) =>
                                                   -- Comme un paramètre IN,
```

Pour **P : access Constant ...;** P.ALL à gauche d'une affectation n'est pas valide.

Pour respecter les règle de sûreté d'ADA et ne pas manipuler n'importe quoi, une variable ou constante dont l'adresse sera manipulée doit être déclarée **ALIAS**.

**Exemple :**

```
X : Integer;                 -- On ne peut pas manipuler l'adresse de X
Y : ALIASED Integer;         -- On le peut par Y'access
```

Pour donner une valeur à P de type accès général :

```
Type int_ptr IS access ALL Integer;
P : int_ptr;
P := Y'access;           -- On peut toujours faire P := Q;  P:=New....
```

**L'attribut access** de Y (**Y'ACCESS**) donne l'adresse de Y.

**Exemple :**
Le tableau *PromptTable* ci-dessous va contenir les adresses des chaînes (string) de différentes tailles. Pour manipuler l'adresse des prompts, on les déclare ALIASED :

```
With ADA.Text_io;
Procedure acces_general is
Type Str_ptr is Access ALL String;          -- le mot ALL rend le pointeur général.

Prompt1 : Aliased String := "entrer une commande >";
Prompt2 : Aliased String := "Please . ";
Prompt3 : Aliased String := "Err, recommencer. ";
Prompt4 : Aliased String := "Au revoir. ";

PromptTable : array(1..4) of Str_ptr :=
                        (Prompt1'access, Prompt2'access, Prompt3'access, Prompt4'access);

Begin    -- afficher les chaînes
    for I in PromptTable'range Loop
        Ada.Text_io.put(PromptTable(I).ALL);
    End Loop;
End acces_general;
```

Pour manipuler l'adresse d'une string constante, on déclarera :

```
Type Str_ptr is Access CONSTANT String.
```

# Class-Wide types (T'Class)

Dans l'exemple des salaires, on a déclaré une variable par classe spécifique. On va déclarer des variables (ou des tableaux dont les éléments) qui pouvent prendre une valeur de n'importe laquel (des objets, instances) de nos classes.

Tout type taggé T a un **attribut T'Class**.
Cela représente la hiérarchie complète dont T est le parent. Par exemple, **Personne'Class** peut recevoir n'importe laquelle des 5 types de la hiérarchie ou tout autre dérivation future. Personne'Class est appelé un *class-wide type* et une variable de ce type est une *class-wide variable*.

Une variable class-wide doit être **immédiatement** initialisée par une valeur; on pourra changer sa valeur mais pas son type. ?
Cette règle ne concerne pas les pointeurs class-wide.

N.B. : on a dit "variable class-wide"; ce n'est donc pas le cas d'un pointeur class-wide, ni un tableau de pointeurs, ... Dans ces cas, pas d'initialisation immédiate obligatoire. C'est une raison pour préférer les pointeurs.

C'est à peu près la même règle qu'une variable d'un enregistrement à champ variable. La différence par rapport à un enregistrement à champ variable est que le compilateur connaît toutes les valeurs du type du discriminent, mais pour les classes, on n'en sait rien (des extensions futures).

Le compilateur ne peut pas savoir ce qui arrivera à une classe (d'autres dérivations). Il peut pas figer la hiérarchie de T'Class mais veut la connaître à tout moment (pendant l'exécution).

On aimerait déclarer un tableau du type "Employé" qui contiendra différents objets. On ne sait pas trop comment initialiser ce tableau au préalable. On ajoutera en plus d'autres sous classes à l'Employé.

**Exemple** (de class-wide et de dispatching) :

```
With ADA.Text_io; Use ADA.Text_io;
With Dates, Personnes, Personnel, salaires;
Use Dates, Personnes, Personnel, salaires;

Procedure tableau_paie IS
     George : Aliased Personne;              -- Si l'on veut manipuler l'adresse de "George" ...
     Marie : Aliased Employé;
     Martha : Aliased Technicien;
     Virginia : Aliased Commercial;
     Herman : Aliased Conseil;

Type Salaire_ptr is Access ALL Personne'Class;

Type tab_paie is array(1..5) of  Salaire_ptr;
Compagnie : tab_paie;

-- Construction de George, ...., Herman comme avant (appel à Makexx)
Compagnie := (Herman'Access, ... , George'Access);

-- Affichages
For I in Compagnie'range Loop
     Put(Compagnie(I).All);              -- Le bon PUT est sélectionné par "dispatching"
End loop;

End tableau_paie;
```

## Dispatching dynamique

Dans l'exemple ci-dessus, on a

```
For I in Compagnie'range Loop
    Put(Compagnie(I).All);              -- Le bon PUT est sélectionné par "dispatching"
End loop;
```

Si l'on avait utilisé un enregistrement à champ variable, on aurait dû "dispatcher" nous même à la main à l'aide d'un "CASE". Ici, le bon PUT est sélectionné automatiquement et pendant l'exécution (technique importante de la P.O.O : on dit que le PUT correct a "été dispatché").
**Remarque** : les adresses placées dans le tableau Compagnie par ALIASED auraient pu y être rangées par l'allocation dynamique (par NEW).
**Remarque** : le dynamique dispatching est intimement lié aux pointeurs et se réalise avec des pointeurs déclarées ALIASED ou crées par NEW. Aussi, il concerne les opérations primitives d'une classe. Dans le cas ci-dessus, PUT est soit hérité par défaut soit redéfini dans la sous classe (type dérivé).

## Exemple-2 : salaires avec Listes Hétérogènes

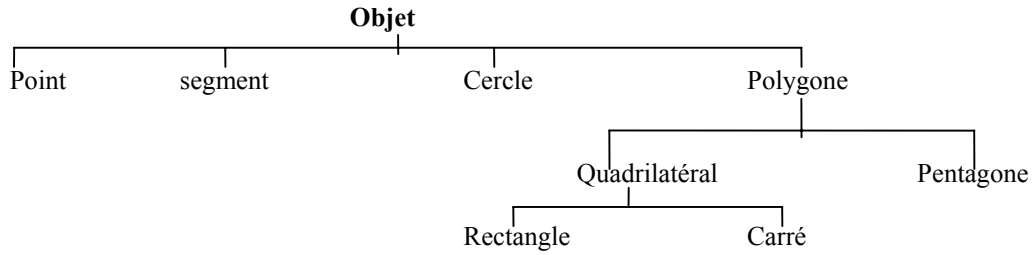
En utilisation le paquetage Listes_Génériques, on crée une liste de pointeurs initialisée par 5 pointeurs sur les personnes de l'entreprise. Ensuite, on affiche cette listes.

```
With ADA.Text_io; Use ADA.Text_io;
With Dates, Personnes, Personnel, salaires;
Use Dates, Personnes, Personnel, salaires;
With Listes_generiques;

Procedure Liste_paie IS
Type Salaire_ptr is Access ALL Personne'Class;      -- Class-wide ptr

Procedure PUT_personne(Item : Salaire_ptr) IS
Begin
    Put(Item.All);              -- Le bon PUT est sélectionné par "dispatching"
End PUT_personne;

Package Liste_salaire IS NEW Listes_generiques(
                                        Element => Salaire_ptr, Display => PUT_personne);
Use Liste_salaire;

Compagnie : Liste;              -- Type exporté du paquetage Liste Générique
Temp : Salaire_ptr;         -- Reçevera n'importe quel pointeur de la hiérarchie


BEGIN       -- Liste_paie

-- construire tout le monde dynamiquement
Temp := NEW Personne'(Personnes.constructeurs.MakePersonne(
            Nom => "George",
            G => masc,
            D_naiss => Date_of(2,nov,1971)));

AddToEnd(Compagnie, Temp);          -- Insère_queue

Temp := NEW Employé'(Personnel.constructeurs.MakeEmploye(
            Nom => " Marie ",
            G => fem,
            D_naiss => Date_of(21,oct,1950),
            D_embauche => Date_of(1,jun,1989),
            ID => 1234));
AddToEnd(Compagnie, Temp);          -- Insère_queue
```

```
Temp := NEW Technicien'(Salaires.constructeurs.MakeTechnicien(
                Nom => "Martha",
                G => fem,
                D_naiss => Date_of(8,juilt,1947),
                D_embauche => Date_of(6,juin,1985),
                ID => 2222,
                Salaire_mensuel => 5000.00));
AddToEnd(Compagnie, Temp);

Temp := NEW Commercial'(Salaires.constructeurs.MakeCommercial(
                Nom => " Virginia",
                G => fem,
                D_naiss => Date_of(1,fev,1955),
                D_embauche => Date_of(1,jan,1990),
                ID => 3456,
                Salaire_hebdo => 2500.00,
                commission => 0.25));
AddToEnd(Compagnie, Temp);

Temp := NEW Conseil'(Salaires.constructeurs.MakeConseil(
                Nom => " Herman ",
                G => masc,
                D_naiss => Date_of(13,may,1975),
                D_embauche => Date_of(1,jul,1991),
                ID => 1557,
                Tarif_horaire => 7.50));
AddToEnd(Compagnie, Temp);

-- Affichages
Display(Compagnie);          -- Le bon PUT sera appelé selon le type du ptr
End Liste_paie;
```

## Exemple-3 : objets simples géométriques

Pour réaliser la hiérarchie suivante :




Exemple de Pentagone : une étoile.

```
Type Objet is tagged record                    -- type extensible
    X,Y : Float;
End Record;

Type point is NEW Objet with NULL record;      -- Dérivation sans ajout
```

**Exemple de possibilité d'avoir un type privé taggé :**

```
Type Segment is tagged Private;                -- type privé taggé puis ...
Private
    Type Segment is tagged Record ....         -- Que l'on complète
.....
```

**Exemple de définition dans un paquetage** (y compris imbriqué dans le même paquetage que Point, Cercle..)

```
Package Seg_cache Is
Type Segment is NEW Objet WITH Private;          -- type privé taggé
...                                              -- puis on complète
Private
    Type Segment is NEW Objet WITH Record
        -- La partie cachée
    End record;
End Seg_cache;
```

**D'autres dérivations dans la hiérarchie d'ojets géométriques :**

```
Type Cercle is NEW Objet with record             -- Dérivation avec ajout
    rayon : Float;
End record;

Type Polygone is NEW Objet with record           -- Dérivation avec ajout
    Cote : Float;
    Surface : Float;
End record;

Type Quadrilatéral is NEW Polygone with record   -- Dérivation avec ajout
    .......
End record;

Type Rectangle is NEW Quadrilatéral with record  -- Dérivation avec ajout
    .......
End record;

Type carre is NEW Quadrilatéral with record      -- Dérivation avec ajout
    .......
End record;

-- Primitives
Function Distance(O:Objet) return Float is-- Distance depuis l'origine. Opération primitive
                                    -- Pour tout objet depuis (X,Y)
Begin
    Return Sqrt(O.X ** 2 + O.Y **2);
End Distance;

Function Surface(O:Objet) return Float is
Begin
    Return 0.0;                  -- car un "Objet" de base n'a pas de surface
End Surface;

-- Et on déclare, par exemple pour un Cercle
Function Surface(C : Cercle) return Float is
Begin
    Return Pi * C.Rayon**2;
End Surface;

-- Conversions
O : Objet := (1.0, 0.5);         -- un agrégat simple
C : Cercle := (0.0, 0.0, 34.7);  -- Objet + Rayon du cercle
O := Objet(C);                   -- Cercle vers Objet :  Simple et directe
C := (O with 41.2);              -- Objet vers Cercle
```

Une hiérarchie a des attributs et des primitives communs (telle que Distance ou Surface). Ces opérations sont exécutables pour n'importe quel objet de la hiérarchie, sans savoir quel type est en train d'appeler la primitive. Une telle manipulation est possible grâce à la notion de "class" et de "class-wide types" .

A Chaque classe est associé un type appelé "class-Wide type". Il est référencé par "root'class". Il est différent du type root lui même (objet /= objet'class). Dans notre exemple, "Objet" est un type spécifique de la classe (l'ensemble de la hiérarchie désignée par "Objet'Class") dont la racine est "Objet".
ADA différencie l'ensemble de la hiérarchie dont la racine est "Objet" avec le type spécifique "Objet" lui même qui est un élément de la hiérarchie.

Pour l'exemple géométrique, on peut également avoir "polygone'Class" ou tout autre. Toutes les propriétés de "Objet'Class" s'appliquent à "Polygone'Class" mais pas l'inverse.

**Un exemple d'utilisation de class-wide :**

```
Function Moment(OC : Objet'Class) return float is  -- Moment d'une force : poids * distance
Begin                                              -- Ici, on prend Poids=Surface, Distance=X
    return OC.X * Surface(OC);
End Moment;
```

L'objet effectif qui appelle Moment sera connu à l'exécution par son "tag". Quel qu'il soit, il a une coordonnée "X" et peut appeler la primitive "surface". Le dispatching (late binding) a lieu dans ce cas où le paramètre est class-wide.

**Moment n'est pas une primitive**. C'est une opération de "Objet'Class". **Il n'y a donc pas d'héritage dans ce cas** et tout élément (objet) de la hiérarchie (qui appelle la fonction Moment) est converti vers le type class-wide Objet'Class.

Question : Qu'est-ce passe-t-il avec la fonction :

```
Function Moment(O : Objet) return float is         -- Pour Objet. Non class-wide
Begin
    return O.X * Surface(O);
End Moment;
```

Cette fonction renvoie toujours zéro car la fonction "Surface" renvoie zéro pour "Objet".

Si elle est déclarée dans le **même paquetage** que "Objet", elle sera **une primitive héritée** par différents objets de la hiérarchie tel que "Cercle". Dans ce cas, si un Cercle invoque "Moment", par héritage statique (Objet -> Cercle), **on aura toujours zéro** (même pour un Cercle) car par early-binding, la fonction "Surface" utilisée sera celle d' "Objet" (et non du Cercle).

Il n'y a donc pas d'adaptation dynamique au Cercle et **le code exécuté est celui de la fonction "Surface" pour "Objet", même si Cercle hérite de "Surface".**

Ce code fonctionne sur la partie "Objet" du "Cercle" (on dit : Surface a **une vision Objet du Cercle**).
La seule **solution** est de surcharger "Moment" pour "Cercle".

On constate qu'une solution utilisant "Class-wide" et le "Dispatching", est préférable. Ainsi, le code ne sera pas dupliqué, les futures sous classes pourront s'en servir et il n'y aura pas de recompilation en cas d'extension.

**Pour quoi des pointeurs Class-Wides ?**
Une des difficulté de class-wide est que l'on ne connaît pas l'espace nécessaire pour le futur objet qui appellera "Moment".

**A vérifier :**
Si on déclare une variable de type class-wide, elle doit être immédiatement initialisé à un objet et ne pourra plus en changer (on a dit plus haut que la valeur pouvait changer mais pas le type !!).
i.e.

```
C: Cercle := ....
S : Segment := ...
T : Objet'Class := C;
T := S;    Pas Possible ?
```

Une autre difficulté est d'avoir un tableau de class-wide (même initialisé) car ses composants peuvent être de différents types, de différente tailles et donc impossible à indexer efficacement (par le compilateur ?).
**D'où l'idée d'utiliser des pointeurs class-wides.**

**Création d'une liste de pointeurs class-wides :**

```
Type Ptr is access Objet'Class;        -- ZZZ : pas de ALL donc seulement par NEW ?
Type Cell;                             -- ➔ on ne pourra pas manipuler l'adresse d'un objet, ni
Type Cell_ptr is access cell;      -- affecter, ... car pas de 'ALL'
Type Cell IS record
    Element : Ptr;
    Next : Cell_ptr;
End record;
```

Dans une telle liste chaînée, on pourra avoir un pointeur sur un Cercle, un autre sur un Triangle, un sur un Point... On pourra manipuler chaque objet de cette liste pour par exemple calculer le total des "moments" (voir le code) :

```
Function Total_moments(L : Cell_ptr) return Float is
Local : Cell_ptr := L;
Résultat : Float := 0.0;
Begin
    Loop
        If local=Null then return Resultat; End if;
        Resultat := résultat + Moment(Local.Element.All);        -- Moment de l'objet courant
        Local := Local.Next;
    End loop;
End Total_moments
```

## Les types abstraits

Utilisé pour définir un type qui sera le fondement (la base commune) d'une hiérarchie avec certaines propriétés communes de tous mais sans avoir besoin de déclarer des objets (instances) de ce type. Dans l'exemple précédent, on n'aura peut être pas besoin d'instancier la classe "Objet".
Aussi, on sait que la fonction "Surface" est nécessaire (si l'on en n'a pas besoin pour nos formes, une classe point nous suffirait!). Mais on sait que "Surface" pour "Objet" est inutile car elle sera surchargée dans les classes dérivées. Mais il est nécessaire de fournir cette fonction à tous les types dérivés (appelé par exemple dans "Moment"). On pourra donc écrire :

```
Package Objets Is
    Type Objet is abstract tagged record          -- Classe abstraite : sans instance
        X,Y : Float;
    End record;
    Function Distance (O : objet) return Float;          -- Opération concrète pour tous
    Function Surface (O : Objet) return Float is Abstract;     -- A surcharger.
End Objets;                                              -- Pas de corps pour "Objet"
```

**Remarques :**

- Il n'y a pas d'objet (variable) d'un type Abstract.
- Un sous programme Abstract n'a pas de corps (ne peut pas être appelé).
- Quand on dérive une classe concrète d'une classe abstraite, il faut surcharger ses opérations Abstraites.

**Les avantages de classe Abstract :**

- On ne déclarera pas par erreur une instance (inutile, erronée) de la classe "Objet" (contrôlé par ADA).
- On n'héritera pas de la fonction "Surface" (genante!).
- On déclarera pas par erreur une fonction "Moment" pour la classe spécifique "Objet" car le corps de la fonction "Surface" qui y est apellée ne sera pas concrètement défini pour "Objet".
- On pourra malgré tout déclarer "Moment" pour le type "Objet'Class". "Moment" fonctionnera toujours car on ne peut pas déclarer une variable d'un type Abstract, et, tout paramètre effectif doit être d'un type concret qui aura sa fonction Surface appropriée.

**Remarque** : La fonction "Distance" a un paramètre formel de type "Objet". On peut se demander si cela ne contredit pas la règle ci-dessus. Non car, la fonction "Distance" sera héritée par les sous classes qui ne l'appelleront jamais avec une instance d'"Objet" car tout simplement, on ne peut pas créer une telle instance !

# Exemple-4 : automobiles

NB : il y a une déclaration class-wide non initialisée !! A vérifier.

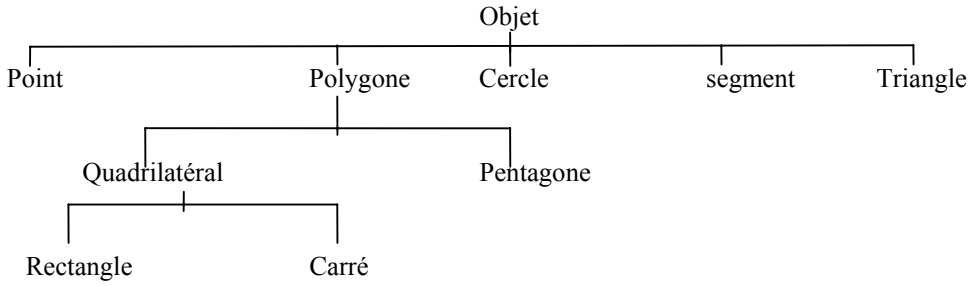
```
Type Vehicule_moteur is TAGGED RECORD                    -- équivalent de classe mère (root)
     Puissance : Positive;
     Volume_moteur : Positive;
     Poids : Positive;
End Record;
Function Puissance_de(M : Vehicule_moteur) Return Positive; --primitive de Vehicule_moteur

Type Suspension IS (soft, hard);
Type Automobile is NEW Vehicule_moteur With Record       -- Sous classe
     Portes : Positive;
     Susp : Suspension;
End Record;

Function SuspOf(A : Automobile) Return Suspension;              -- primitive d'Automobile

-- un bout de code
A : Automobile;
A := MakeVehicule(     (Puissance => X, Volume_moteur => E, Poids => W)          -- conv. bas-haut
                       With  Portes => 4, Susp => Soft
                  );

V : Vehicule_moteur'Class;                    -- V peut être Vehicule_moteur, Automobile
                                                 -- ou       tout autre sous-classe
                                                 -- V n'est pas initialisée ????

Type V_ptr Is ACCESS ALL Vehicule_moteur'Class;     -- Peut pointer un Vehicule_moteur,
                                                    -- Automobile ou tout autre sous-classe

U : ALIASED Automobile;                          -- Peut être pointé par un V_ptr

VP : V_ptr;
VP := V'ACCESS;                               -- VP contient un pointeur sur V
VP := U'Access;
```

**On reprend les notions déjà vues pour les expliquer plus en détails à travers des exemples.**

# Exemple-5 : objets géométriques (Barnes)

* type taggé : possibilité d'ajout d'attributs, d'opérations.
* Pour avoir les informations, ADA utilise le TAG pour connaître le type d'un objet (pendant le Dispatching par exemple). Ce "tag" est similaire à celui des records de Pascal qui permet de faire un choix dans les enregistrements à champ variable.
* L'extension des types privés taggés est également possible.
* On peut reconnaître un type extensible avec les mots "TAGGED" ou "WITH".

* On reprend l'exemple des objets géométriques ci-dessus.




```
Package OBJETS Is
Type Objet is tagged record                    -- type extensible
     X,Y : Float;
End Record;

Function Distance(O:Objet) return Float;       -- Distance depuis l'origine. Opération primitive
Function Surface(O:Objet) return Float;

End OBJETS;
-------------------------------------------------------------------------
With OBJETS; Use OBJETS;
Package FORMES Is
Type point is NEW Objet with NULL record;      -- Dérivation sans ajout

Type Cercle is NEW Objet with record           -- Dérivation avec ajout
     rayon : Float;
End record;

Function Surface(O:Cercle) return Float;       -- Redéfinie pour Cercle. Distance est pour tous

Type Triangle is NEW Objet with record         -- Dérivation avec ajout
     A,B,C : Float;                 -- les cotés du triangle
End record;

Function Surface(O:Triangle) return Float;
End FORMES;
```

**Exemple de conversion de types :**

```
O : Objet := (1.0, 0.5);
C : Cercle := (0.0, 0.0, 34.7);
T, : Triangle;
P: Point;
O := Objet(C);                 -- Cercle → Objet

C := (O with 41.2);            -- Objet → Cercle

T := (O with 3.0, 4.0, 5.0);
```

```
C := (O with Rayon => 41.2);          -- On peut nommer les champs.
C:= (Objet(P) with 53.3);             -- A gauche de "with", un objet quelconque (de la hiérarchie ?)

TT := (T with null record);           -- S'il n'y a pas d'élément à ajouter, on met "with null record"

-- Si l'on ne peut pas nommer l'objet ancêtre
C := (Objet with rayon => 41.2);      -- initialisation par défaut d'un "Objet" :
                                          -- Coordonnées du cercle non définies dans ce cas.

-- les deux lignes ci-dessous sont équivalentes à  "C := (Objet with rayon => 41.2);"
Obj : Objet;          -- sans valeur initiale : Initialisation par défaut d'un "Objet"
C := Cercle := (Obj with rayon => 41.2);  -- Coordonnées du cercle non définies dans ce cas
```

**Remarques sur les types taggés**:

- Seuls les composants existants sont hérités.
- L'héritage, surcharge, ajout de primitives sont permis dans un même endroit mais **l'ajout d'opération n'est possible que si la dérivation a lieu dans une spécification de paquetage.**

- La dérivation peut avoir lieu dans le même paquetage que le parent. On héritera alors des primitives du parent **Mais** on ne peut pas ajouter d'autres primitives à celles du parent !! (voir l'exemple précédent des paquetages "Objet" et "Forme").

- Seul un record peut être un type taggé. Seuls les types taggés peuvent recevoir des ajouts.

**Vérifier**

- La conversion de type taggé n'est permise que **sur une branche du graphe** (vers l'ancêtre ??). (donc on ne peut pas convertir un Cercle vers un Triangle ?) Pour les types non taggés, il n'y a pas cette restriction.

- L'opérateur ACCESS peut être appliqué aux primitives **héritées** (i.e. les primitives sont pas intrinsèques).

- Si une primitive est surchargée, les règles de conformités des subtype s'appliquent (alors que pour les non taggés, seul les règles de conformités des types s'appliquent). Expliqué plus loin.

- Un type dérivé doit avoir la même accessibilité que le parent. ( Ce n'est pas requis pour les types non taggés : on peut dériver n'importe où dans le scope du parent). On ne peut donc pas faire de l'extension de type dans les blocs plus internes (paquetage ou sous-programme).

On reprend l'exemple :

```
-- Primitives
Function Distance(O:Objet) return Float is      -- Distance depuis l'origine. Opération primitive
Begin                                                 -- Pour tout objet depuis (X,Y)
    Return Sqrt(O.X ** 2 + O.Y **2);
End Distance;

Function Surface(O:Objet) return Float is
Begin
    Return 0.0;              -- car un "Objet" de base n'a pas de surface
End Surface;
```

```
Function Surface(C : Cercle) return Float is
Begin
     Return Pi * C.Rayon**2;
End Surface;

-- Exemples de conversions
O : Objet := (1.0, 0.5);              -- un agrégat simple
C : Cercle := (0.0, 0.0, 34.7);           -- Objet + Rayon du cercle
O := Objet(C);                        -- Cercle vers Objet :  Simple et directe
C := (O with 41.2);                   -- Objet vers Cercle
```

# Exemple-6 : Réservation de voyage (Barnes)

Réservation d'ADA airlines.

**Opérations** :

* Création d'une demande de réservation par interaction avec un opérateur;
* Traitement des demandes par un système central avec le résultat;
* 3 sortes de voyages : **simple**, **agréable**, **luxe**.
* Les options d'**agréable** :
    - choix de sièges (couloir, fenêtre)
    - choix de repas (végétarien, poisson, viande)
* Les options de **luxe** : comme agréable + transports à l'hôtel (destination).

On commence par une version simple.

```
Package systeme_de_reservation  IS

Type position IS (couloir, fenêtre);
Type Type_repas IS (végétarien, poisson, viande);

Type réservation is tagged
    record
      Num_vol : Integer;
      Date_voyage : Date;
      Num_siege : String(1..3) := "   ";
    End record;
Procedure Enregistrer(R : In Out  Réservation);
Procedure Choix_siege(R : In Out  Réservation);                    -- Affecter un siège (e.g. "56A")

Type Reservation_simple Is NEW Réservation With NULL record;

Type Reservation_agréable Is NEW Réservation With
  record
     genre_siege : position;
     Repas : Type_repas;
  End record;
Procedure Enregistrer(RA : In Out Reservation_agréable);          -- surcharge
Procedure Cmd_repas(RA : In Reservation_agréable);

Type Reservation_Luxe Is NEW Reservation_agréable With
  record
     Destination : Adresse;
End record;
Procedure Enregistrer(RL : In Out Reservation_Luxe);                    -- surcharge
Procedure Reserve_Limo(RL : In Reservation_Luxe);

End systeme_de_reservation;
```

Le corps du paquetage :

```
Package BODY  Systeme_de_Reservation  IS

Procedure Enregistrer(R : In Out Reservation) Is
Begin
    Choix_siege(R);                    -- Affecter un siège
End Enregistrer;

Procedure Enregistrer(RA : In Out Reservation_agréable) IS
Begin
    Enregistrer(Reservation(RA));      -- Comme une réservation simple (conversion)
    Cmd_repas(RA);                     -- La commande de repas va être enregistrée ailleurs
End Enregistrer;
```

```
Procedure Enregistrer(RL : In Out Reservation_Luxe) IS
Begin
    Enregistrer(Reservation_agreable(RL));                    -- Conversion
    Reserve_Limo(RL);
End Enregistrer;

Procedure Choix_siege(R : In Out réservation) IS Separate;        -- Affecter un siège
Procedure Cmd_repas(RA : In Reservation_agréable) IS Separate;
Procedure Reserve_Limo(RL : In Reservation_Luxe) IS Separate;

End systeme_de_reservation;
```

* Le choix d' "enregistrer" est statique; par simple résolution de surcharge.
* On remarque l'appel de **Choix_siege**. Tous ses appels transmettent un type Réservation (de base). On verra qu'ici, un paramètre class-wide serait mieux et éviterait les conversions.
* Si plus tard, Ada Airlines se procure des sièges de Concorde, on pourra simplement ajouter la catégorie "SuperSonic" sans recompiler (et plus important, peut-être, **sans devoir tester**) :

```
With systeme_de_reservation;
Package Systeme_Reservation_SuperSonic IS
Type Reservation_ SuperSonic Is NEW systeme_de_reservation.Réservation With
  Record
     Champagne : millésime;
     .........
  End record;
Procedure Enregistrer(RA : In Out Reservation_SuperSonic);          -- surcharge
......
End Systeme_Reservation_SuperSonic;
```

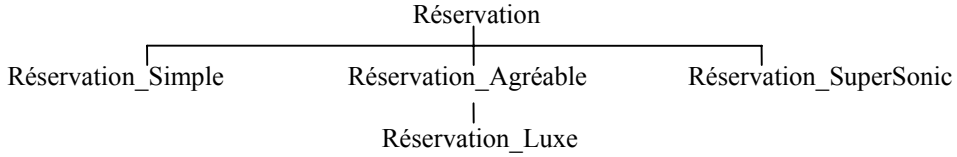
On devrait rendre ce paquetage un "fils" (child) du paquetage "Systeme_de_Reservation". Ceci aurait évité la clause "With systeme_de_reservation" et de plus, aurait souligné qu'il est une partie a part entière du système de réservation.
Les entités du paquetage père seraient immédiatement visibles et la notation préfixée serait évitée. De plus, le fait qu'un paquetage "child" puisse voir la partie private du père peut être précieux.

## Polymorphisme, Class-wide et general-Access

- On a vu la définition de type et son extension.
- On a créer différents types de réservation distincts mais liés ensemble.
- On aimerait manipuler n'importe quel type de réservation mais le traiter selon son vrai type (classe).
- Ceci se fait par **class-wide** qui procure le **polymorphisme.**
- Tout type taggé a un **type** associé **T'Class**. Ce type comprend l'ensemble des types dérivés de l'arbre dont la racine est la classe T.
- Les valeurs de T'class sont les valeurs de T ainsi que toutes celles de ses types dérivés.

Une valeur de n'importe quel type dérivé de T peut être implicitement convertie dans le type T'Class. (Même principe qu'un String borné vers String tout court).

La figure suivante donne **l'arbre** de la hiérarchie des réservations :




- Une valeur de n'importe quel type de réservation peut être implicitement convertie dans le type Réservation'Class.

Notons que *Réservation_Agréable'Class* n'est pas la même chose que *Réservation'Class*. *Réservation_Agréable'Class* est seulement composé de *Réservation_Agréable* et de *Réservation_Luxe*.

- Toute valeur de Class-wide a un **tag** qui identifie son type dans cet arbre à l'exécution. Ce Tag est comme un composant caché.
- **T'Class** est traité comme un type indéfini (comme un tableau non contraint). On n'en connaît pas la taille car le type peut être étendu à l'infini. Pour cette raison, **on doit initialiser** une variable class-wide avec une valeur d'un type spécifique; elle sera alors contrainte avec le Tag de ce type **(peut-on en changer ?)**

```
RA : Réservation_Agréable;
..
RC : Réservation'Class := RA;
```

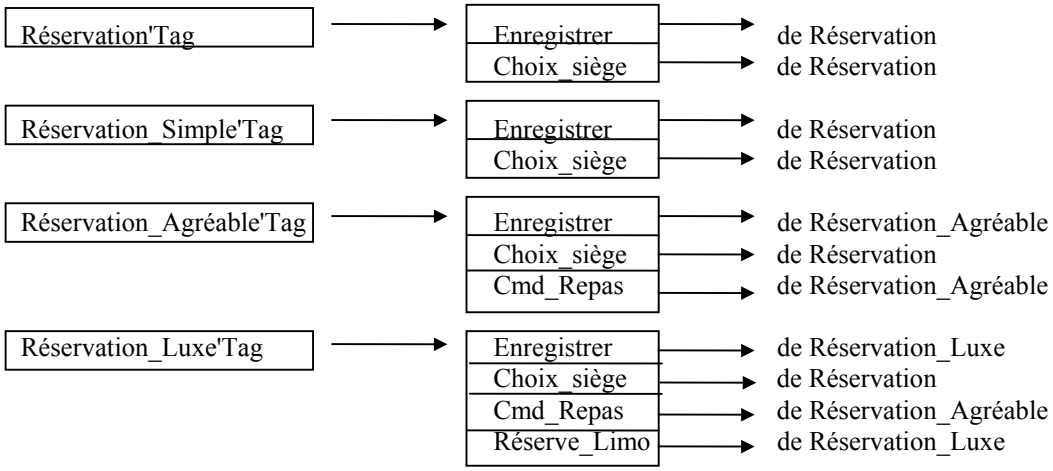
- Un paramètre formel peut donc être class-wide alors que le paramètre effectif est d'un type spécifique dans la classe (classe au sens ADA = la hiérarchie). On remarque encore l'analogie avec les tableaux ou les String comme cité ci-dessus).
Si analogie avec les tableaux non contraints, alors effectivement on peut changer de valeur mais pas de type (i.e. de famille de type) car la variable aura toujours une valeur appartenant à la même hiérarchie. Pour String, c'est encore plus claire : ca sera toujours un tableau de caractères.

- On va créer plusieurs réservations et les enregistrer par une routine centrale (unique).
- Cette routine ne connaît pas le type effectif mais doit fonctionner sans être recompilée.

```
Procedure Process_Réservation (RC : in out Réservation'Class) IS
Begin
...
    Enregistrer(RC);        -- Dispatch selon le TAG
End Process_Réservation;
```

- A l'exécution, le TAG permet de connaître le type effectif, il y aura une conversion du paramètre RC vers le type effectif et l'appel aura lieu.
- **C'est le Dispatching**. Elément important de la "programmation par classes".

- Dispatching peut être implantée par exemple par une technique telle que:
    Avec le TAG, on regarde dans une table qui donne un pointeur vers le code de la primitive appropriée.
    Tout type taggé a son tag par exemple au début de l'enregistrement.
- La figure suivante montre comment on peut implanter la table de Dispatch. L'héritage ou le remplacement des primitives sont également montrés. Pour simplifier le schéma, les primitives prédéfinies telles que l'égalité ne sont pas données.




- Le dispatching est efficace car à l'exécution, rien n'a besoin d'être contrôlé. Il n'y aura pas de primitive manquante (contrôle à la compilation, l'extension ajoute plutôt des choses et n'enlève rien...). On peut fixer les déplacements d'adresses dans la table de Dispatch pour ne pas les calculer à l'exécution.

- Dans le schéma, on voit l'importance de la distinctions entre Réservation'Class et Réservation_agréable'Class. On peut dispatcher vers "Cmd_repas" seulement dans la classe représentée par Réservation_agréable'Class.

**Explication des restrictions sur les dérivations des types taggés :**
Une opération surchargée doit avoir une conformité de "subtype" pour que l'appel fonctionne toujours dynamiquement. L'extension doit également avoir le même niveau d'accessibilité de sorte que toutes les opérations de dispatching soient au même niveau. Ce qui évite les problèmes éventuels avec des variables non locales.

Dans l'exemple de réservation, voyons comment différentes demandes peuvent être conservées dans une liste hétérogène de réservations. On déclare :

```
Type Pointeur_Réservation is access ALL Réservation'CLASS;
```

On peut prendre des variables de ce type accès pour désigner n'importe quelle valeur class-wide. On ne peut pas changer le type spécifique de l'objet référence à un moment donné par un autre type mais on peut référencer des objets de type spécifiques différents.

C'est comme une variable pointeur qui pointe des tableaux de différents tailles -- soit le tableau non contraint -- on ne peut pas changer la taille d'un tableau particulier que l'on pointe à un moment donné mais on peut pointer des tableaux de différentes tailles à des moments différents.

La liste hétérogène sera construite par :

```
Type Cell;
Type Cell_pointeur is access cell;                    -- Pas de ALL : création par NEW seulement
Type Cell is record
    Element : Pointeur_Réservation;
    Next : cell_pointeur;
End record;
```

Et la routine principale peut manipuler les réservations en utilisant un paramètre de type access.

```
Procedure Process_Réservation (RP : in out  Pointeur_Réservation) IS
Begin
...
    Enregistrer(RP.all);          -- Dispatch vers la routine "Enregistrer" appropriée
End Process_Réservation;
```

Utilisation :

```
Liste : Cell_pointeur ;          -- La liste des réservations
...
While Liste /= null loop
    Process_Réservation(Liste.Element);
    Liste := Liste.Next;
End loop;
```

Dans cette version, la valeur de l'objet pointé par RP a un type class-wide qui inclut son TAG indiquant son type spécifique. "RP.ALL" est déréférencé, la valeur du TAG donne le choix de "Enregistrer" et le paramètre est implicitement converti avant de passer à la procédure Enregistrer.

En class-wide programming, il est fondamental de manipuler des objets via des références. Car les objets peuvent être de taille différentes. Les objets référencés par pointeurs peuvent être ajoutés, supprimés, déplacés des listes sans que l'on manipule directement les objets (les records) eux mêmes : ainsi, les objets peuvent être de type **LIMITED**.

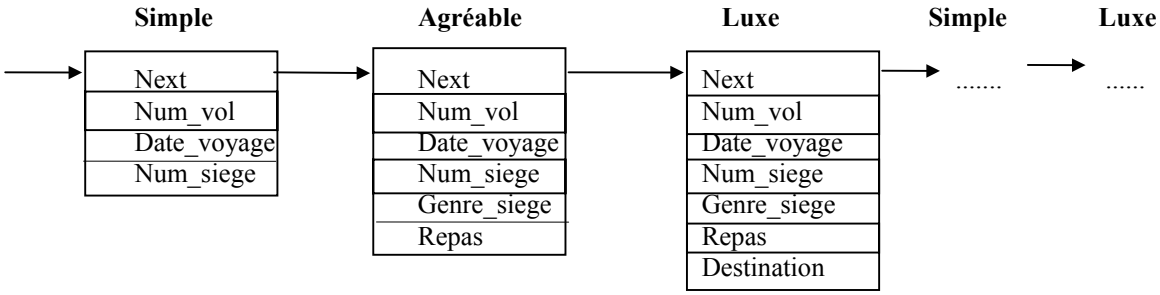
**Si l'on garanti qu'un objet n'est que dans une seule liste à tout moment,** on peut mieux gérer la liste. L'idée est de déclarer un type root qui contient le pointeur vers le prochain élément dans la liste. On étend les types pour pouvoir les placer dans la liste :

```
Type Element;
Type Pointeur_Element IS access all Element'Class;
Type Element is tagged
    Record
       Next : Pointeur_Element;
    End record;
```

Tous les objets de la classe Element'Class peuvent être chaînés ensemble à travers le seul élément commun (le champ Next).
On modifie la Réservation comme dérivé du type Element:

```
Type Réservation Is NEW Element WITH          -- Il était root lui même
    Record
        Num_vol : Integer;
        Date_voyage : Date;
        Num_siege : String(1..3) := "   ";
    End record;
```

Le reste du système est inchangé. Les réservations variées seront liées ensemble comme dans la liste suivante :




**Remarque** (moi) : Voilà pourquoi l'exemple C++ de la Biblio de HP est conçu comme ceci. Designe sacrifié pour l'efficacité.
Ce genre de manipulations est très courant en P.O.O. On peut envisager un paquetage standard de gestion de Listes-Queues :

## Interface du paquetage Queues

```
Package Queues IS
    Queue_Err : Exception;
    Type Queue is LIMITED PRIVATE;
    Type Element is tagged PRIVATE;
    Type Element_Pointeur is access all Element'Class;

    Procedure Join(Q : Access Queue; E : Element_Pointeur);        -- access xx en paramètre
    Function Remove(Q : Access Queue) return Element_Pointeur;
    Function Length(Q : Access Queue) return Integer;

PRIVATE
    Type Element is tagged
    Record
        Next : Element_Pointeur;
    End Record;

    Type Queue is limited
    Record
        Count : Integer := 0;
        First, Last : Element_Pointeur ;
    End Record;
End Queues;
```

**Points importants illustrés dans le paquetage Queues :**
- Les types **Queue** et **Elément** sont Private (cachés).
- Le type **Queue** est **limited** car on ne veut pas autoriser l'affectation de queues (on risque de "rater" les pointeurs, à moins de copier le contenu de la Queue élément par élément, ce qui nécessite de revoir les affectations des éléments internes à chaque poste).
- La déclaration complète de **Queue** est également **limited** (on verra que l'on peut faire autrement). Ceci est pour assurer que dans le **body** du paquetage, on ne fera pas, par erreur, une affectation de Queue. La déclaration complète de Queue dans la section Private concerne le body et n'a pas d'effet à l'extérieur, sur les utilisateurs du paquetage.
- Rappelons que plus tard, on peut écrire un paquetage CHILD qui pourra voir la partie PRIVATE d'où l'intérêt de Limited.
- Le type PRIVATE **Elément** est taggé. C-à-d que la déclaration complète (dans la partie Private) doit également être taggé et permet à l'utilisateur d'étendre depuis le type SANS CONNAITRE ses détails.
- Les sous programmes JOIN et REMOVE prennent un paramètre ACCESS plutôt qu'un paramètre IN OUT. Un avantage de ceci est que REMOVE est une FONCTION (Et si la Queue devient VIDE ??!!). Un inconvénient sera de devoir déclarer le Queue ALIASED ou créer la Queue par un allocateur (NEW).

En utilisant ce paquetage, on peut déclarer le type root du paquetage Réservation :

```
With Queues;
Package Système_de_Réservation IS
...
Type Reservation is NEW QUEUES.Element WITH
    record
      Num_vol : Integer;
      Date_voyage : Date;
      Num_siege : String(1..3) := "   ";
    End record;
....
End Système_de_Réservation;
```

On crée et place les réservations dans la Queue par :

```
Type Pointeur_Queue is access Queue;
The_Queue : Pointeur_Queue := NEW Queue;
...
New_Resvn : Pointeur_Réservation := NEW Réservation_Agréable;
....
Join(The_Queue, New_Resvn);
...
```

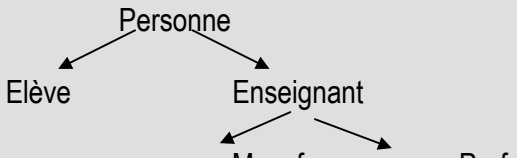
Et pour supprimer une réservation de la queue :

```
Next_Réservation  : Pointeur_Réservation;
....
Next_Réservation  := Pointeur_Réservation(Remove(The_Queue));          -- conversion
Process_Réservation (Next_Réservation);
...
```

Notons qu'il faut explicitement convertir le résultat de REMOVE vers le type Pointeur_Réservation car REMOVE renvoie un Pointeur_Elément. Rien ne nous a empêché de mettre n'importe quel type dérivé d'Elément dans la queue et donc il n'y a aucune garanti que celui-ci soit une Réservation. La conversion contrôle que le type référencé est bien un membre de **Réservation'Class**. Il y aura un **Constraint_Error** si le contrôle échoue. (On verra plus loin comment l'on peut garantir que seules des Réservations soient mises dans la queue.

**Résumé des règles de conversions et types taggés:**

Classes **Personne(Nom, Date) → Elève(+Année)**
**Personne(Nom, Date) → Enseignant(+Dépt)**
**Enseignant → Mconf(+Nb_TD); Prof(+Nb_Crs**)




Cl-Wide **Personne'Class; Elève'Class; Enseignant'Class**; ...
Cl Var **P** : Personne; **El** : Elève; **En** : Enseignant; **MC** : Mconf; **Pr** : Prof;
Cl-W var **PC** : Personne'Class; **ElC** : Elève'Class; **EnC** : Enseignant'Class
(Avec des initialisations nécessaires)

♦ Le principe général : on ne peut convertir que vers (towards) le root. Des contrôles run-time peuvent être nécessaires pour s'en assurer si la source est Class-wide. Si le contrôle échoue, une Constraint_Error est levée.

♦ La conversion entre deux types spécifiques est seulement admise si elle est vers le root.
**P := Personne(El)** OK?
**En := Enseignant(El)** NON?

♦ La conversion entre un type spécifique et un type class-wide de TOUT ANCETRE est implicitement admise (Il n'y a pas de conversion à écrire, c'est implicite).
**PC := Personne'Class(En)** OK ? Dans les appels de Procédures

♦ La conversion vers un type class-wide qui n'est pas d'un ANCETRE n'est pas permise. (donc il faut monter quand on a un type class-wide dans la conversion).
**ElC := Elève'Class(En);** NON ?

♦ La conversion entre un class-wide et un type spécifique est seulement admise si le type de la valeur actuelle convertie est un descendant du type spécifique. Un contrôle dynamique est nécessaire. (un exemple serait bien venu !)
**ElC : Elève'Class := El;** -- Initialisation à Elève. La valeur actuelle convertie = l'Elève El.
**En := Enseignant(ElC);** NON ? Car l'Elève El n'est pas descendant de Enseignant.
**P := Personne(ElC);** OUI ? Car l'Elève El est descendant de Personne.

♦ La conversion entre deux types class-wide est admise si la valeur actuelle est dans la classe destination. Un contrôle dynamique est nécessaire si la classe source n'est pas sous classe de la classe destination.
**ElC : Elève'Class := El;** -- Initialisation à Elève. La valeur actuelle convertie = l'Elève El.
**EnC := Enseignant(ElC);** NON ? Car l'Elève El n'est pas dans la classe Enseignant.
**PC := Personne(ElC);** OUI ? Car l'Elève El est dans la classe Personne.

♦ La conversion entre les types ACCESS est permise si les types désignés peuvent être convertis dans la même direction (c-à-d, ci-dessus). Un contrôle peut être nécessaire. Dans une conversion d'ACCESS, on change de vue d'un même objet et il faut vérifier que cette vue est une interprétation permise.
Dans l'exemple de Réservation, on a une conversion entre Pointeur_Elément et Pointeur_Réservation. Elle provoque un contrôle pour savoir si l'objet désigné a un type spécifique dans la classe Réservation'Class.

Aussi

- Les paramètres de type taggé sont TOUJOURS passés **par référence** et considérés comme ALIASED.
- Le dispatching a lieu avec TOUT MODE de passage de paramètres y compris les paramètres ACCESS.
- Le dispatching N'A PAS LIEU avec un paramètre d'un type ACCESS nommé

# Type et sous-programmes abstraits

- **Type taggé Abstrait** : pour définir les fondements de la construction d'une hiérarchie et de ses dérivations.
- On ne peut pas déclarer des objets (instances, variables) d'un type abstrait.
- Un type Abstrait peut avoir des primitives ABSTRAITES. Ces primitives ne peuvent pas être appelées directement et n'ont pas de Body. Elles sont des "place holders" pour les opérations que l'on ajoutera après.
- Les dérivations déclareront ces opérations de manière concrète.
- Si tous les sous programmes Abstract sont remplacés par des concrets dans le type dérivé, alors le type dérivé n'a pas besoin d'être Abstract et on pourra déclarer des objets de ce type de manière habituelle.

On modifie l'exemple de Réservation en déclarant Réservation comme une classe Abstract. Ceci permet de programmer et de compiler **l'infrastructure** telle que *Process_Réservation* qui manipule les réservations sans se préoccuper des types concrets qui viendront plus tard.

```
Package Systeme_de_Reservation  IS
    Type Reservation is  Abstract tagged null record;
    Type Pointeur_Réservation is access ALL Réservation'CLASS;
    Procedure Enregistrer(R : In Out Reservation) IS Abstract;          -- place Holder (bouche trou)
End Systeme_de_Reservation;
```

La procédure abstraite **Enregistrer** et le paquetage n'ont pas de Body.

On installe l'infrastructure et les 3 types de réservations que l'on aura après.

```
Package Systeme_de_Reservation.Subsonic  IS                    -- CHILD
    Type position IS (couloir, fenêtre);
    Type Type_repas IS (végétarien, poisson, viande);        -- Vert, Blanc, Rouge

    Type Reservation_Simple Is NEW Réservation With
    record
        Num_vol : Integer;
        Date_voyage : Date;
        Num_siege : String(1..3) := "   ";
    End record;

-- Sous programme effectif pour les Abstracts
    Procedure Enregistrer(R : In Out Réservation_Simple);
    Procedure Choix_siege(R : In Out Réservation_Simple);      -- Exemple de siège : "56A"

    Type Reservation_agréable Is NEW Réservation_Simple With
    record
        genre_siege : position;
        Repas : Type_repas;
    End record;
    Procedure Enregistrer(RA : In Out Reservation_agréable);          -- surcharge
    Procedure Cmd_repas(RA : In Reservation_agréable);

    Type Reservation_Luxe Is NEW Reservation_agréable With
    record
        Destination : Adresse;
    End record;
    Procedure Enregistrer(RL : In Out Reservation_Luxe);              -- surcharge
    Procedure Reserve_Limo(RL : In Reservation_Luxe);

End Systeme_de_Reservation.Subsonic;       -- Fin de CHILD
```

**Remarque :** La procédure **Choix_siege** n'est pas déclarée Abstract. Ce n'est pas la peine car cette procédure n'est pas GENERALE au même sens que **Enregistrer** qui elle, est nécessaire pour l'infrastructure donnée par le paquetage "Systeme_de_Réservation".

On rend Abstract ce qui est commun et qui va être redéfini par les types dérivés.

Lors de l'ajout de "Réservation_SuperSonic", on peut le dériver de la base "Réservation" comme avant. On pourra également le dériver d'un autre point de la hiérarchie selon les besoins.

## Règles de Types/Procédures Abstract

**Exemple des types abstraits Personne, Homme et Femme :**
- On n'a pas besoin d'objet de type Personne car c'est un type incomplet et très général contenant les propriétés communes des Hommes et des Femmes.
**Personne sera Abstract**. Les vrais Personnes seront des Hommes et des Femmes.

- On peut empêcher la déclaration des objets Personnes en la déclarant Abstract.

```
Type Personne is Abstract tagged
    record
        Date_naiss : Date;
    End record;
```

Ceci n'empêchera pas d'avoir des paramètres de type Personne (Abstract) :

```
Procedure Print_details(P:Personne) Is
Begin
    Print_Date(P.Date_naiss);
End Print_details;
```

- On ne dispatchera jamais dans cette procédure car **il n'y aura jamais un objet de type Personne** lui-même. Mais, cette procédure peut être utilisée pour afficher les informations communes des types dérivés de Personne.
- Quand on dérive d'un type Abstract, on **n'est pas obligé** de fournir une procédure concrète pour chaque procédure Abstract.
Mais si tel est le cas, le type dérivé sera lui aussi Abstract et doit être déclaré comme tel.
- On peut définir des procédures Abstract pour un type **non Abstract** (par ajout ou remplacement des procédures héritées). Par conséquent, le type dérivé doit être déclaré Abstract.

**Peut on déduire la règle suivante : ?**

- Quand un type déclare une procédure Abstract, alors ce type doit être Abstract. (Ca ne s'applique pas si l'on hérite d'Abstract). ?
- Un type dérivé peut décider d'être Abstract (e.g. en déclarant des procédures Abstract) sans que son parent soit Abstract.

- On peut déclarer une procédure non Abstract dont un paramètre est Abstract(e.g. Print_Details ci-dessus).

Une fonction qui renvoie un type Abstract doit être Abstract.

**Cas de fonction qui renvoie un type non abstrait mais, par la suite, le type est étendu :**
La fonction ne peut pas renvoyer une valeur du type étendu car elle ne sait pas comment fournir des données pour les nouveaux composants (elle ne peut pas faire une conversion vers le bas car elle ne sait pas quoi mettre avec WITH de la conversion).
Pour que le nouveau type dérivé puisse utiliser cette fonction, elle doit être surchargée pour le nouveau type dérivé. Si l'on ne fournit pas une telle fonctions surchargée, alors elle devient Abstract pour le type dérivé (par héritage ?) et alors, le nouveau type doit être déclaré Abstract car elle a une fonction Abstract. Cette règle s'applique même si l'extension est de fait NULL.

**Règle** : Si une fonction renvoie un type NON-Abstract et puis, ce type est étendu alors surcharger la fonction pour le type dérivé.

# Dispatching

- Par leur Tag, les objets sont auto-identifiables.
- Soit

```
Type Personne is Abstract tagged
     record
          Date_naiss : Date;
     End record;
Type Homme is New Personne With
     record
          barbu : boolean;
     End record;
Type Femme is New Personne With
     record
          Enfants : Integer;
     End record;
```

On remarque que l'information "sexe" n'est pas explicitement représentée. Mais cette information est implicite dans le Tag et on peut y accéder.

**Le tag peut être testé :**

```
P : Personne'Class;          -- pas d'init ?
If  P in Femme then          -- P dans (la classe) Femme?
     ...
end if;
```

ou par l'**attribut TAG**

```
If (P'Tag = Femme'Tag) then
     ...
end if;
```

- L'attribut **TAG** peut être appliqué à une valeur d'un type class-wide (e.g. une variable class-wide) ainsi qu'à un type taggé lui-même (i.e. pas aux variables de ce type mais lui-même).
- La valeur de l'attribut TAG est de type PRIVATE **tag** (mais pas LIMITED) déclaré dans le paquetage **ADA.Tags**.
- On peut déclarer des variables de type **Tag** comme d'habitude.
- L'attribut **Tag** ne peut pas être appliqué à une valeur d'un type spécifique (mais à class-wide ou au type taggé lui-même).

On peut écrire

```
if  P in Femme'Class then ...          -- P dans la classe Femme?
```

- Femme'Class (c-à-d "comme Femme") couvre la classe Femme et ses dérivés.
- On ne peut pas obtenir d'autres informations de Tag (e.g. des informations sur les extensions). Il vaut mieux toujours tester l'appartenance (par IN) qui est indépendant des extensions.

- Il est important de savoir dans quels cas on utilise le dispatching par opposition à la résolution statique des liens.

**Principe de base de dispatching** : le dispatching est utilisé seulement lorsqu'un opérande de contrôle (un des paramètres lors d'appel d'un sous programme) est de type class-wide.

Dans

```
RC : Réservation'Class;          -- Remarquer le dispatching sans pointeur
Enregistrer(RC);
```

de la procédure Process_Réservation (qu'on a vu), "Enregistrer" est un appel avec dispatching. La valeur du Tag de RC est utilisée pendant l'exécution pour déterminer quelle procédure "Enregistrer" doit être activée.

**Remarque** : le dispatching sans pointeur. En C++, il faut un pointeur (car la notion de T'Class n'existe pas).

D'un autre coté, un appel tel que

```
NR : Réservation_Agréable;
Enregistrer(Réservation(NR));
```

dans le Body du paquetage Système_de_Réservation n'est pas un appel avec dispatching car le type exact de l'opérande (Réservation) est connu à la compilation.

- Il est aussi possible de dispatcher sur le résultat d'une fonction quand le contexte de l'appel détermine un type spécifique.

Pour montrer ces cas :

```
Package Exemple IS
    Type T is tagged ....;
    Procédure P(X : T; Y : T);
    Function F return T;
    Function G(Z : T) return T;
    Procédure Q(U : T; V : T := F);
    Type TT is NEW T With ...;          -- TT hérite des opérations de T
    ...
End Exemple;
```

P, Q, F et G sont des opération à Dispatch de T (pour quoi?).
Les paramètres U, V, X, Y et Z sont des **opérande-contrôles** (controlling-operand) et les résultats de F et G sont des **résultat-contrôles** (controlling results). La procédure Q a un paramètre par défaut initialisé à l'appel de F.

Les fonction F et G sont à "cotrolling results".

**Principe** : tous les *opérande-contrôles* et les *résultat-contrôles* d'un appel doivent être **du même type**.

Si ces contrôles sont déterminés statiquement alors les vérifier à la compilation.
Si ces contrôles sont déterminés dynamiquement (comme le cas des variables class-wide) alors encore les valeurs actuelles doivent être d'un même type spécifique (test d'égalité des TAG) et un **Constraint_Error** est levé si ce test échoue.
Pour éviter la complication, le cas où certains opérandes sont dynamiques et d'autres statiques **n'est pas autorisé**.

Un exemple récapitulatif de ces cas (n répète les déclarations) :

**Exemple :**

```
Package Exemple IS
    Type T is tagged ....;
    Procédure P(X : T; Y : T);
    Function F return T;
    Function G(Z : T) return T;
    Procédure Q(U : T; V : T := F);
    Type TT is NEW T With ...;          -- TT hérite des opérations de T
    ...
End Exemple;

-- utilisation et appels
A, B : T;
AA, BB : TT;
C : T'Class := ...   -- Doit être initialisé car variable class-wide
D : T'Class := ...   -- Doit être initialisé car variable class-wide

P(A, B);             -- Sans Dispatch. Type T.
P(AA, BB);           -- Sans Dispatch. Type TT
P(C, D);             -- Dispatching. ********** L'appel a lieu si (C' tag = D' tag) **********

-- Deux cas ILLEGAUX détectés à la compilation
P(A, BB);            - ILLEGAL (mélange de types spécifiques). Pourtant TT dérive de T. Il faut convertir ?
P(A, C);             - ILLEGAL (mélange statique/dynamique)

-- Les fonction F et G à "controlling results"
P(A, F);             -- Sans Dispatch. Type T.
P(C, F);             -- Dispatching.
```

- Les fonction F et G sont à "cotrolling results".

- Dans **P(A, F)**; le controlling-operand A est statique => l'appel de F est statique. Puisqu'il n'y a pas de mélange possible donc F doit être statique.
- Dans le dernier cas **P(C, F)**; C est dynamique. Donc, parmi les F possibles, on prend celle avec le même type que celui de C. Il n'y a pas de vérification car un seul "controlling-operand" est utilisé pour déterminer le type. L'appel de F est comme un caméléon et s'adapte aux circonstances. On dit que cet appel est "Tag indeterminate".

- On peut avoir une **imbrication** de situations.

Dans

```
P(C, G(D));          -- Dispatching.
```

Même raisonnement que ci-dessus. En plus, les Tags de C et de D sont vérifiés et doivent être **identiques** sinon il y aura Constraint_error.

- Un cas tel que G(F) est "indeterminate" et on peut avoir :

```
P(A, G(F));          -- Sans Dispatch.
P(C, G(F));          -- Dispatching.
```

Dans **P(C, G(F))**; l'appel de G est déterminé par le type spécifique de C; ce qui détermine l'appel de F à son tour.

- On peut avoir un appel à F qui détermine une valeur par défaut :

```
Q(A);          -- Sans Dispatch. Noter le second paramètre par défaut de Q.
Q(C);          -- Dispatching.
```

Dans **Q(A)**; l'appel par défaut de F dans la procédure Q est statiquement défini et sera de type T.
Dans **Q(C)**; l'appel par défaut de F dans Q est dynamiquement défini par le type spécifique de la valeur de C.

L'expression par défaut d'un "controlling-operand" doit être "Tag indeterminate" et doit être un appel de fonction comme **F** ou comme **G(F)**. Pas autrement ?

**La raison** : on doit pouvoir utiliser l'expression par défaut dans les deux contextes dispatching ou sans Dispatch.

- Une autre utilisation d'une expression "indeterminate" est comme une valeur initiale d'un objet class-wide ou comme un paramètre effectif class-wide. Par exemple

```
C : T'Class := F;          -- Sans Dispatch. initialisation de class-wide
```

Dans ce cas, le type de F est **statiquement** déterminé = **T**. (Ne pas oublier que l'on peut avoir beaucoup de F; ne serait-ce que par dérivation et héritage).

- Dans **P(F, F)**; on a une opération **illégale** car l'appel est ambiguë. Par l'héritage, on ne sait pas si l'on manipule **P**et **F** de type **T** ou **TT**. La résolution échoue. Il n'y a pas de dispatching car il n'y a pas de class-wide.

- Il n'est pas possible pour un sous programme d'avoir des "controlling operand/result" de différents types taggés.
- On peut bien sur déclarer deux types taggés dans un même paquetage, mais on ne peut pas déclarer un sous programme dans ce paquetage avec des "operand/result" des **deux** types.

On peut bien sur faire ceci hors du paquetage auquel cas le sous programme n'est pas hérité et il n'est plus question de Dispatching.

L'opérateur d'égalité est une opération primitive qui est dispatché dans le cas général. **Mais des règles légèrement différentes s'appliquent à l'égalité** (résumés dans l'encadré ci-dessous):

- Si l'on compare deux valeurs class-wide et elles ont des Tags différents alors le résultat est FALSE plutôt que de lever une exception Constraint_Error qui sera levée dans le cas des autres opérations avec deux "controlling operand".
- Si l'on affecte une valeur class-wide à un objet d'un type class-wide, on aura une Constraint_Error si les Tags ne sont pas les mêmes.
- Un objet class-wide est contraint par sa première valeur.

Il y a une analogie forte avec les opérations sur les tableaux. On peut affecter un tableau à un autre seulement si leur tailles sont identiques. Le test d'égalité de deux tableaux ne lève jamais une Constraint_Error mais renvoie simplement FALSE si les tailles sont différentes. Mais les opérations telles que **AND** et **OR** sur des tableaux mono dimensionnels lèvent une Constraint_Error si les tailles sont différentes.

Pour résumer

```
*   class-wide_valeur = class-wide_valeur                 =>  False si Tags différents (no Constraint_Error)
*   class-wide_valeur autres_Oper_bin class-wide_valeur   =>  Constraint_Error si Tags différents
*   class-wide_type := class-wide_valeur                  =>  Constraint_Error si Tags différents
*   Array := Array                                        =>  Constraint_Error si Tailles différentes
*   Array = Array                                         =>  False si Tailles différents (no
Constraint_Error)
*   Array_mono and_or Array_mono                              =>  Constraint_Error si Tailles différentes
```

Par rapport à l'héritage, également, l'égalité est différente car elle est toujours prédéfinie et disponible.

On peut décider de redéfinir l'égalité mais l'héritage d'un tel opérateur peut amener des surprises.
**Exemple** : soient deux objets (géométriques) et l'opérateur d'égalité : supposons que deux objets sont égaux s'ils sont placés au même point (avec une variation Delta).

```
Function "=" (A,B : Objet) return boolean is
Begin
    return ( (A.X - B.X) ** 2 + (A.Y - B.Y) ** 2) < Delta**2);
End "=";
```

Le "Point" et le "Cercle" héritent de cet opérateur. "Cercle" oubliera complètement son Rayon. D'un autre coté, si l'on ne redéfinit pas "=", l'opérateur prédéfini exigera que tous les composants de deux cercles soient identiques, y compris les Rayons. On a donc deux comportements complètement différents.

**La solution** est de ne pas hériter "=" pour "Cercle". On peut le redéfinir pour "Cercle" où les centres devraient être identiques ainsi que leur Rayons.
Pour l'opérateur "=" :

Une règle est d'utiliser l'opérateur "=" éventuellement redéfini pour les parties parentes et puis tester l'égalité des composants ajoutés.

## Conversions et Redispatching

* Les conversions vers le root de l'arbre du type taggé sont permises. On a converti Réservation_Agréable vers Réservation par :

```
Enregistrer (Réservation(R_Ag));
```

Pour convertir dans l'autre sens, il faut utiliser **With** même s'il n'y a aucun composant nouveau.

On peut convertir une valeur class-wide vers un type spécifique :

```
R_Ag := Réservation_Agréable(R_cl);          -- Avec R_cl : Réservation'Class
```

**Il y aura un test run-time** pour savoir si la valeur courante de R_cl est d'un type pour lequel la conversion est possible.
Ici, R_cl doit être de type Réservation_Agréable ou un de ses types dérivés de sorte que la conversion soit pas hors la racine (ou la branche de ?) l'arbre.

En d'autres termes : dans "R_Ag := Réservation_Agréable(R_cl)"
**on teste si la valeur de R_cl est dans Réservation_Agréable'Class.** On aura Constraint_Error si non.

- La plupart des conversions (y compris des types taggés) sont des **changements de vue**. L'objet n'est pas modifié mais on le regarde autrement. Par exemple :

```
Enregistrer (Réservation(R_Ag));
```

est un changement de vue. Par "Réservation(R_Ag)", on ne voit plus les composants de R_Ag relatif à la Réservation_Agréable (ils sont masqués) mais l'objet R_Ag n'a pas changé.

**Mais le tag désigne toujours l'objet d'origine** qui peut d'ailleurs être du type Réservation_Luxe.

- De plus, si l'on fait une affectation

```
R_Ag := Réservation_Agréable(R_Luxe);
```

**Le tag de R_Ag ne change pas**. Seuls les composants de Réservation_Agréable sont copiés depuis R_Luxe.

On peut même avoir une conversion de vue comme destination de l'affectation :
**Réservation_Agréable(R_Luxe) := R_Ag ;**
Dans R_Luxe, on copie seulement les composants pour Réservation_Agréable. Les autres ne changent pas.

**- Un principe important :**

Le tag d'un objet (spécifique ou class-wide) ne change jamais.

- De ce fait découle le **Redispatching**.

Après un dispatching, il arrive souvent que l'on applique encore une opération (héritée) à l'objet qui entraîne un autre dispatching vers le type d'origine. C'est pour cela que le tag ne doit pas changer.
Par exemple :

```
Procédure Enregistrer (R_A : in out Réservation_Agréable) IS
Begin
    Enregistrer(Réservation(R_A));        -- comme une réservation de base
    Cmd_Repas(R_A);                       -- retour à l'objet d'origine
End Enregistrer;
```

L'appel à "Cmd_repas" ne nécessite pas de dispatching. Dans "Cmd_Repas", on peut tester l'origine du paramètre pour par exemple éviter de commander un repas luxe (! Why not?, les repas Luxe sont mieux !)

On aurait pu écrire (pour montrer le propos seulement) :

```
Procédure Cmd_Repas(R_A : Réservation_Agréable) IS      -- Ce n'est pas le vrai Cmd_Repas
    R_A_C : Réservation_Agréable'Class := R_A;              -- initialisation immédiate
Begin
    If (R_A_C in Réservation_Luxe) Then                     -- supposons qu'on a le droit !!
            -- commander un repas Luxe
    Else    -- commander un repas Agréable
    End if;
End Cmd_Repas;
```

NB moi : doit on passer forcément par "R_A_C" pour faire le test ? Apparemment OUI. Voir règle de Dispatch.

La conversion dans le class-wide permet de regagner le type d'origine.

On aurait pu éviter l'affectation ci-dessus par :

```
R_A_C : Réservation_Agréable'Class renames Réservation_Agréable'Class(R_A);
```

Ou écrire le test :

```
If (R_A_C' Tag = Réservation_Luxe' Tag) then ...
```

Rappelons que l'

On ne peut pas appliquer l'attribut TAG à un objet de type spécifique

**On n'aurait donc pas pu faire l'économie de la déclaration de R_A_C.**

- L'écriture de "If (R_A'Tag = Réservation_Luxe'Tag) Then .." est interdite car le TAG peut être complètement différent alors que le contenu de l'objet correspond à son type actuel, différent du Tag.

- Dans la **fausse version** de Cmd_Repas, il est bien sur contre l'idée de la programmation par extension (P.O.O) qui doit permettre d'écrire Cmd_repas sans considérer les extensions futures du système. De plus, le type Luxe est créé plus tard et dans un autre paquetage NON visible ici.
L'approche propre est de re-dispatcher. On écrit Cmd_Repas(L_R : Réservation_Luxe) et on redispatche dans :

```
Procédure Enregistrer (R_A : Réservation_Agréable) Is
Begin
    Enregistrer(Réservation(R_A));
    Cmd_Repas(Réservation_Agréable'Class(R_A));          -- Redispatch.
End Enregistrer;
```

N.B. moi : ca n'est pas évident de penser à la conversion. Que se passe-t-il si on oublie la conversion.? Peut être que l'appel "Cmd_Repas(R_A)" active simplement celle de Réservation_Agréable même si en fait, l'objet est Luxe. Mais alors, s'il n'y a pas d'affectation, les conversions s'enchaînent et ca sera bon (?). Au fait, arrive-t-on dans cette Procédure Enregistrer pour une réservation Luxe ???

Redispatching a lieu car on a converti dans class-wide. Rappelons que

Dispatching a lieu seulement si le paramètre effectif est class-wide et le paramètre formel est d'un type spécifique.

Donc ici, nos passagers Luxe auront bien leur repas Luxe!.
- Il y a le même problème avec Choix_Siège.

- Le fait de ne pas changer de Tag dans une conversion explique pourquoi l'on peut déclarer une procédure Non_Abstract pour un type Abstract (Personne) comme la procédure "Print_Détails" qu'on a vue pour le type Personne. On n'aura jamais un objet de type Personne et alors un dispatching pour cette procédure n'est pas possible. Par contre, un appel statique avec une conversion (de vue) est possible :

```
Procédure Print_Details(W : Femme) Is
Begin
    Print_Details(Personne(W));
    Print_Integer(W.Enfants);
End Print_Details;
```

Le tag ne changera pas et on est sauf !.
- Cette règle + le principe de dispatching d'un class-wide paramètre effectif permettent d'éviter les surprises que l'on rencontre dans certains langages.

**Les points importants :**
* Un type record peut être taggé. Un type Private peut être Taggé.
* Ces valeurs ont un Tag. Le Tag donne le type Spécifique.
* Un type taggé peut être étendu par dérivation.
* Le tag d'un objet ne change jamais.
* Les objets ont des primitives prédéfinies.
* Les primitives peuvent être surchargées.
* Si la dérivation a lieu dans une spécification de paquetage, alors d'autres primitives peuvent être ajoutées.
* Les types et les sous programmes peuvent être Abstract.
* Un sous programme Abstract n'a pas de corps mais on peut en fournir par dérivation.
* Seul un type TAGGED ABSTRACT peut avoir des Abstract sous programmes.
* T'Class dénote le type class-wide dont la racine est T.
* T'Class est un type indéfini.
* Un type access peut pointer n'importe quelle valeur de T'Class.
* La conversion de types doit être vers le root.
* La conversion implicite d'un type spécifique vers un type ancètre class-wide est permise.
* Les paramètres de type TAGGE sont toujours passés par référence.
* Ils sont également considérés ALIASED pour que l'attribut ACCESS puisse s'appliquer.
* L'appel d'une primitive avec un paramètre actuel de type class-wide provoque le dispatching : choix de la primitive selon le TAG.

Le tag fait partie de l'objet mais c'est un détail d'implantation. Une implantation peut décider que le tag pourrait être absent quand on n'en a pas besoin et le créer quand une valeur class-wide est créée.

## Type Private et extensions

- On a vue dans le paquetage Queues :

```
Type Element is Tagged Private;                -- on dit que c'est une vue partielle ou externe d'Element
```

La déclaration complète (la vue complète) doit aussi être taggée.

On aurait pu déclarer Element de type Abstract aussi :

```
Type Element is Abstract Tagged Private;
```

Ca sera un avantage car l'utilisateur ne déclare pas des objets de type Element.

Règle : La vue complète (interne) doit respecter les propriétés de la vue partielle (vue externe).
Si la vue externe est taggée alors la vue interne doit aussi être taggée mais l'inverse n'est pas vraie.

**La vue externe peut être non-taggée alors que la vue interne est taggée.** Dans ce cas, l'utilisateur ne pourra pas faire de l'extension.

**L'inverse** de cette règle s'applique à **Abstract**.

Si la vue externe est Abstract, la vue interne **n'est pas obligée** d'être Abstract mais l'inverse n'est pas vraie : **si la vue partielle n'est pas Abstract alors la vue interne ne peut pas être Abstract..**

| Si Vue partielle (interne) est | Vue complète (externe) peut/doit être |
|---|---|
| Tagged | Tagged (obligé) |
| UnTagged | Libre (pas d'extension par les clients) |
| Abstract | Libre |
| NonAbstract | NonAbstract (obligé) |

**Quelques points liés à Abstract-Private et Agrégats.**

* On doit utiliser un agrégat normal si tous les composants sont visibles (comme d'habitude).
* On peut utiliser un agrégat d'extension (extension avec With) si la partie ancètre est **Private** :

```
Un_Element : Element;       -- de ci-dessus (Private)
..
(Un_Element with ...)
```

* Si le type est "Abstract" tel que l'objet Un_element ne peut pas être déclaré alors **on peut simplement utiliser le nom** du (sous) type (ici, Element):

```
(Element With ...)          -- le type Element est ici Abstract Private, on utilise son nom
```

et tout composant correspondant au type Element seront initialisés par défaut par ce qui a été prévu (puisqu'on se sert d'Element pour construire quelque chose).

- Le type Element montre que l'

on peut étendre un type Taggé Private avec des composants additionnels visibles à l'utilisateur même si les composants originaux sont cachés de l'utilisateur (le type Element est Private – voire Abstract – et on peut l'étendre)

* L'inverse est vrai. On peut **étendre un type existant** avec des composants et garder **ces composants** additionnels **cachés** de l'utilisateur.

On peut déclarer un type Private "Forme" et montrer qu'il dérive du type "Objet" mais garder ses composants additionnels cachés :

```
Package Forme_Cachée Is
    Type Forme is New Objet With Private;          -- La dérivation est visible dans la vue externe
    ...
Private
    Type Forme is New Objet With                   -- La vue Interne
    record
        ...  -- Composants Private
    End record;
    Function Surface (F : Forme) return Float;     -- Une fonction Privée. A quoi elle va servir
End Forme_Cachée;
```

Dans ce cas, il n'est pas nécessaire que la déclaration complète de Forme de dériver directement d"Objet". Il peut y avoir une chaîne de dérivations intermédiaires ("Forme" peut dériver de "Cercle" !!). Tout ce qui importe est que Forme dérive ultimmement d'Objet. S'il n'y a pas de composants additionnels, on écrit alors "with null record".

Bien sûr, Forme ne peut jamais être un type existant (en dériver quelque chose ?). Ecrire

```
Private
...                                        -- Forme dérive de Cercle !!!
Type Forme is New Cercle with null record; -- Forme est  Private, pas Cercle.
```

dans la partie privée ne rend pas Forme identique à Cercle mais une dérivé de Cercle.

- On peut déclarer et surcharger des primitives (comme Surface) dans la partie privée ou non privée. Les primitives nouvelles ou surchargées dans la partie Private amènent des possibilités intéressantes.

Une nouvelle primitive dans la partie Private crée un slot dans la table de Dispatch même si elle n'est pas visible par tout.

Il y a une seule table de Dispatch par type et il se peut que certaines opérations ne soient pas visibles dans toutes les vues.

Aussi,

Un type Abstract n'a pas d'opération Private Abstract car l'on ne peut pas les surcharger et donc le type ne peut pas être étendu.

Ceci serait une violation de l'abstraction et l'utilisateur ne verra pas l'opération Private si l'on le laissait la surcharger.

Un autre restriction similaire est qu'

une fonction avec un "controlling-result" ne peut pas être une opération Private car elle ne peut pas être surchargée.

(car Private ?. (redire ici ce que c'est que "controlling-result" : un taggé ou class-wide ?)).

Enfin, quand on surcharge une opération héritée comme Surface pour Forme dans la partie Private. Bien qu'elle ne soit pas directement visible, l'opération surchargée peut être utilisée par un appel direct ou indirect via dispatching.
C'est une principe important que la même opération soit appelée directement ou indirectement pour le même Tag; cela permet de tester un fragment avec le binding (lien) statique et on penser qu'elle fonctionnera lors d'un dispatching.
- Une curiosité :

un renommage dans un paquetage avant la surcharge se réfère toujours à l'ancienne version;

Que la surcharge soit dans la partie privée ou non.! (le renommage **après** la surcharge réfère la nouvelle version)

```
Function old_surface(F : Forme) return Float renames Surface;
...
Function Surface(F : Forme) return Float;
```

Le renommage après la surcharge réfère la nouvelle opération.

Tous les renommages dans une spécification de paquetage sont comme des opération primitives et ont leur entrée dans la table de dispatching, initialisée avec l'opération COURANTE au moment de renommage.

Ces noms nouveaux (donnés dans le renommage) sont des opérations primitives et **peuvent donc être surchargées** dans les héritages ultérieurs. Ceci peut paraître bizarre car le principe de renommage est de NE PAS créer de nouvelle entité. Mais ce principe est toujours vrai: le nouveau slot donne une autre manière de référencer des entités existantes.

Un bon exemple d'extensions private est donné en considérant le système de réservation. Pour le propos ici, la visibilité totale des types individuels n'est pas nécessaire.

## Exemple d'Extension Private (réservation de voyage)

```
Package systeme_de_reservation  IS
Type reservation is Abstract Tagged Private;
Procedure Enregistrer(R : In Out  Réservation);
Type Reservation_simple Is NEW Réservation With Private;
Type Reservation_agréable Is NEW Réservation With Private;
Type Reservation_Luxe Is NEW Reservation With Private;   -- Ce n'est pas une erreur  (v. +bas)

PAIVATE
Type reservation is tagged
    record
      Num_vol : Integer;
      Date_voyage : Date;
      Num_siege : String(1..3) := "   ";
    End record;
Procedure Choix_siege(R : In Out  Réservation);                    -- Affecter un siège (e.g. "56A")

Type Reservation_simple Is NEW Réservation With NULL record;

Type position IS (couloir, fenêtre);
Type Type_repas IS (végétarien, poisson, viande);

Type Reservation_agréable Is NEW Réservation With
  record
     genre_siege : position;
     Repas : Type_repas;
  End record;
Procedure Enregistrer(RA : In Out Reservation_agréable);          -- Surcharge
Procedure Choix_siege(R : In Out  Réservation_agréable);          -- dito
Procedure Cmd_repas(RA : In Reservation_agréable);                -- Nouvelle
```

```
Type Reservation_Luxe Is NEW Reservation_agréable With          -- Dans la vue externe, c'était
  record                                                        -- Réservation
     Destination : Adresse;
   End record;
Procedure Enregistrer(RL : In Out Reservation_Luxe);      -- surcharge
Procedure Cmd_repas(RA : In Reservation_Luxe);            -- dito
Procedure Reserve_Limo(RL : In Reservation_Luxe);         -- Nouvelle
End systeme_de_reservation;
```

La seule chose visible à l'extérieur : les types et la procédure *Enregistrer*. Les relations entre les types ne sont pas visibles car on voit seulement qu'ils dérivent de Réservation. Cet exemple montre que

dans la déclaration complète (interne) d'un type Private, on n'est pas obligé de dériver directement de l'ancètre donné dans la partie partielle (partie externe).

Dans l'exemple, Réservation_Luxe dérive de Réservation_Agréable alors qu'on avait donné Réservation tout court dans la déclaration partielle.

**NB moi :** que fait on alors si par ce passage, on se récupère **des choses en plus**. Ici, "Réservation_Luxe" va avoir en plus les attributs "Repas" et "Genre_Siège" alors qu'en dérivant de Réservation, il ne les aurait pas eu. ? Peut être que dans la partie externe, on indique le root alors que dans la partie interne, on donne l'un des ancêtres ? De même pour les primitives ? De plus, dans les dessins des méthodes OOA, on n'aura pas la même chose. Faut Rester abstrait ? Remarquons aussi que "Choix_Siège" et "Cmd_Repas" sont des Primitives : on a les différentes versions proprement pour que les voyageurs aient leur choix de Siège et de Repas. NB. Moi : on ne fait pas jouer l'héritage ?

On avait vu que l'on ne pouvait pas ajouter des primitives supplémentaires à un type dès qu'une dérivation de ce type a eu lieu (cela fige le type).
Une déclaration d'instance gèle également le type et empêche l'ajout de primitives.

La représentation du type est dite "gelée" (voir Chap. 21 Barnes) car elle détermine la table de dispatching. En plus,

dériver un type gèle son parent, s'il ne l'a pas été. Cette règle s'applique seulement à la définition complète, non à la partielle.

Sinon, on n'aurait pas pu déclarer la Primitive Private "Choix_Siège" qui est dispatching (il n'y en a pas pour Luxe). (NB. Moi : est-ce parce que type Private ? ou vrai pour tout type taggé?).

**La déclaration complète de par exemple "Réservation_Simple" gèle le type Réservation** et on ne peut plus y ajouter d'autres Primitives. Un autre point :

On ne peut dériver d'un type Private que s'il a été complètement défini(déclaré). Il est donc important que divers types et primitives soient déclarés dans le bon ordre.

On peut maintenant ajouter la déclaration du paquetage SuperSonic :

```
With Systeme_de_Reservation;
Package Systeme_Reservation.SuperSonic IS          -- C H I L D

Type Reservation_ SuperSonic Is NEW Réservation With Private ;   --  Réservation visible ?

Private
Type Reservation_ SuperSonic Is NEW ....    With                 -- voir ci-dessous
Record
     Champagne : millésime;  .....
End record;

Procedure Enregistrer(RA : In Out Reservation_SuperSonic);          -- surcharge (voir ci-dessous)
Procedure Choix_siege(R : In Out  Réservation_ SuperSonic);         -- overriden (voir ci-dessous)
......
End Systeme_Reservation.SuperSonic;
```

**Remarque** : Réservation est visible et un CHILD n'a pas besoin de clause with/use car un CHILD a accès à tout du père, y compris la partie Private.

Le type **Reservation_ SuperSonic** peut maintenant une extension de **n'importe quel membre de l'arbre**. Le paquetage est CHILD, il peut voir les détails des types et appeler leur opérations.
Notons que **Choix_siege** surcharge proprement malgré le fait d'être une Private privée de Réservation. C-à-d, Choix_Siège étant Private à Réservation, on pourrait penser que cette version de SuperSonic est une primitive qui ne surcharge pas les autres; mais ce n'est pas le cas et Choix_Siège surcharge bien car ELLE VOIT TOUT de son père (Child). On n'a pas le même problème avec Enregistrer qui n'est pas Private.

## Type Contrôlés

Ces types permettent le contrôle total des initialisations et des finalisations (l'avant destruction) des objets ainsi que **l'affectation définie par l'utilisateur**.
Principe général : il y a 3 sortes d'activités primitives concernant le contrôle des objets :
- Initialisation après Création;
- Finalisation avant Destruction;
- Ajustement après Affectation.

L'utilisateur a la possibilité d'appeler des procédures et faire le nécessaire à différents points de la vie d'un objet. C'est procédures sont : **Initialize**, **Finalize** et **Adjust** qui prennent l'objet (this ?) en **paramètre**. Elles sont appelées AUTOMATIQUEMENT (faut utiliser le paquetage Ada.Finalization).
Quand une de ces procédures est appelée (par le code du compilateur), on dit que **l'objet a été contrôlé**.

**Exemple :**

```
declare
    A : T;              -- Création de A, Initialize(A) appel automatique de la procédure de contrôle
Begin
    A := E;             -- Finalize(A), Copie la valeur, Adjust(A)
    .....               -- Finalize(A)   car A va disparaître
End;
```

L'objet A est déclaré, toute initialisation "normale" et par défaut est faite, puis **Initialize** est appelée.
Lors de l'affectation, **Finalize** est d'abord appelée pour récupérer l'objet qui va être réécrit et donc détruit. Puis la copie physique (de E dans A) est faite puis on appelle **Adjust** (pour A) pour faire ce qu'il faut pour la nouvelle copie. A la fin du bloc, **Finalize** est appelée avant la destruction de A.

Dans A := E, l'objet A est **finalisé** avant de recevoir une valeur par affectation. A la fin de l'affectation, A est **ajusté**.

Dans le cas de blocs et structures imbriqués où les **composants** internes peuvent être contrôlés, la règle est :

**Règle** : les composants (des objets de composition) sont initialisés et ajustés avant l'objet lui-même. Lors des finalisations, tout est fait dans l'ordre inverse.

Il y a beaucoup de cas où les contrôles sont appelés : appel des allocateurs (new...), évaluation d'agrégats, ... les détails ne sont pas donnés mais le principe est simple.

**Appels des Procédures de contrôle :**

*** Initialize :**
- à la création et après toute initialisation par défaut ou autre, Initialize est appelée.

*** Finalize :**
- à une affectation, on appelle d'abord Finalize pour l'objet LHS (qui va disparaître)
- à la fin des blocs, Finalize est appelée avant la destruction des variables.

*** Adjust :**
- après l'affectation physique, Adjust est appelée pour "ajuster" la nouvelle copie.

*** Ordre :**
Création/Ajustement : les composants puis l'objet
Destruction : ordre inverse.

Pour qu'un type soit contrôle, il doit être une extension d'un des (deux) types taggés déclarés dans le paquetage de la librairie **Ada.Finalization** dont la spécification est :

```
Package Ada.Finalization Is
    Type Controlled is Abstract Tagged Private;

Procedure Initialize(Object : In Out Controlled);
Procedure Adjust(Object : In Out Controlled);
Procedure Finalize(Object : In Out Controlled);

Type Limited_Controlled is Abstract Tagged Limited Private;
Procedure Initialize(Object : In Out Limited_Controlled);          -- pas d'Adjust car Limited (no ":=")
Procedure Finalize(Object : In Out Limited_Controlled);

Private
.......
End Ada.Finalization;
```

Pas **d'Adjust** pour "Limited_Controlled" car Limited ne permet pas la copie (ou l'affectation).

**Exemple simple** : on crée des objets, compter le nombre d'objets (instances, valeurs) existants, noter un numéro d'identité de l'objet dans l'objet lui-même...

```
With Ada.Finalization; Use Ada.Finalization;
Package Choses_tracées Is
    Type Chose is new Controlled with
    record
        Num_Id : Integer;
            ..... autres données
    End record;
Procedure Initialize(Object : In Out chose);          -- Surcharge
Procedure Adjust(Object : In Out chose);
Procedure Finalize(Object : In Out chose);
End Choses_tracées;
-------------------------------------------------------------------------------
Package Body Choses_tracées Is
    Compteur : integer := 0;
    Prochain : Integer := 1;
    Procedure Initialize(Object : In Out chose) is
    Begin
        Compteur := Compteur + 1;            -- Compter les objet : un en plus
        Object.Num_id := Prochain;           -- Noter le numéro d'identité de l'objet dans l'objet
        Prochain := Prochain +1;                  -- Le numéro du prochain objet
    End initialize;

    Procedure Adjust(Object : In Out chose) renames Initialize;       -- Voir ci-dessous

    Procedure Finalize(Object : In Out chose);
    Begin
        Compteur := Compteur - 1;            -- Compter les objets : un en moins
    End Finalize;
End Choses_tracées;
```

Dans cet Exemple, chaque instance (valeur) de "chose" est une nouvelle valeur et donc "Adjust" = "Initialize". Ainsi, même lors d'une affectation d'une instance déjà existante, on considère une nouvelle création. Le renommage fournit ainsi un corps de procédure pour Adjust. On peut faire autrement (Exercice).

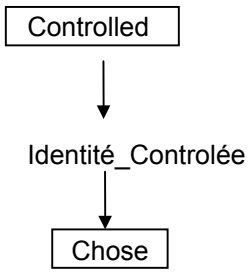
**Remarque** : le "Num_Id" de chaque "chose" est visible. Ce qui peut être parfois dangereux.

**Pour éviter ceci, on peut déclarer un paquetage CHILD dans lequel on étend le type (Controlled).**

Ceci permet d'avoir des vues différentes de l'objet et de créer un type avec des données cachées (Son numéro d'identité) et des visibles (en passant par une hiérarchie plus grande.

**Exemple de type contrôlé avec des données Private et des données visibles :**

```
Package Choses_tracées Is
    Type Identité_Controlée is Abstract Tagged Private ;
Private
    Type Identité_Controlée is Abstract New Controlled with
    record
        Num_Id : Integer;          -- Données cachées
    End record;

    Procedure Initialize(Object : In Out chose);
    Procedure Adjust(Object : In Out chose);
    Procedure Finalize(Object : In Out chose);
End Choses_tracées;

Package Choses_tracées.Vue_Utilisaateur  is        -- C H I L D : Accès à tout du père
    Type Chose is New Identité_Controlée with
    record
        -- Données visibles.
    End record;
Choses_tracées.Vue_Utilisaateur;
```




Le type "Identité_Controlée" est Abstract. Donc pas d'instance de ce type et l'utilisateur ne verra même pas que c'est un "Controlled-Type". Les **primitives** "Initialize", "Adjust" et "Finalize" seront dans la partie Private et elles **seront également cachées.**

Le paquetage CHILD n'est pas vraiment nécessaire ici car rien n'est partagé; cependant le nommage commun (?) est utile.
De plus, des types peuvent dériver de "Identité_Controlée" tous partageant le même mécanisme de contrôle.
Une autre façon de faire serait de déclarer d'abord les données visibles puis déclarer les données invisibles (Num_Id).

- Les types **controlled** et **Limited_Controlled** sont des types Abstraits.
- Les procédures **Initialize**, **Adjust** et **Finalize** ne sont **pas Abstraits**. Elles ne font rien par défaut et on peut les **surcharger**.

Une raison pour rendre ces procédures Null par défaut est que souvent pour un type, "Finalize" doit appeler Finalize du type parent. On a:

```
Type T is new parent with
    record
        -- Composants supplémentaires
    End record;
.....
Procedure Finalize(Object : in out  T) is
Begin
    -- Opérations pour finaliser les composants supplémentaires
    Finalize(Parent(Objet));          -- Finaliser les données du Parent
End Finalize;
```

Si le parent est un paramètre formel générique, tout ce que l'on peut savoir est que le Parent est "type contrôlé" et une procédure Finalize Null peut être appelée sans difficulté.

Les "Controlled-Types" sont un bon exemple d'utilisation d'agrégat d'extension où la partie ancêtre est juste donnée par une marque de "Subtype". On peut typiquement écrire (l'ancêtre "Controlled" est juste nommé) :

```
X : T := (Controlled with ...);      --déclaration de X suivie d'une affectation d'initialisation
```

Remarquons que l'on ne peut pas facilement donner une expression pour la partie ancêtre car il est Abstract.

# Abstraction et implantations multiples

Une des caractéristiques de la POO : différentes implantations pour une même abstraction.
Les types taggés et l'héritage permettent aux types d'être traités comme différentes réalisations d'une même abstraction.
**Le tag d'un type indique son implantation et permet une lien dynamique entre le client et l'implantation appropriée.**

On peut traiter différentes implantations d'une abstraction telle qu'une famille de types ensemble en commençant par un type Abstract :

```
Package Abstract_Set is
    Type Set is Abstract tagged null record;          -- rien pour l'instant

    function Empty return Set is Abstract ;                -- Ensemble vide
    function Unit(E : Element) return Set is Abstract ;    -- Construire un Set à un élément
    function Union(S, T : Set) return Set is Abstract ;
    function Intersection(S, T : Set) return Set is Abstract ;
    Procedure Take(From : in out Set;  E : out Element) is Abstract ;   -- E est un élé arbitraire
End Abstract_Set ;
```

On suppose que le type Element est un subtype discret tel que Integer ou Couleur.
Le paquetage définit une collection de Primitives abstraites pour le type Abstrait Set.
Une implantation de l'abstraction peut dériver du type Set (root) avec des composants et surcharger les primitives.

Une implantation possible serait avec des **booléens**. Un tableau de booléens où chaque élément représente la présence ou l'absence d'un membre de l'ensemble.

```
With Abstract_set;              -- Element est un type discret comme Integer, Couleur, Char...
Package Bit_Vector_Sets Is
    Type Bit_Set is New Abstract_Set.Set with Private;

    function Empty return Bit_Set;                  -- Ensemble vide
    function Unit(E : Element) return Bit_Set;      -- Construire un Set à un élément
    function Union(S, T : Bit_Set) return Bit_Set ;
    function Intersection(S, T : Bit_Set) return Bit_Set;
    Procedure Take(From : in out Bit_Set;  E : out Element) ;
Private
    Type Bit_Vector is array(Element) of boolean;
    Type Bit_Set is New Abstract_Set.Set with
    Record
        Date : Bit_Vector;
    End record;
End Bit_Vector_Sets ;
-------------------------------------------------------------------------
Package Body Bit_Vector_Sets Is
    function Empty return Bit_Set is
    Begin
        Return (Data => (Others => False));
    End Empty;
    function Unit(E : Element) return Bit_Set is
    S : Bit_Set := Empty;
    Begin
        S.date(E) := True;
        Return S;
    End Unit;
    function Union(X, Y : Bit_Set) return Bit_Set  is -- ZZZ : on a mis X,Y au lieu de S,T. Ca va?
    Begin
        return (X.Date or Y.Date);
    End Union;
    .....
End Bit_Vector_Sets ;
```

Cette mplantation conviendrait si le nombre d'éléments dans le Set n'est pas important (on pourrait utiliser le **Pragma Pack** pour compacter le tableau).

Une autre implantation : par une liste chaînée.

Pour cette version là, il faut définir **l'égalité** et **l'affectation** (par les démons ?? OUI. V. ci-dessous) ainsi que les opérations abstraites.

Le plus intéressant est d'avoir un programme qui convertit une implantation dans l'autre (remarquer **l'abstraction** des opérations **indépendantes** des implantations des Sets en paramètre)

```
Procedure convert(From : Set'Class; To : out Set'Class) is   -- Convertir les implantation
Temp : Set'Class := From;
E : Element;
Begin
     To := Empty;
     While temp /= Empty loop
          Take(Temp, E);
          To := Union(To, Unit(E));
     End loop;
End Convert;
```

## Détails de dispatching :

- Le paramètre From est copié dans Temp.
- La variable class-wide "Temp" doit être initialisée car un type class-wide est indéfini.
- Dans la commande **To := Empty;** : la fonction Empty a un "controlling-result" mais aucun "controlling-operand" (paramètre) pour déterminer son tag (savoir quelle version d'Empty appeler); **le choix de la fonction Empty à appeler doit être déterminé par le tag du paramètre class-wide "To" qui est la destination de l'affectation**.
- Dans la commande **While Temp /= Empty Loop** : le dispatching-opérateur "=" a deux "controlling-operand". Ils ne sont pas statiques et donc il doit y avoir un test dynamique pour voir si les tags sont identiques. Mais on a vu que "Empty" n'a pas de tag (tag-indeterminate); tout ce qui est fait est d'utiliser le tag de Temp pour déterminer quelle Empty et quel opérateur "=" appeler.
- Dans la commande **Take(Temp, E)**, il y a Temp (qui est single controlling-operand: Temp); Take va dispatcher selon le tag de Temp qui est celui de From.
- Finalement, dans **To := Union(To, Unit(E))**, on provoque un dispatching de "Unit" et de "Union" selon le tag de "To".

Fin Détails.

-Une propriété intéressante de "Convert" est que rien n'y pourra mal fonctionner : il ne sera pas possible que les "controlling operand" aient différents tags pour lever Constraint_Error. Il n'y a pas de test (contrôle) mais assez d'information pour dispatcher correctement.

- On a dit que l'affectation est aussi dispatching. Ce n'est pas toujours apparent. Dans l'exemple, si le type de "From" était une liste chaînée, alors une vraie copie serait nécessaire sinon la valeur d'origine peut être endommagée lorsque la copie (résultat de l'affectation ?) sera décomposée. Cette copie peut être faite par les "contrôles" (démons).
L'idée est que le type Ensemble est implanté par un record contenant un composant contrôlé interne. Ce composant est un accès (pointeur) sur une liste contenant les éléments. Lors que ce composant contrôlé est assigné (affecté), on fera une nouvelle copie de la liste complète.

Remarquons que le type Linked_Set (si on implante Set par une liste) ne peut pas être contrôlé comme un tout car il est dérivé directement de "Abstract_Sets.Set". Cependant, l'affectation d'une valeur à Linked_Set provoque l'affectation de son composant interne.

L'implantation peut être :

```
with Abstract_Set;
with Ada.Finalization; use Ada.Finalization;
Package Linked_Sets is
    Type Linked_Set is NEW Abstract_Sets.Set with private;
        -- différentes opérations sur une liste
Private
    Type Cell;
    Type Pointeur_Cell is access Cell;
    Type Cell is
        record
            E : Element;
            Next : Pointeur_Cell;
        End record;

    Function Copy(P : Pointeur_Cell) return Pointeur_Cell;          -- la copie
    Type inner is new Controlled with                    -- Sous type de "démon"
        record
            The_Set : Pointeur_cell;
        End record;

    Procedure Adjust(Object : in out Inner);
    Type Linked_Set is New Abstract_Set.Set with
        record
            Component : Inner;                           -- Ce qui est contrôlé
        End record;
End Linked_Sets;
-------------------------------------------------------------
Package Body Linked_Sets is
    Function Copy(P : Pointeur_Cell) return Pointeur_Cell is        -- la copie "deep"
    Begin
        If (P = Null) then return Null;
        Else return new Cell'(P.E, copy(P.Next));        -- on enchaîne récursivement sur copy.
        End if;
    End Copie;

    Procedure Adjust(Object : in out Inner) is        -- ZZZ le type ici n'est pas directement l'objet
    Begin
        Object.The_Set := Copy(Object.The_Set);
    End Adjust;
    .....
End Linked_Sets;
```

Le type visible Linked_Set est simplement un record qui contient un composant de type contrôlé inner. Rien de toutes les opérations de copie, des démons, ... n'est visible à l'utilisateur. On n'a pas besoin de fournir Initialize et donc pas de finalize. Mais on peut s'en servir si l'on veut récupérer l'espace non utilisé par les pointeurs. Il veut mieux avoir un ordre sur la liste pour simplifier l'opération d'égalité.

## Résumé O.O. d'ADA95

* type tagged : type extensible (= Classe)
* sous forme de record
* conversion dans les deux sens
* Package interne pour cacher des méthodes que l'on ne veut pas faire hériter.
* La racine : "tagged", les fils "With record.."
* Comme en C++, dans une classe fils, on appelle la routine de la classe mère (conversion bas-haut) puis on alimente les champs restants propre à la classe dérivée fils (constructeurs, PUT, ...)

* Type Int_ptr is access **ALL** Integer; -- recevoir l'adresse d'un entier
* Y : ALIASED Integer; -- pour manipuler l'adresse de Y
* P : Int_ptr := Y'access; -- l'attribut ACCESS donne l'adresse de Y

* **T'Class** : la classe du type taggé T : représente la hiérarchie de T
* Une variable class-wide doit être **immédiatement** initialisée.
* Type cl_ptr is ACCESS ALL T'Class;-- All + Class à ne pas oublier
* Dispatching Dynamique est lié aux pointeurs ? On passe soit par ALIASED, soit par une allocation par NEW.
* Peut on avoir une variable class-wide (pas de pointeur) et imposer dispatching ?
* Sans pointeur, peut on créer une situation où ADA ne pourrait pas faire du early-binding? A tester.
* L'équivalent de virtuel de C++ se réalise avec ABSTRACT.
* Une classe abstraite se déclare avec ce même mot (en C++, virtuel + code=0 donne l'abstraction)
* Un paquetage CHILD voit la partie PRIVATE du parent.
* Dans la partie private d'un paquetage, on peut mettre des restrictions (telle que LIMITED) qui ne seront appliquées que dans le Body sans que les clients soient concernés.
* Un paramètre ACCESS peut remplacer en général un paramètre IN OUT.
* On peut empêcher la déclaration des objets d'un type T en déclarant T Abstract. On pourra cependant avoir des routines R dont les paramètres seraient de type T. On dispatchera jamais dans R car il n'y aura jamais d'objet de type T mais R peut être utilisée par des dérivés de T.
* Si pour U dérivé de T, on ne donne pas de procédure concrète pour celles Abstract, alors U doit aussi être déclaré Abstract.
* Une fonction qui renvoie un type Abstract doit être Abstract.
* **Cas de fonction qui renvoie un type non abstrait mais, par la suite, ce type est étendu (Barnes 271) :**
Si une fonction renvoie un type non Abstract et puis, ce type est étendu, alors surcharger la fonction pour le type dérivé.

* Pour respecter TOUT FONCTION des TDA, on peut prendre des fonctions avec un paramètre ACCESS. E.g. dans FUNCTION DEPILER(P : ACCESS PILE) return ELEMENT, on renvoie l'élément en même temps que la pile est modifiée sans que le paramètre de la fonction soit touché – SAUF si la pile devient VIDE !!! A moins d'avoir toujours un élément bidon dans la pile.
* **Type xxx is record NULL; End record;** peut être remplacé par **Type xxx is NULL record;**
* Test de classe :
**P : Personne'Class;** -- On a personne dont "Femme" et "Homme" dérivent
**If P in Femme then ... end if;** -- l'objet P est de type Femme.
* Voir les points sur le dispatching de Barnes. (aussi le livre P. 272...).
* Il n'est pas possible pour un sous programme d'avoir des "controlling operand/result" de différents types taggés.
On peut bien sur déclarer deux types taggés dans un même paquetage, mais on ne peut pas déclarer un sous programme dans ce paquetage avec des "operand/result" des **deux** types. (On peut le faire hors le paquetage => pas d'héritage => non Dispatch)

D'autres éléments
* Paquetage Standard (p.770 de Feldman),
* Paquetage ADA.Text_Io (p.774 de Feldman),
* Paquetage ADA.Calender (p.781 de Feldman),
* Paquetage Math (p.782 de Feldman),
*Paquetage String en ADA95 (p.785 de Feldman)
* Les exceptions d'ADA95 (p.791 de Feldman), les explication de chacune,...
* De Pascal à ADA (p.795 de Feldman),

## Terminologie OO en ADA

En ADA : une "classe" = l'ensemble de la hiérarchie.
En ADA : les opérations "primitives" = Méthodes dans C++
EN ADA : l'appel d'une primitive implique le "matching" (lien) des paramètres, statiquement ou dynamiquement selon que le paramètre est d'un type spécifique ou "class-wide". En C++, les règles sont plus complexes et dépend si le paramètre est un pointeur ou non et si l'appel est préfixé par le nom de sa classe ou non.
En ADA, le pointeur de "class-wide" n'est pas obligatoire (moi ?).
En ADA, une opération avec des paramètres "class-wide" est toujours liée statiquement (??? Tjs? Ce doit être une erreur de frappe. Cela contredit ci-dessus) même si cet appel s'applique à tous les types de la classe (de la hiérarchie).
An ADA, "dispatching" correspond à l'appel d'une primitive avec lien dynamique. De fait, les appels avec types accès sont une forme de liens dynamiques.
En ADA, le type "Abstract" correspond à la "classe abstraite" des autres langages. Les sous programmes "Abstract" sont les "méthodes virtuelles pures" de C++ et "defered" de Eiffel.
En ADA, le type "parent" ou un type "dérivé" correspondent à la "super classe" et "sous classe" des autres langages. La notion de "Subtype" en ADA n'a pas de correspondant dans les autres langages qui n'ont pas de "range check". Le "Substype" d'ADA n'a pas de relation avec la notion de "sous classe" car un subtype n'a jamais plus de valeur que son type de base alors que le lien classe-sousclasse est l'inverse.
En ADA, les "Génériques" sont les "templates" mais an ADA, ils procèdent au "type checking" alors que les autres langages traitent les "templates" comme des "macros".
En ADA, contrairement à d'autres langages qui n'ont que les classes pour l'Encapsulation, les paquetages (sans lien particulier à l'extension de type et l'héritage et le type privé). L'effet "private" et "protected" de C++ se réalise avec les type privés et les paquetages "child". Les "childs" sont en quelque sorte des "friends".

# Exemple Complet (Avions Hot ADA)

En rouge : Vehicle, Aircraft et Ship seront Abstract.

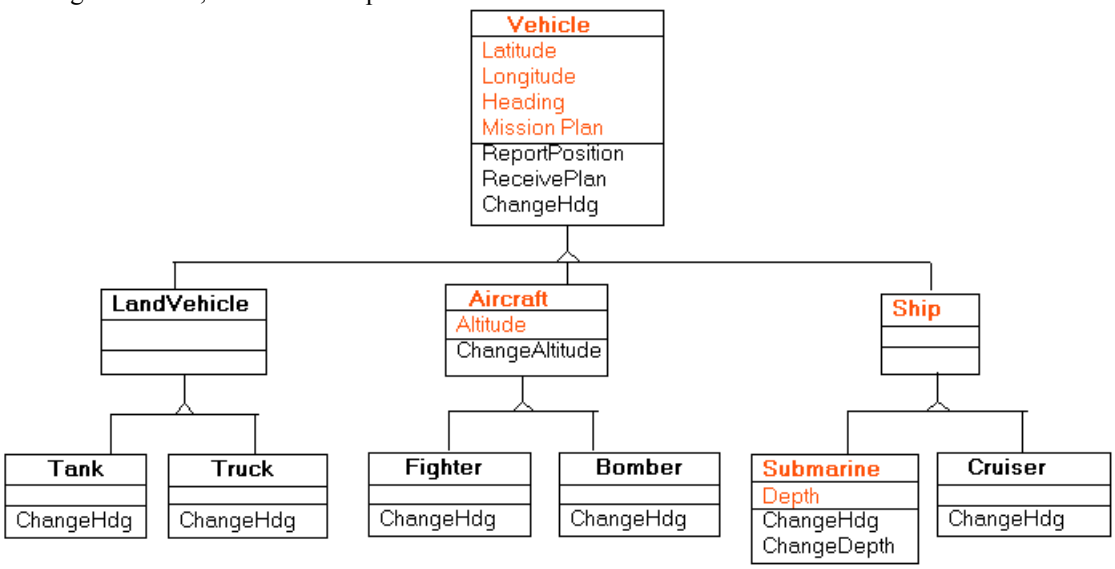
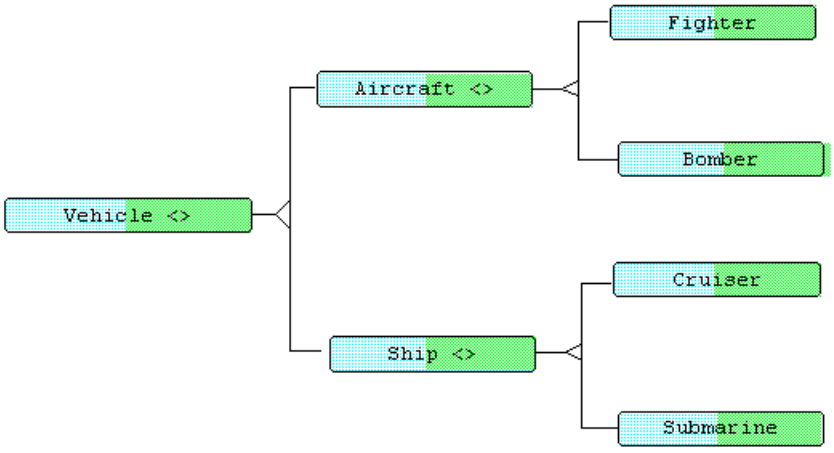
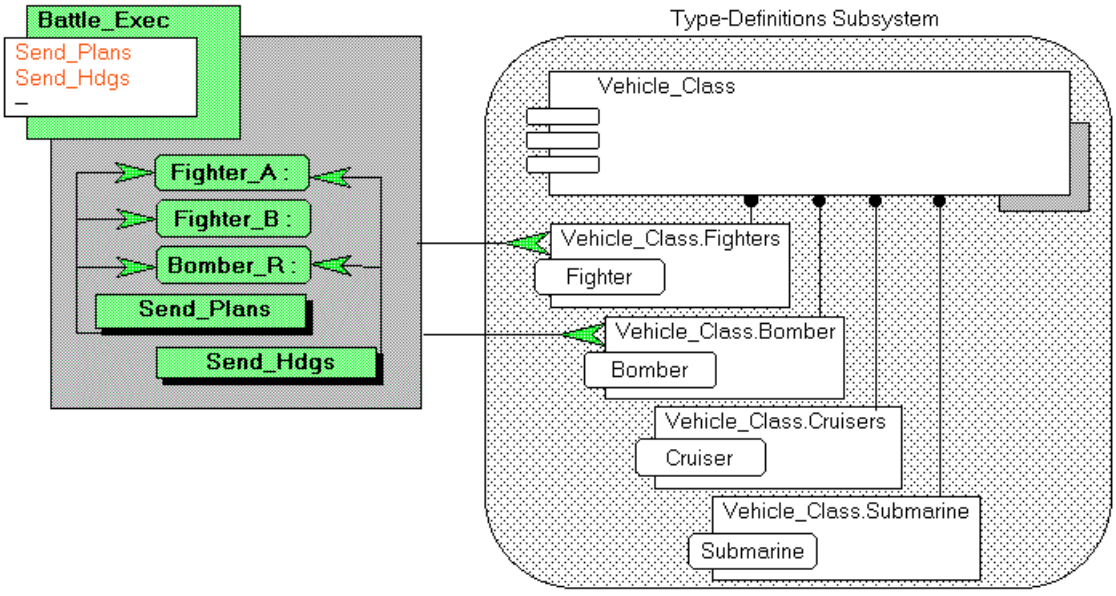



Schéma simplifié :




On voit que les classes "vehicle", "Aircraft" et "ship" n'auront pas directement des instances. De ce fait, elles sont déclarées ABSTRACT. Submarine sera abstrait seulement dans sa déclaration complète (interne au paquetage).

**Dispatching :**



## Le code ADA95 des spécifications

Graphe d'héritage ADA95. 7 morceaux de spécifications.

```
type Vehicle is Abstract tagged limited private;

procedure Report_Position (V : in Vehicle'Class; Lat : out Float; Long : out Float);
procedure Receive_Plan    (V : in out Vehicle'Class; MP : in Plan_Type);
procedure Change_Hdg      (V : in Vehicle; H : in Float) is Abstract;
procedure Change_Speed    (V : in Vehicle; S : in Float) is Abstract;

private
  type  Vehicle is Abstract tagged limited
    record
      Latitude : Float;
      Longitude : Float;
      Heading : Float;
      Speed : Float;
      Mission_Plan : Plan_Type;
     end record;
```

```
type Aircraft is Abstract new Vehicle with limited private;

procedure Change_Altitude (AC : in out Aircraft; A : in Float) is Abstract;

private
  type Aircraft is Abstract new Vehicle with limited
    record
      Altitude : Float;   -- new component
    end record;
```

```
type Ship is Abstract new Vehicle with limited private;

procedure Take_Sounding;

private
  type Ship is  Abstract new Vehicle with limited  null record;              -- no new components
```

```
type Fighter is new Aircraft with limited private;

procedure Change_Hdg   (F : in Fighter; H : in Float);
procedure Change_Speed (F : in Fighter; S : in Float);

private
  type Fighter  is new Aircraft with limited null record;                    -- no new components
```

```
type Bomber is new Aircraft with limited private;

procedure Change_Hdg   (B : in Bomber; H : in Float);
procedure Change_Speed (F : in Bomber; S : in Float);

private
  type Bomber  is new Aircraft with limited null record;             -- no new components
```

```
type Cruiser is new Ship with limited private;

procedure Change_Hdg   (C : in Cruiser; H : in Float);
procedure Change_Speed (C : in Cruiser; S : in Float);

private
  type Cruiser is Abstract new Ship with limited null record;            -- no new components
```

```
type Submarine is new Ship with limited private;

procedure Change_Hdg   (Sub : in Submarine; H : in Float);
procedure Change_Speed    S : in Float);
procedure Change_Depth (Sub : in Submarine; D : in Float);

private
  type Submarine is Abstract new Ship with limited
    record
      Depth : Float;      -- new component
    end record;
```

# Type access : pointeur de sous-programme

(p. 195 de Barnes).
* un type access peut être un discriminent de record (p. 361 Barnes)

**Exemple1** : pointeur sur procédure (sans paramètre) et son appel

```
Type pointeur_ss_prog is access procedure;        -- Déclaration
tab_point : array(1..N) of pointeur_ss_prog;      -- si on veut
...
    tab_point(i).all;                             -- l'appel
```

**Exemple2**: pointeur sur une fonction de certain profile et ses appels avec ou sans "all"

```
Type trig_fonc is access function(F:float) return Float;
T : trig_fonc;
X,Theta : Float;

Function Sin(X:Float) return Float;          -- la spécification d'une fonction
T := Sin'Access;
X:= T(Theta);
X:= T.all(Theta);        -- all est facultatif ici mais serait nécessaire si pas de paramètre.
```

**Exemple3**: pointeur sur une fonction de certain profile. Une fonction avec un "paramètre pointeur" sur fonction, les appels.

```
Type integrand is access function(F:float) return Float;
Function Integrate(F: Integrand, A,B : Float) return Float;        -- paramètre access fonction
...
Area := integrate(Log'Access, 1.0, 2.0);                  -- Fonction Log classqiue
```

**Exemple4**: cas de procédure avec paramètre. Exemple complet.
Gestion d'atelier. Créer des bouton dont un d'URGENCE. Associer la procédure d'Urgence à ce Bouton. Si l'événement arrive (Bouton Rouge d'Urgence poussé) alors exécuter.

```
Type bouton;
Type pt_reponse is access procedure(B : in out Bouton);
Type Bouton is record
    rep : pt_reponse;                -- un champ pointeur sur procédure
    ...
end record;
procedure associer(B : in out Bouton; ...);
procedure pousser(B : in out Bouton);
procedure set_reponse(B : in out Bouton; R : pt_reponse);

procedure pousser(B : in out Bouton) is
Begin
    B.rep(B);                        -- appel indirect
End pousser;
procedure set_reponse(B : in out Bouton; R : pt_reponse) is
Begin
    B.rep := R;
End set_reponse;

Bouton_rouge : Bouton;

procedure Urgence(B:in Bouton) is        -- On va utiliser cette procédure
Begin
    Broadcast("MayDay");
    Eject(Pilote);
end Urgence;
```

```
-- Exemple de fonctionnement
Associer(Bouton_rouge, ...);
set_reponse(Bouton_rouge, Urgence'Access);
Pousser(Bouton_rouge);
...
```

On se sert des pointeurs sur sous programmes pour l'interfacage avec d'autres langages.
**Remarque** : On ne peut pas manipuler l'adresse des sous programmes (dits intrinsèques) comme "+" du paquetage Standard, les énumérés (considérés comme fonction sans paramètre), la fonction implicite "/=" (implicitement déclarée avec "="), des attributs tels que Pred et Succ (qui sont en fait des fonctions).