Pratique d'un langage de programmation orienté objet.

1) Introduction :

Les exemples suivants reprennent la syntaxe du langage Ada 2005.
Une des raisons de ce choix est sans doute que l'on peut trouver aisément ailleurs des exemples pour d'autres langages ;-).
Les structures Ada sont basées sur les excellents articles "Apport d'Ada 95 aux paradigmes orientés objet", "A Naming Convention for Classes in Ada 9X" de J.P. Rosen (www.adalog.fr) et "Rationale for Ada 2005" de J. Barnes.

2) Encapsulation :

Propriété à la base de la programmation orientée objet (POO), l'encapsulation passe en premier lieu par une bonne réflexion sur le monde qui nous entoure. Laissons le clavier de côté. Notamment, réfléchissons sur le problème qui nous préoccupe pour obtenir la meilleure abstraction convenable.

Nous allons donc créer des composants logiciels objets permettant l'affichage de formes géométriques, exemple trivial à souhait mais néanmoins instructif.
Dans cette approche, l'objet de base sera une figure définie par ces coordonnées d'où seront dérivés d'autres objets "concrets" comme un point, un cercle ou un rectangle :

```
package Figure is
type Coordonnee is range -100 .. 100;
subtype Surface is Float;
type Instance is tagged record
-- remarquez le mot clé "tagged" qui déclare l'objet
        X, Y : Coordonnee := 0;
        end record;
subtype Class is Instance'Class;
-- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut l'instance et
tous ces descendants
end Figure;
```

En Ada, une "classe" au sens objet se défini avec une unité package avec la définition du type objet (tagged) et de ses opérations associées ou "méthodes" en langage objet.

Nous pouvons pousser plus loin l'encapsulation en rendant invisible la structure interne de l'instance pour en faire un type de données abstrait :

```
package Figure is
type Coordonnee is range -100 .. 100;
subtype Surface is Float;
type Instance is tagged private;
-- remarquez le mot clé "private" qui rend la structure interne de l'objet invisible
subtype Class is Instance'Class;
-- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut l'instance et
tous ces descendants
private
type Instance is tagged record
-- remarquez le mot clé "tagged" qui déclare l'objet
        X, Y : Coordonnee := 0;
        end record;
end Figure;
```

On va associer cet objet à une première procédure qui permet de donner une valeur aux coordonnées et deux fonctions pour retrouver ces valeurs :

```
package Figure is
type Coordonnee is range -100 .. 100;
subtype Surface is Float;
type Instance is tagged private;
-- remarquez le mot clé "private" qui rend la structure interne de l'objet invisible
subtype Class is Instance'Class;
-- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut l'instance et
tous ces descendants
procedure Positionne (Objet : in out Instance; X, Y : Coordonnee);
function RetourneX (Objet : in Instance) return Coordonnee;
function RetourneY (Objet : in Instance) return Coordonnee;
private
type Instance is tagged record
-- remarquez le mot clé "tagged" qui déclare l'objet
        X, Y : Coordonnee := 0;
        end record;
end Figure;
```

```
package body Figure is
procedure Positionne (Objet : in out Instance; X, Y : Coordonnee) is
  begin
  Objet.X := X;
  Objet.Y := Y;
  end;
function RetourneX (Objet : in Instance) return Coordonnee is
  begin
  return Objet.X;
  end;
function RetourneY (Objet : in Instance) return Coordonnee is
  begin
  return Objet.Y;
  end;
end Figure;
```

La procédure Positionne est un "constructeur" et les fonctions RetourneX, RetourneY sont deux "sélecteurs" de notre objet suivant les dénominations du langage objet.

L'utilisation de l'objet est une simple déclaration :

```
with Figure;
…
MaFigure : Figure.Instance; -- instanciation de l'objet
...
begin
...
Figure.Positionne (MaFigure, 10, 20);
…
end;
```

Avec Ada 2005, nous pouvons utiliser la notation préfixée dans le style de Java :

```
with Figure;
…
MaFigure : Figure.Instance; -- instanciation de l'objet
...
begin
...
MaFigure.Positionne (10, 20); -- la référence à l'unité Figure est donnée par MaFigure
…
end;
```

3) Héritage simple :

Accélérons et déclinons notre objet de base pour en faire un point :

```
with Figure;
package Point is
type Instance is new Figure.Instance with private;
-- noter le mot clé "new" permettant l'héritage
-- et le mot clé private pour l'encapsulation
subtype Class is Instance'Class;
-- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut l'instance et
tous ces descendants
private
type Instance is new Figure.Instance with null record;
-- et le mot clé "with" pour ajouter des champs, ici on n'ajoute rien pour faire un point
end Point;
```

L'objet hérite du constructeur et des sélecteurs de Figure :

```
with Point;
…
MonPoint : Point.Instance; -- instanciation de l'objet
...
begin
...
Point.Positionne (MonPoint, 15, 25);
-- ou Ada 2005 : MonPoint.Positionne (15, 25);
...
end;
```

4) Surcharge :

Déclinons à nouveau notre objet de base pour en faire un cercle de rayon R :

```
with Figure;
package Cercle is
type Instance is new Figure.Instance with private;
-- noter le mot clé "new" permettant l'héritage
-- et le mot clé private pour l'encapsulation
subtype Class is Instance'Class;
-- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut l'instance et
tous ces descendants
procedure Positionne (Objet : in out Instance; X, Y, R : Figure.Coordonnee);
function RetourneR (Objet : in Instance) return Figure.Coordonnee;
function Aire (Objet : in Instance) return Figure.Surface;
private
type Instance is new Figure.Instance with record
-- noter le mot clé "new" permettant l'héritage
-- et le mot clé "with" pour ajouter des champs
        R : Figure.Coordonnee := 0;
        -- l'initialisation par défaut à zéro ce qui revient à un Point ;-)
        end record;
end Cercle;
```

Un constructeur surchargé de l'ancêtre et un nouveau sélecteur sont définis pour l'objet. Nous en profitons pour ajouter une méthode calculant l'aire du cercle. Les autres méthodes sont héritées de la même façon que pour le point :

```
package body Cercle is
procedure Positionne (Objet : in out Instance; X, Y, R : Figure.Coordonnee) is
  begin
  Positionne (Objet, X, Y);
  -- ou Ada 2005 : Objet.Positionne (X, Y);
  -- on utilise le constructeur hérité de Figure
  Objet.R := R;
  end;
function RetourneR (Objet : in Instance) return Figure.Coordonnee is
  begin
  return Objet.R;
  end;
function Aire (Objet : in Instance) return Figure.Surface is
  begin
  return 3.14 * Figure.Surface(Objet.R) * Figure.Surface(Objet.R);
  end;
end Cercle;
```

Le constructeur de l'objet figure est surchargé pour une utilisation avec notre cercle :

```
with Cercle;
…
MonCercle : Cercle.Instance; -- instanciation de l'objet
...
begin
...
Cercle.Positionne (MonCercle, 10, 10, 30);
-- ou Ada 2005 : MonCercle.Positionne (10, 10, 30);
Put_Line("Aire cercle : " & Cercle.Aire (MonCercle)'Img);
-- ou Ada 2005 : Put_Line("Aire cercle : " & MonCercle.Aire'Img);
...
end;
```

L'appel du constructeur du cercle est déterminé par l'objet de type Cercle. Il appelle lui-même le constructeur de type Figure. L'appel est déterminé ici par la différentiation du nombre de paramètres. Si ce n'est pas le cas, l'appel se fait avec une conversion de type vers l'objet ancêtre :

```
Figure.Positionne (Figure.Instance(Objet), X, Y);
-- on force l'utilisation du constructeur hérité de Figure
```

Avec Ada 2005, l'appel à l'objet ancêtre est plus simple voire transparent :

```
MonCercle.Positionne (4005, 5005);
```

Contrairement à d'autres langages qui ne peuvent appeler que le père, Ada permet d'appeler n'importe quel ancêtre.

Ada 2005 fourni un indicateur "not overriding" pour assurer qu'une méthode est une surcharge et non pas une superposition :

```
not overriding
procedure Positionne (Objet : in out Instance; X, Y, R : Figure.Coordonnee);
```

5) Superposition :

Déclinons notre objet Cercle pour en faire un carré de côté R :

```
package Carre is
type Instance is new Cercle.Instance with private;
-- noter le mot clé "new" permettant l'héritage
-- et le mot clé private pour l'encapsulation
subtype Class is Instance'Class;
-- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut l'instance et
tous ces descendants
function Aire (Objet : in Instance) return Figure.Surface;
private
type Instance is new Cercle.Instance with null record;
-- et le mot clé "with" pour ajouter des champs, ici on n'ajoute rien pour faire un carré à
partir du cercle
end Carre;
```

Le constructeur ainsi que les sélecteurs sont hérités de l'ancêtre Cercle. Par contre le calcul de surface n'est plus valable, nous superposons alors le calcul de surface d'un carré par une méthode de même nom et de même paramètre :

```
package body Carre is
function Aire (Objet : in Instance) return Figure.Surface is
  begin
  return Figure.Surface(RetourneR(Objet)) * Figure.Surface(RetourneR(Objet));
  -- ou Ada 2005 :
  --   return Figure.Surface(Objet.RetourneR) * Figure.Surface(Objet.RetourneR);
  end;
end Carre;
```

Remarquer que nous ne pouvons pas utiliser le rayon avec Objet.R car le champ R de Cercle est caché. Nous utilisons alors le sélecteur adéquat. Vous remarquerez également qu'avec Ada 2005 la syntaxe est quasi identique Objet.RetourneR.

La méthode de l'objet Cercle est remplacée pour une utilisation avec notre carré :

```
with Carre;
…
MonCarre : Carre.Instance; -- instanciation de l'objet
...
begin
...
Carre.Positionne (MonCarre, 15, 20, 40);
-- ou Ada 2005 : MonCarre.Positionne (15, 20, 40);
Put_Line("Aire carré : " & Carre.Aire (MonCarre)'Img);
-- ou Ada 2005 : Put_Line("Aire carré : " & MonCarre.Aire'Img);
...
end;
```

L'appel de la méthode du carré est déterminé par l'objet de type Carre. Là aussi, l'appel vers la méthode de l'ancêtre se fait avec une conversion de type vers l'objet ancêtre :

```
Put_Line("Aire cercle avec carré : " & Cercle.Aire (Cercle.Instance(MonCarre))'Img);
-- on force l'utilisation du constructeur hérité de Cercle
```

Évidemment, le résultat du calcul est incorrect ;-)
Avec Ada 2005, l'appel à l'objet ancêtre est plus simple voire transparent :

```
Put_Line("Aire cercle avec carré : " & Cercle.Instance(MonCarre).Aire'Img);
```

Ici aussi, contrairement à d'autres langages qui ne peuvent appeler que le père, Ada permet d'appeler n'importe quel ancêtre.

Ada 2005 fourni un indicateur "overriding" pour assurer qu'une méthode est une superposition et non pas une surcharge :

```
overriding
function Aire (Objet : in Instance) return Figure.Surface;
```

6) Généricité :

Nous allons créer une méthode d'affichage pour l'objet Point en utilisant les procédures génériques Ada :

```
generic
with procedure Allume (X, Y : Figure.Coordonnee);
with procedure Eteins (X, Y : Figure.Coordonnee);
procedure Affiche (Objet : Instance; EstVisible : Boolean);
```

et son code :

```
procedure Affiche (Objet : Instance; EstVisible : Boolean) is
  begin
  if EstVisible then
    Allume(RetourneX(Objet), RetourneY(Objet));
    -- ou Ada 2005 : Allume(Objet.RetourneX, Objet.RetourneY);
  else
    Eteins(RetourneX(Objet), RetourneY(Objet));
    -- ou Ada 2005 : Eteins(Objet.RetourneX, Objet.RetourneY);
  end if;
  end;
```

Cette nouvelle méthode utilise les procédures formelles "Allume" et "Eteins", que l'on pourra coder en fonction de notre système favori.
On obtient alors une procédure générique puisqu'elle ne dépend pas du système utilisé.

```
with Point;
…
MonPoint : Point.Instance; -- instanciation de l'objet
procedure Allume (X, Y : Figure.Coordonnee);
procedure Eteins (X, Y : Figure.Coordonnee);
procedure MonAffiche is new Point.Affiche(Allume, Eteins);
...
begin
...
Point.Positionne (MonPoint, 15, 25);
...
MonAffiche (MonPoint, False);
 -- ou Ada 2005 : pas de notation préfixée, la méthode n'est pas déclarée avec l'objet
...
end;
```

Cette méthode ne fonctionnera pas pour notre cercle, nous devrons la surcharger par une méthode spécifique :

```
with Figure;
generic
with procedure Allume (X, Y : Figure.Coordonnee);
with procedure Eteins (X, Y : Figure.Coordonnee);
procedure Affiche (Objet : Instance; EstVisible : Boolean);

procedure Affiche (Objet : Instance; EstVisible : Boolean) is
  PointCourant : Point;
  begin
  Positionne (PointCourant, Objet.X + Objet.R, Objet.Y);
  while CercleNonFini loop
  if EstVisible then
    Allume(RetourneX(Objet), RetourneY(Objet));
    -- ou Ada 2005 : Allume(Objet.RetourneX, Objet.RetourneY);
  else
    Eteins(RetourneX(Objet), RetourneY(Objet));
    -- ou Ada 2005 : Eteins(Objet.RetourneX, Objet.RetourneY);
  end if;
    CaculeProchain (PointCourant);
  end loop;
  end;
```

Une autre façon d'aborder la question avec Ada est de définir un objet Figure abstrait. Pour cela on lui adjoint deux méthodes abstraites Allume et Eteins et la méthode Affiche basée sur les deux précédentes. L'objet ne peut plus alors être instancié. L'instanciation se fera à partir des enfants qui devront définir réellement les deux méthodes Allume et Eteins.

```
package Figure is
type Coordonnee is range -100 .. 100;
subtype Surface is Float;
type Instance is abstract tagged private;
 -- remarquez le mot clé "abstract" qui déclare l'objet abstrait
subtype Class is Instance'Class;
 -- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut l'instance et
tous ces descendants
procedure Positionne (Objet : in out Instance; X, Y : Coordonnee);
function RetourneX (Objet : in Instance) return Coordonnee;
function RetourneY (Objet : in Instance) return Coordonnee;
procedure Allume (Objet : Instance) is abstract;
procedure Eteins (Objet : Instance) is abstract;
procedure Affiche (Objet : Class; EstVisible : Boolean);
-- ou Ada 2005 : procedure Affiche (Objet : in Instance'Class; EstVisible : Boolean);
private
type Instance is abstract tagged record
-- remarquez le mot clé "abstract" qui déclare l'objet abstrait
        X, Y : Coordonnee := 0;
        end record;
end Figure;

package body Figure is
procedure Affiche (Objet : Class; EstVisible : Boolean) is
-- ou Ada 2005 : procedure Affiche (Objet : in Instance'Class; EstVisible : Boolean) is
  begin
  if EstVisible then
    Allume(Objet);
    -- ou Ada 2005 : Objet.Allume;
  else
    Eteins(Objet);
    -- ou Ada 2005 : Objet.Allume;
  end if;
  end;
end Figure;
```

Nous dérivons Point de Figure :

```
with Figure;
package Point is
type Instance is new Figure.Instance with private;
-- noter le mot clé "new" permettant l'héritage
-- et le mot clé private pour l'encapsulation
subtype Class is Instance'Class;
-- noter l'attribut "'Class" pour permettre le polymorphisme, Class inclut l'instance et
tous ces descendants
private
-- déclaration des méthodes réelles
procedure Allume (Objet : Instance);
procedure Eteins (Objet : Instance);
type Instance is new Figure.Instance with null record;
-- et le mot clé "with" pour ajouter des champs, ici on n'ajoute rien pour faire un point
end Point;

package body Point is
procedure Allume (Objet : Instance) is
begin
…
end;
procedure Eteins (Objet : Instance) is
begin
…
end;
end Point;
```

L'utilisation est naturelle :

```
with Point;
…
MonPoint : Point.Instance; -- instanciation de l'objet
...
Point.Positionne (MonPoint, 15, 25);
-- ou Ada 2005 : MonPoint.Positionne (15, 25);
Figure.Affiche (MonPoint, True);
-- ou Ada 2005 : MonPoint.Affiche (True);
...
end;
```

La méthode Affiche définie dans l'objet racine abstrait est alors valable pour tous les descendants. Mais les méthodes "Allume" et "Eteins" sont propres à chaque figure.

7) Polymorphisme :

Imaginons maintenant que nous voulions faire évoluer ces figures sur l'écran en suivant le principe :
a: on cache la figure
b: on déplace la figure
c: on affiche la figure

D'une part, on se rend compte ici que la gestion de l'affichage est propre à chaque figure comme codé plus haut.
D'autre part, le déplacement n'implique que les coordonnées de la figure, partie commune de nos objets. On va donc créer une seule procédure polymorphe "Deplace" pour l'objet Figure avec l'attribut "'Class". Elle sera ainsi utilisée par tous les descendants.

```
procedure Deplace (Objet : in out Class; DX, DY : Coordonnee) is
-- ou Ada 2005 :
--   procedure Deplace (Objet : in out Instance'Class; DX, DY : Coordonnee) is
  begin
  Affiche (Objet , False); -- on cache la figure
  -- ou Ada 2005 : Objet. Affiche (False);
  Positionne (Objet, Objet.X + DX, Objet.Y + DY); -- on déplace la figure
  -- ou Ada 2005 : Objet.Positionne (Objet.X + DX, Objet.Y + DY);
  Affiche (Objet, True); -- on affiche la figure
  -- ou Ada 2005 : Objet. Affiche (True);
  end;
```

Surprise, la procédure Deplace, bien qu'écrite pour l'objet Figure bénéficie de l'héritage dynamique et va donc bien appeler la procédure Affiche qui appelera ainsi les procédures Allume et Eteind correspondantes de l'objet réel :

```
with Cercle;
…
MonCercle : Cercle.Instance;
...
Cercle.Positionne (MonCercle, 10, 20, 30);
 -- ou Ada 2005 : MonCercle.Positionne (10, 10, 30);
Figure.Affiche (MonCercle, True);
-- ou Ada 2005 : MonCercle.Affiche (True);
Figure.Deplace (MonCercle, 10, 20);
 -- ou Ada 2005 : MonCercle.Deplace (10, 20);
…
end;
```

En fait, le comportement de Deplace est comparable à celui vu avec Affiche dans le cas de l'utilisation d'un objet abstrait.
Par contre, dans le cas de la première construction de la méthode Affiche avec une procédure générique, la méthode Deplace devra être aussi générique pour pouvoir instancier correctement Affiche.
Nous n'avons pas ce souci avec la seconde construction par méthodes abstraites.

Cela veut dire que l'intérêt des objets abstraits est d'utiliser le polymorphisme mais le polymorphisme s'utilise aussi sans.
Créons une procédure polymorphe AfficheAire pour l'objet Cercle avec l'attribut 'Class qui sera appelée par tous ses descendants et qui apellera la bonne fonction de calcul de l'aire suivant l'objet réel :

```
procedure AfficheAire (Objet : in Class; Commentaire : String) is
-- ou Ada 2005 :
--  procedure AfficheAire (Objet : in Instance'Class; Commentaire : String) is
  begin
  Put_Line(commentaire & Aire (Objet)'Img);
  -- ou Ada 2005 : Put_Line(commentaire & Objet.Aire'Img);
  end;
```

L'utilisation est directe. Surprise, l'aire correcte s'affiche pour les objets Cercle et Carré :

```
Cercle.AfficheAire(MonCercle, "Aire cercle : ");
...
Cercle.AfficheAire(MonCarre, "Aire carré : ");
```

L'utilisation est encore plus naturelle avec Ada 2005 :

```
MonCercle.AfficheAire("Aire cercle : ");
...
MonCarre.AfficheAire("Aire carré : ");
```

8) Héritage multiple :

Toujours avec l'objectif de séparation des problèmes à résoudre, la POO met à disposition du programmeur la possibilité d'hériter de plusieurs pères. Cela ne va pas sans créer des quelques difficultés de mise en oeuvre. Notamment lors de conflits de noms identiques entre composants de plusieurs pères qui peuvent être de types différents. Un fils ne peut pas avoir plusieurs composants de types différents ayant la même identification. La même constatation est à faire sur les primitives. Un fils ne peut pas avoir plusieurs primitives de comportement différent ayant la même identification.

Exemple :

```
type PA is tagged record
  D : Integer := 65;
  end record;
procedure Affiche (O : PA) is
  begin
  Put_Line (O.D'Img);
  end;

type PB is tagged record
  D : Character := 'B';
  end record;
procedure Affiche (O : PB) is
  begin
  Put_Line (O.D);
  end;

type F is new PA and PB with null record;
MonObjet : F;
```

Quels sont les composants de MonObjet ? D entier ou D caractère ?
Que va afficher l'appel MonObjet.Affiche ? Un entier ou un caractère ?

Ada dans sa version 2005 a diminué ces difficultés en restreignant les pères multiples à des objets abstrait sans composants. Comme Java, ce nouveau type objet a pris le nom d'interface.
Un fils peut ainsi hériter d'un père de type objet (comme l'héritage simple) et de plusieurs pères de type interface.

Le type interface permet de définir des primitives abstraites et des méthodes polymorphiques qui utilisent ces primitives et qui seront définies que plus tard dans l'objet héritant de l'interface.

Ada 2005 permet également de définir des méthodes "nulles" pour un objet interface. Ces dernières sont équivalents à des méthodes ayant un code vide par défaut.
Les méthodes d'un objet abstrait sont une sorte de contrat pour les descendants. L'héritage multiple est alors un regroupement des contrats qui permet la séparation des problèmes à résoudre en plusieurs contrats.

Reprenons notre exemple initial où notre objet de base Figure mélangeait des primitives de manipulation de structure géométrique de notre Figure, des primitives d'affichage et des primitives de mouvement.
Restructurons le ainsi en trois parties.

L'objet Figure restreint aux propriétés "géométriques" :

```
package Figure is
   type Coordonnee is range -100 .. 100;
   type Instance is tagged private;
   -- remarquez le mot clé "abstract" qui déclare l'objet abstrait
   subtype Class is Instance'Class;
   -- noter l'attribut "'Class" pour permettre le polymorphisme, Class
   --inclut l'instance et tous ces descendants
   procedure Positionne (Objet : in out Instance; X, Y : Coordonnee);
   function RetourneX (Objet : in Instance) return Coordonnee;
   function RetourneY (Objet : in Instance) return Coordonnee;
private
   type Instance is tagged record
   -- remarquez le mot clé "abstract" qui déclare l'objet abstrait
     X, Y : Coordonnee := 0;
   end record;
end Figure;
```

L'objet Dessine est tout à fait "générique" :

```
package Dessine is
   type Instance is interface;
   -- remarquez le mot clé "interface" qui déclare l'objet abstrait sans composant
   subtype Class is Instance'Class;
   -- noter l'attribut "'Class" pour permettre le polymorphisme, Class
   --inclut l'instance et tous ces descendants
   procedure Allume (Objet : in Instance) is abstract;
   procedure Eteins (Objet : in Instance) is abstract;
   procedure Affiche (Objet : in Instance'Class; EstVisible : Boolean);
end Dessine;
```

Enfin l'objet Figure_Mobile retrouve l'ensemble des propriétés :

```
  package Figure_Mobile is
    type Instance is abstract new Figure.Instance and Dessine.Instance with null
record;
    -- remarquez le mot clé "abstract" qui déclare l'objet abstrait
    -- remarquez le mot clé "and" permet l'héritage multiple
    subtype Class is Instance'Class;
    -- noter l'attribut "'Class" pour permettre le polymorphisme, Class
    --inclut l'instance et tous ces descendants
    procedure Deplace (Objet : in out Instance'Class; DX, DY : Figure.Coordonnee);
  end Figure_Mobile;
```

Remarquez bien que l'objet Dessine n'est pas spécifique aux figures mais il est tout à fait d'un emploi général ou "générique".

Pascal Pignard mars 2001, août-novembre 2005, septembre 2007.