Introduction à Ada 95

Programmation par Objets

- Approche "**Type Abstrait de Données**"
- Le système est décrit en terme d'"**objets**" représentant des entités réelles ou abstraites du domaine
- Chaque objet contient ses propres données et un ensemble d'opérations pour manipuler ces données
- La structure et le comportement d'un objet sont décrits par son **type**
- Ada supporte cette approche avec les notions de type et de paquetage

Extension de type

- Ada 95 permet de créer un nouveau type à partir d'un type donné en lui ajoutant des composants
- Ada 95 s'appuie pour cela sur le concept de type dérivé
- Pour pouvoir être étendus, les types doivent être des types articles déclarés étiquetés ("tagged")

Exemple

```
type Compte is tagged
   record
       titulaire:Unbounded_String;
       numero:String( 1..5 );
       solde:Float;
   end record;

type CompteRemunere is new Compte with
   record
       taux:Float;
   end record;
```

Extension de type

- Les opérations du type père sont héritées par les types fils dérivés.
- Elles sont appelées opérations primitives
- Ce sont les opérations déclarées dans le paquetage du type père ou bien qui ont le type père en paramètre ou en résultat.

Type article avec partie variante vs type étiqueté (1/2)

```
type Genre is( masculin,feminin );
type Personne( sexe:Genre ) is
  record
    naissance:Date;
    case sexe is
      when masculin => barbe:Boolean;
                       age  :Natural;
      when feminin  => enfants:Natural;
    end case;
  end record;
```

Type article avec partie variante vs type étiqueté (2/2)

```
type Genre is( masculin,feminin );
type Personne is tagged
  record
    naissance:Date;
   end record;
type Homme is new Personne with
   record
       barbe:Boolean;
       age  :Natural;
   end record;
type Femme is new Personne with
   record
       enfants:Natural;
   end record;
```

Types avec partie variante

```
pierre:Personne(masculin);
jeanne:Personne(feminin);

pierre:=(masculin,(12,2,19
   50),false,34);

pierre.barbe:=true;
```

Types étiquetés

```
x:Personne:=( 2,11,1962 );
pierre:Homme
   :=((12,2,1950),false,34);
jeanne:Femme;

-- conversion Homme->Personne
x:=Personne( pierre );
-- conversion Personne->Homme
pierre:=( x with true,age=>42 )
```

Types avec partie variante

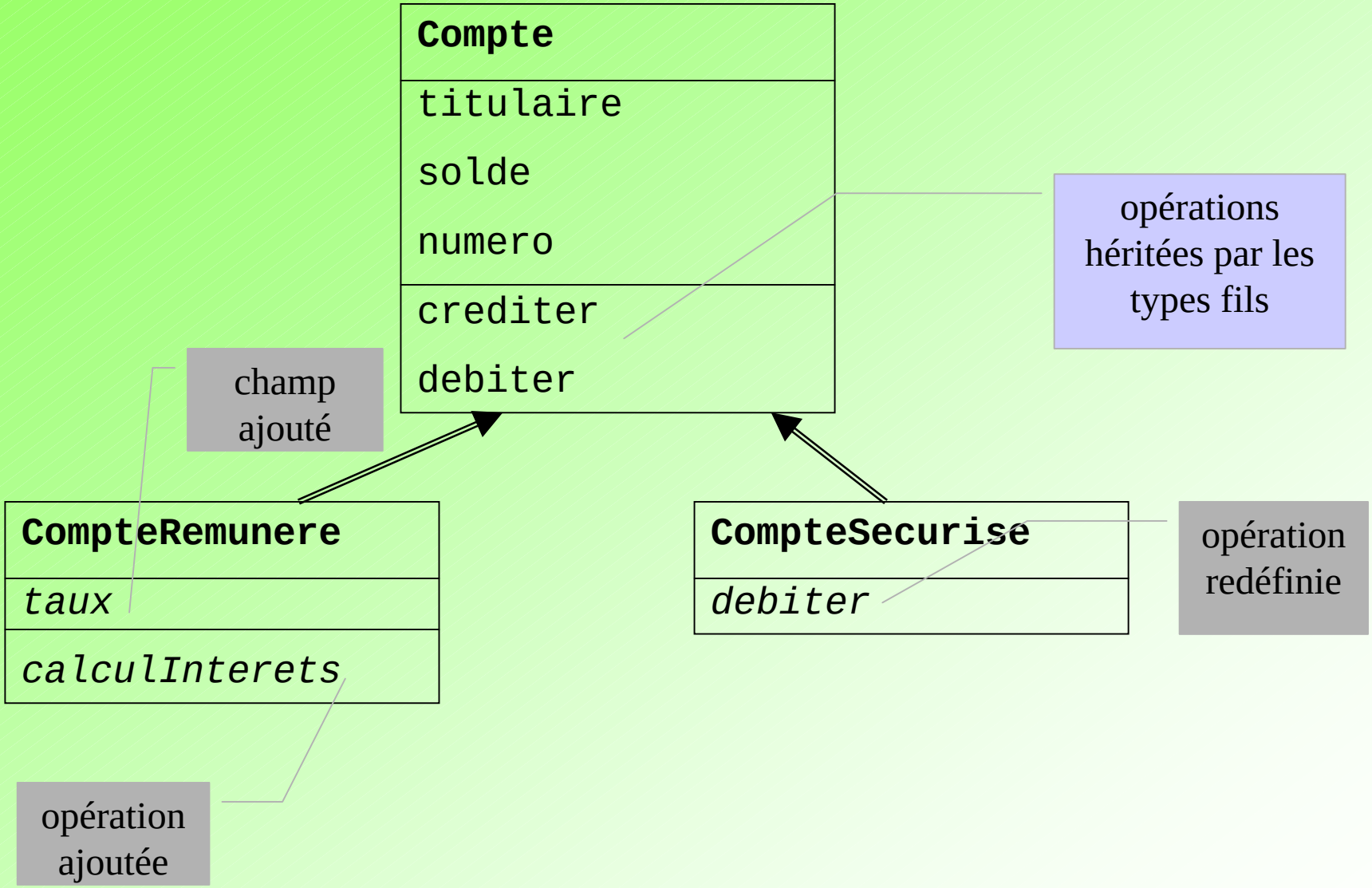
```
procedure put
          (p:in Personne)is
begin
 put("date de naissance:(");
 put( p.naissance.jour,2 );
 put( p.naissance.mois,2 );
 put_line(p.naissance.an,2);
 case p.sexe is
   when masculin =>
 put( "age=" );put(p.age,2);
 put( "barbe=" );
 put(Boolean'image(p.barbe));
   when feminin =>
put(p.age,2);put(" enfants");
 end case;
end put;
```

Types étiquetés

```
procedure put(p:in Personne)is
begin
 put( "date de naissance:(" );
 put( p.naissance.jour,2 );
 put( p.naissance.mois,2 );
 put_line( p.naissance.an,2 );
end put;
procedure put( p:in Homme )is
begin
 put( Personne(p) );
 put( "age=" );put( p.age,2 );
 put( "barbe=" );
 put( Boolean'image(p.barbe) );
end put;
```

Approche Orientée Objet

- Introduction du concept d'**héritage**
- De nouveaux types peuvent être introduits **par extension** de types existants
- L'ensemble des types forment une **hiérarchie**
- Tout nouveau type créé par héritage :
  - dispose des données et opérations du type père
  - peut définir de nouvelles opérations et/ou données
  - peut redéfinir les opérations héritées


Compte
titulaire
solde
numero
crediter
debiter
opérations héritées par les types fils
champ ajouté
CompteRemunere
taux
calculInterets
opération ajoutée
CompteSecurise
debiter
opération redéfinie

```
package Comptes is
--------- type Compte -------------------------
   type Compte is tagged
      record
         titulaire   : String( 1..10 );
         solde       : Float;
         numero      : String( 1..5 );
      end record;
   procedure debiter(  C: in out Compte;S: in Float );
   procedure crediter( C: in out Compte;S: in Float );
   procedure initialiser
        ( T: in String;S: in Float;N:in String;C: out Compte );
-------- type CompteRemunere -----------------
   type CompteRemunere is new Compte with
      record
         taux       :Float;
      end record;
   function calculinterets( C:CompteRemunere ) return Float;
   procedure initialiser
(T:in String;S:in Float;N:in String;taux:in Float;C:out CompteRemunere );
-------- type CompteSecurise -----------------
   soldeInsuffisant:exception;
   type CompteSecurise is new Compte with null record;
   procedure debiter( C: in out CompteSecurise;S: in Float );
end comptes;
```

```
package body comptes is
---------- type Compte --------------------
   procedure debiter( C: in out Compte;S: in Float ) is
   begin
      C.solde:= C.solde-S;
   end debiter;
   procedure crediter( C: in out Compte;S: in Float )is
   begin
      C.solde := C.solde+S;
   end crediter;
   procedure initialiser
       ( T:in String;S:in Float;N:in String;C:out Compte )is
   begin
      C := ( T,S,N );
   end initialiser;
```

```
---------- type CompteRemunere ----------------------------
   function calculinterets( C:CompteRemunere ) return Float is
   begin
      return C.solde*C.taux;
   end calculinterets;
   procedure initialiser
       ( T: in String;S: in Float;N: in String;
         taux: in Float;C: out CompteRemunere )is
   begin
      initialiser( T,S,N,C ); C.taux:=taux;
   end initialiser;
---------- type CompteSecurise ----------------------------
   procedure debiter( C:in out CompteSecurise;S:in Float )is
   begin
      if C.solde>=S
      then C.solde := C.solde-S;
      else raise soldeinsuffisant;
      end if;
   exception
      when soldeInsuffisant=>
               put_line( "Votre solde est insuffisant" );
   end debiter;
end comptes;
```

Hérite de la procédure initialiser de Compte

C est "une sorte de" Compte

```
with comptes;use comptes;

procedure clientcomptes is
   courant:Compte;
   remunere:CompteRemunere;
   securise:CompteSecurise;
begin
   initialiser( "Victor",200.0,"34567",courant );
   initialiser( "Pierre",0.0,"78231",0.037,remunere );
   initialiser( "Daniel",150.0,"77553",securise );

   debiter( courant,100.0 );
   crediter( remunere,650.0 );
   debiter( securise,200.0 );
   debiter( remunere, 300.0 );

end clientcomptes;
```

Héritage

- Ajoutons la déclaration de la procédure `put` au type `Compte` dans `comptes.ads`

```
procedure put(C:in Compte);
```

- Ajoutons la déclaration du corps de la procédure `put` au type `Compte` dans `comptes.adb`

```
procedure put(C:in Compte)is
begin
   put( "titulaire : " );put( C.titulaire );new_line;
   put( "solde     : " );put( C.solde,5,1,0 );new_line;
   put( "numero    : " );put( C.numero );new_line;
end put;
```

```
with comptes;use comptes;
with Ada.Text_Io;use Ada.Text_Io;
procedure clientcomptes is
   courant:Compte;
   remunere:CompteRemunere;
   securise:CompteSecurise;
begin
   initialiser( "Victor",200.0,"34567",courant );
   put(courant);new_line;
   initialiser( "Pierre",0.0,"78231",0.037,remunere );
   put( remunere );new_line;
   initialiser( "Daniel",150.0,"77553",securise );
   put( securise );new_line;
   put_line( "--------------------" );
   debiter( courant,100.0 );
   put(courant);new_line;
   crediter( remunere,650.0 );
   put( remunere );new_line;
   debiter( securise,200.0 );
   put( securise );new_line;
end clientcomptes;
```

la procédure `put` de `Compte` est exécutée car `remunere` et `securise` sont d'un type dérivé du type étiqueté père `Compte`

Polymorphisme (1/2)

- A chaque type étiqueté `T` correspond un type Classe `T'Class`
- Un type `T'Class` est l'union de tous les types dérivés à partir de `T` (y compris `T`)
- Par exemple, `Compte'Class` inclut les types `Compte`, `CompteRemunere` et `CompteSecurise`
- Le type `Compte'Class` remplace donc tous les types inclus
- Par exemple, une procédure `print` dans `clientComptes` est un objet polymorphe (le type réel du paramètre formel est inconnu).

```
procedure print( C: in Compte'Class ) is
begin put( C );new_line; end print;
```

`print` pourrait s'appliquer aux paramètres inclus dans `Compte'Class`

```
print( courant );
print( securise );
print( remunere );
```

Polymorphisme (2/2)

- Le choix de l'opération `print` est réalisé dynamiquement
- Cela est rendu possible car chaque type étiqueté inclus un "tag" invisible qui renseigne Ada sur le type effectif du paramètre
- On appelle **liaison dynamique** le mécanisme mis en oeuvre pour utiliser un sous-programme en l'adaptant aux paramètres effectifs

Polymorphisme : exemple 1 (1/2)

```
with comptes,Ada.Text_Io;use comptes,Ada.Text_Io;
procedure clientcomptes_acces is
   procedure print( C: in Compte'class ) is
   begin put( C );new_line; end print;
   courant  : Compte;
   remunere: CompteRemunere; securise: Comptesecurise;
   type Tabcomptes is array( 1..3 ) of Compte'class;
begin
    initialiser( "Victor",200.0,"34567",0.045,courant );
    initialiser( "Pierre",0.0,"78247",0.037,remunere );
    initialiser( "Daniel",150.0,"77553",securise );
    declare
      t:Tabcomptes:=( courant,remunere,securise );
    begin
      for i in t'range loop print( t(i) ); end loop;
   end;
end clientcomptes_acces;
```

Type non contraint=> erreur à la compilation

Polymorphisme : exemple 1 (2/2)

- Dans l'exemple précédent, on constate que les types à échelle de classe ne sont intéressants qu'avec le typage des paramètres formels
- Comment créer un tableau dont les éléments appartiennent à la même hiérarchie de types ?
- La solution précédente provoque une erreur à la compilation : le type des éléments du tableau est non contraint puisqu'ils peuvent être de type différent (`Compte`, `CompteSecurise`, ...).

Polymorphisme : exemple 2 (1/3)

- En Ada 83, les objets créés dynamiquement ne pouvaient être référencés qu'à travers des variables de type `access`

- Ada 95 nous donne la possibilité de manipuler une variable à travers son pointeur à condition de déclarer un type accès général

```
type Ptr_Compte is access all Compte'Class;
p: Ptr_Compte;
```

- Pour pouvoir affecter l'adresse d'une variable du type `Compte'Class`, il suffit de la déclarer comme "`aliased`"

```
remunere: aliased CompteRemunere;
```

- Pour affecter une valeur à un pointeur, il n'est plus nécessaire de construire un objet dynamiquement (opérateur **`new`**), il est possible d'accéder à l'adresse d'un objet statique par l'attribut **`Access`**

```
p:=remunere'Access;
```

Polymorphisme : exemple 2 (2/3)

```
with comptes, Ada.Text_Io;use comptes, Ada.Text_Io;
procedure clientcomptes_acces is
   procedure print(C:in Compte'class) is
   begin put(C);new_line; end print;
   courant  : aliased Compte;
   remunere: aliased CompteRemunere;
   securise: aliased CompteSecurise;
   type Ptr_Compte is access all Compte'Class;
   type Tabcomptes is array(1..3) of Ptr_Compte;
   t:Tabcomptes;
begin
   initialiser( "Victor",200.0,"34567",0.045,courant );
   initialiser( "Pierre",0.0,"78231",0.037,remunere );
   initialiser( "Daniel",150.0,"77553",securise );
   t(1):=courant'access;t(2):=remunere'access;t(3):=securise'access;
   for i in t'range loop
     print( t(i).all );
   end loop;
end clientcomptes_acces;
```

la taille des composants est connue

Polymorphisme : exemple 2 (3/3)

- Contrairement à l'exemple 1, le compilateur dispose de toutes les informations pour allouer l'espace mémoire aux composants du tableau.
- Le tableau `t` contient donc des références sur des objets de type différent.
- Le type réel de l'objet n'est reconnu qu'au moment de l'exécution

Abstraction de données

- Mécanisme pour cacher à l'utilisateur la représentation d'un type donné
- Pour écrire un programme utilisant un tel type, l'utilisateur ne peut plus se fonder que sur l'interface du type
- Ce concept facilite la maintenance et la fiabilité du logiciel
- Le type est alors déclaré :

  ```
  tagged private
  ```

  ou bien

  ```
  new Père with private
  ```

- Exemple

  ```
  type Compte is tagged private;
  type CompteRemunere is new Compte with private;
  ```

```
package Comptes is
   type Compte is tagged private;
   procedure debiter( C:in out Compte;S:in Float );
   procedure crediter( C:in out Compte;S:in Float );

   type CompteRemunere is new Compte with private;
   function calculinterets( C:CompteRemunere ) return Float;

   type Comptesecurise is new Compte with private;
   procedure debiter( C:in out CompteSecurise;S:in Float );

private
   type Compte is tagged
       record
         titulaire   :String( 1..6 );
         solde       :Float;
         numero      :String( 1..5 );
      end record;
   type Compteremunere is new Compte with
      record
         taux        :Float;
      end record;
   type CompteSecurise is new Compte with null record;
end comptes;
```

Types abstraits

- Un type est abstrait lorsque, bien qu'il ne représente aucun objet réel, il regroupe l'ensemble des caractéristiques (données+opérations) communes à tous les types dérivés présents et à venir

- Un type abstrait peut posséder des sous-programmes abstraits

- On garantit ainsi que tout nouveau type dérivé ajouté à la hiérarchie implantera les sous-programmes du type abstrait. Le compilateur sera en mesure de signaler cet oubli.

- On favorise ainsi le polymorphisme

```
package Comptes is
--- Compte -------------------
   type Compte is abstract tagged private;
   procedure debiter( C:in out Compte;S:in Float ) is abstract;
   procedure crediter( C:in out Compte;S:in Float );
   procedure put( C:in Compte );
--- CompteRemunere -----------
   type CompteRemunere is new Compte with private;
   function calculInterets( C:CompteRemunere) return Float;
   procedure initialiser( T:in String;S:in Float;N:in String;
                          taux:in Float;C:out CompteRemunere );
   procedure debiter( C:in out CompteRemunere;S:in Float );
--- CompteSecuriseRemunere ---
   soldeinsuffisant:exception;
   type CompteSecuriseRemunere is new CompteRemunere with private;
   procedure debiter( C:in out CompteSecuriseRemunere;S:in Float )
--- CompteSecurise -----------
   type CompteSecurise is new Compte with private;
   procedure initialiser( t:in String;s:in Float;n:in String;
                          C:out CompteSecurise );
   procedure debiter( C:in out CompteSecurise;S:in Float );
```

```
private
   type Compte is abstract tagged
       record
         titulaire   :String(1..6);
         solde       :Float;
         numero      :String(1..5);
      end record;
   type CompteRemunere is new Compte with
      record
         taux       :Float;
      end record;
   type CompteSecurise is new Compte with null record;
   type CompteSecuriseRemunere is new CompteRemunere
                                        with null record;

end comptes;
```

```
with Ada.Text_Io;use Ada.Text_Io;
with Ada.Float_Text_io;use Ada.Float_Text_io;

package body Comptes is
   ------------------
   -- type Compte --
   ------------------
   procedure crediter( C: in out Compte;S: in Float )is
   begin
      C.solde := C.solde+S;
   end crediter;
   procedure put( C:in Compte )is
   begin
      put( "titulaire : " );put( C.titulaire );new_line;
      put( "solde     : " );put( C.solde,5,1,0 );new_line;
      put( "numero    : " );put( C.numero );new_line;
   end put;
```

```
--------type CompteRemunere -------------------------
function calculinterets( C:CompteRemunere ) return Float is
begin
   return C.solde*C.taux;
end calculinterets;
procedure initialiser( T:in String;S:in Float;N:in String;
                       taux:in Float;C:out CompteRemunere )is
begin
   C:=( T,S,N,taux );
end initialiser;
procedure debiter( C:in out CompteRemunere;S:in Float ) is
begin
   C.solde:=C.solde-S;
end debiter;
--------type CompteSecuriseRemunere --------------------
procedure debiter(C:in out CompteSecuriseRemunere;S:in Float)is
begin
   if C.Solde>=S
   then C.Solde := C.Solde-S;
   else raise Soldeinsuffisant;
   end if;
exception
  when soldeInsuffisant=>put_line("Votre solde est insuffisant");
end debiter;
```

```
   --------- type CompteSecurise -------------------------
   procedure initialiser( T:in String;S:in Float;N:in String;
                          C:out CompteSecurise )is
   begin
      C:=( T,S,N );
   end initialiser;

   procedure debiter( C:in out CompteSecurise;S:in Float )is
   begin
      if C.Solde>=S
      then C.Solde := C.Solde-S;
      else raise soldeInsuffisant;
      end if;
   exception
      when soldeInsuffisant=>
                    put_line( "Votre solde est insuffisant" );
   end debiter;
end comptes;
```

Remarques

- Les opérations abstraites sont nécessairement déclarées et définies dans **tous** les types fils

- On ne peut pas déclarer des objets d'un type abstrait

`C:Compte;` -- illégal

```
with Comptes;use Comptes;
with Ada.Text_Io;use Ada.Text_Io;
procedure clientcomptes is
   procedure print(C:in Compte'Class) is
   begin
      put(C);new_line;
   end print;
   remSec:CompteSecuriseRemunere;
   remunere:CompteRemunere;
   Securise:CompteSecurise;
begin
   initialiser( "Victor",200.0,"34567",0.045,remSec );
   put( remSec );new_Line;
   initialiser( "Pierre",0.0,"78231",0.037,remunere );
   put( remunere );new_Line;
   initialiser( "Daniel",150.0,"77553",securise );
   put( securise );new_Line;
   debiter(remSec,100.0); put( remSec );new_Line;
   crediter( remunere,650.0 ); put( remunere );new_line;
   debiter( securise,200.0 ); put( securise );new_Line;
   print( remSec );
end clientcomptes;
```

Retour sur String (1/3)

- Comment se libérer de la contrainte sur la taille des chaînes de caractères

Retour sur String (2/3)

```
with Ada.Strings.Bounded;
package Comptes is
   package Chaines is
      new Ada.Strings.Bounded.Generic_Bounded_Length( 15 );
   use chaines;
……
   procedure initialiser(t:in BOUNDED_STRING;s:in Float;
                         n:in BOUNDED_STRING;taux:in Float;
                         c:out Compteremunere);
   procedure initialiser(t:in BOUNDED_STRING;s:in Float;
                         n:in BOUNDED_STRING;
                         c:out CompteSecurise);
   type Compte is tagged
       record
         titulaire   :BOUNDED_STRING;
         solde       :Float;
         numero      :BOUNDED_STRING;
      end record;
```

Retour sur String (3/3)

```
procedure Clientcomptes_Acces is
   initialiser( To_Bounded_String("Victorine"),200.0,
                To_Bounded_String("3456725"),0.045,remsec );
   initialiser( To_Bounded_String("Pierre"),0.0,
                To_Bounded_String("782"),0.037,remunere );
   initialiser( To_Bounded_String("Dan"),150.0,
                To_Bounded_String("77553"),securise );
```

Etude de cas

On veut manipuler dans un même programme les objets suivants :

- des images caractérisées par leur couleur et leur position (respectivement du type `Couleur` et `Position`).

Parmi ces images, on trouve:

- des photos, caractérisées par leur taille et que l'on pourra compresser,
- des carrés et des hexagones définis par le nombre et la longueur de leurs cotés
- et plus généralement des polygones réguliers définis par le nombre de cotés et la longueur de leurs cotés et dont on pourra calculer la superficie et le périmètre

On veut pouvoir afficher à l'écran chacun de ces objets et les déplacer

Proposer l'architecture qui vous paraît la plus adaptée. Elle sera exprimée en UML.

On ne s'intéressera donc pas au codage des méthodes mais seulement à leur déclaration.

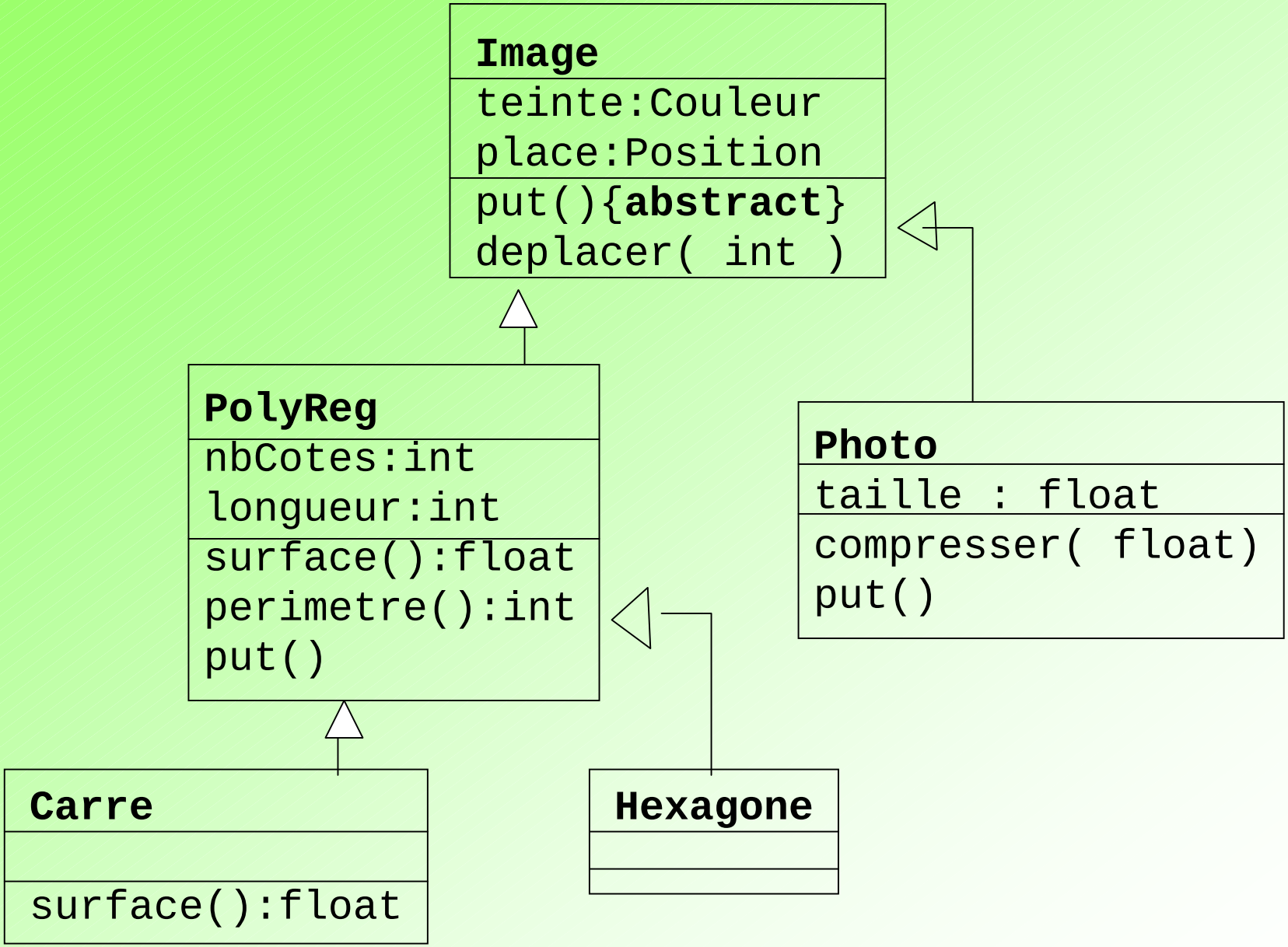


Image
teinte:Couleur
place:Position
put(){abstract}
deplacer(int)
PolyReg
nbCotes:int
longueur:int
surface():float
perimetre():int
put()
Photo
taille : float
compresser(float)
put()
Carre
surface():float
Hexagone

le type Image

```
package images is
   type Couleur is ( rouge,jaune,bleu,blanc,vert,orange,violet);
   type Position is
      record
         x : Natural;
         y : Natural;
      end record;

   type Image is abstract tagged
      record
         teinte : Couleur;
         place  : Position;
      end record;

   procedure put( p: in Image ) is abstract ;
   procedure deplacer(p: in out Image;t: in Natural);
end images;
```

Le type PolyReg

```
with images;use images;

package polyregs is

   type PolyReg is new Image with
      record
         nbCotes   : Positive;
         longueur  : Positive;
      end record;

   function surface( f: PolyReg ) return Float;
   function perimetre( f: PolyReg ) return Natural;
   procedure put( p: in PolyReg );
end polyregs;
```

Le type `Carre`

```
with polyregs; use polyregs;

package carres is

    type Carre is new PolyReg with NULL record;

      function surface( c: Carre ) return Float;

end carres;
```

Le type Hexagone

```
with polyregs;use polyregs;

package hexagones is
   type Hexagone is new PolyReg with
      null record;
end hexagones;
```

Le type Photo

```
with images;use images;

package photos is

   type Photo is new Image with
      record
         taille : Float;
      end record;

   procedure compresser( p: in out Photo;coef: in Float );
   procedure put( p: in Photo );
end photos;
```

Corps du package images

```
with Ada.Text_io;use  Ada.Text_io;
with Ada.Integer_Text_io;use  Ada.Integer_Text_io;

package body images is

   procedure deplacer( p: in out Image;t: in Natural is
   begin
      put("nouvelle place de la figure: ");
      put("(");put(p.place.x+t,3);put(",");
      put(p.place.y+t,3);put(")");
   end deplacer;
end images;
```

Corps du package polyregs(1/2)

```
with Ada.Numerics.Generic_Elementary_Functions;
with images;use images;

package body polyregs is
 package tangente is
     new Ada.Numerics.Generic_Elementary_Functions( Float );
 use tangente;

 function surface( f: PolyReg ) return Float is
 begin
   return Float(f.nbCotes*f.longueur*f.longueur)/
                4.0*tan( 3.14/Float(f.nbCotes) );
 end surface;

 function perimetre( f: PolyReg ) return Natural is
 begin
   return f.nbCotes*f.longueur;
 end perimetre;
end polyregs;
```

Corps du package `polyregs` (2/2)

```
procedure put( p: in PolyReg ) is
begin
      put( "couleur = " );
      put( Couleur'image( p.teinte ) );
      put( ", place = " );put("(");
      put( p.place.x,3 );put( "," );
      put( p.place.y,3 );put(")");
      put( ", nbCotes = " );put( p.nbCotes,2 );
      put( "," );
      put( ", longueur= " );put( p.longueur,3 );
      new_line;
end put;
```

Corps du package carres

```
package body carres is

   function surface( c: Carre ) return Float is
   begin
        return Float(c.longueur*c.longueur);
   end surface;

end carres;
```

Corps du package photos

```
with Ada.Text_io;use  Ada.Text_io;
with Ada.Float_Text_io;use  Ada.Float_Text_io;
with Ada.Integer_Text_io;use  Ada.Integer_Text_io;

package body  photos is
  procedure compresser( p: in out Photo; coef: in Float ) is
  begin
    put( "compresser : nouvelle taille de la photo= " );
    put( p.taille*coef, 3,2,2 );
  end compresser;

  procedure put( p:in Photo ) is
  begin
    put( "couleur = " );put( Couleur'image( p.teinte ) );
    put( ", place = " );put("(");put( p.place.x,3 );put( "," );
    put( p.place.y,3 );put(")");
    put( ", taille=" );put( p.taille,2,2,2 );new_line;
  end put;
end photos;
```

```
with images;use images;
with carres;use carres;
with hexagones;use hexagones;
with photos;use photos;
with Ada.Text_io;use Ada.Text_io;
with Ada.Float_Text_io;use Ada.Float_Text_io;

procedure client_poo is
   c:Carre:=( rouge,(25,50),4,15 );
   h:Hexagone:=( bleu,(10,80),6,10 );
   p:Photo:=( blanc,(23,46),60.0 );
   s:Float;
   procedure afficher( f: in Image'class ) is
   begin
      put ( f );
   end afficher;
begin
   afficher( c );s:=surface( c );put( s,2,2,2 );new_line;
   afficher( h );s:=surface( h );put( s,2,2,2 );new_line;
   afficher( p );deplacer( p,10 );afficher( p );new_line;
   put( c );put( h );put( p );new_line;
end client_poo;
```

Ajout d'un nouveau type

On souhaite ajouter le type `Cercle` est défini par son rayon, sa couleur et sa position. On doit pouvoir les déplacer et les afficher.

On décide donc de le faire hériter du type `Image`.

La méthode `put` de `Image` étant abstraite, il faut la définir dans le type `Cercle` pour que le compilateur ne provoque pas d'erreur

Cercle
rayon:float
put()

Le package `cercles` (1/2)

```
with images;use images;

package cercles is

   type Cercle is new Image with
      record
         rayon:Float;
      end record;

   procedure put( p: in Cercle );

end cercles;
```

Le package `cercles` (2/2)

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Float_Text_io;use Ada.Float_Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;

package body cercles is
  procedure put( p: in Cercle) is
  begin
    put( "couleur = " );put( Couleur'image(p.teinte) );
    put( ", place = " );put("(");put( p.place.x,3 );
    put( "," );put( p.place.y,3 );put(")");
    put( ", rayon = " );put( p.rayon );new_line;
  end put;
end cercles;
```