HEH be
Sciences
et technologies

Avenue V. Maistriau 8a
B-7000 Mons
+32 (0)65 33 81 54
scitech-mons@heh.be

WWW.HEH.BE

Programmations avancées

Cours POO - UML

Bachelier en informatique et systèmes – Finalité Télécommunications et réseaux –Bloc 1.

Nomenclature :

En noir : Le texte

En bleu : Les annotations /remarques /Attention

En rouge : Les notions de codes, le pseudo-code,…

En vert : Les mots importants à connaître sur le bout des doigts

En orange : Les exercices

Table des matières

1. Les Classes et les Objets

1.1 Introduction

Un objet c'est quoi ?

Un objet c'est un ensemble d'entité de données que l'on nomme attribut (ou encore champs) et de fonctions que l'on nomme méthodes.
Seul vos méthodes vont avoir le pouvoir de manipuler vos données, que cela soit pour les utiliser ou les modifier.
Cette propriété s'appel en fait l'encapsulation. On va encapsuler nos données dans notre objet et donc celles-ci ne seront plus visible depuis « l'extérieur » de notre objet. Pour exploiter notre objet, il faudra donc faire appel à nos méthodes.

Une Classe c'est quoi ?

La notion de classe généralise simplement la notion de type d'objet : *une classe n'est rien d'autre qu'une description (**unique**) pouvant donner naissance à plusieurs objets disposant de **la même structure de données** (attributs ayant le même noms et types) et des **mêmes méthodes**.*

Nos objets d'une même classe, seront différents grâce à leurs attributs qui auront des valeurs différentes mais seront de même type et ayant les mêmes méthodes pour les manipuler.

Nous pourrions par exemple imaginer une classe étudiant qui ont tous les méthodes : Etudier, Boire, Marcher, … et des attributs : Nom, Prénom, Age, ….

En POO, nous aurons donc généralement deux possibilités, soit créer une classe qui nous permettra de générer un ou plusieurs objets, soit utiliser des classes existantes pour créer nos objets.
On peut remarquer que c'est un peu comme la création de nos fonctions dans le cours du Q1.

1.2 Parlons en exemple pour mieux comprendre

Imaginons une classe nommée « Point » qui permettra de déplacer un point dans l'espace.
Cette classe aura besoin de 3 méthodes :

- initialise : qui permettra de donner les coordonnées cartésiennes d'un point
- deplace : pour modifier les coordonnées
- affiche : permettant d'afficher la position du point

Cette classe sera également constitué de deux attributs pour représenter les coordonnées (il est évident que nous aurions pu en prendre 3, utiliser les coordonnées polaires, … mais restons simple).

Imaginons cette classe déjà défini (et même bien définie) , pour nous faciliter la compréhension et l'utilisation de celle-ci .

1.2.1 Comment allons-nous pouvoir déclarer notre objet de type « Point » ?

On pourrait penser qu'une simple déclaration comme nous en avons l'habitude pour un entier :

entier n

Ici nous réservions un emplacement mémoire pour une variable de type entier et donc on déclarerait notre objet par défaut de la manière ci-dessous :

Point p

Ici on réserve alors un emplacement pour un objet de type « Point », c'est-à-dire en fait un emplacement pour chacun des attributs (les méthodes ne demandant pas d'emplacement mémoire réservés).

Cependant cela ne suffit pas et ceci dans le but d'aider les programmeurs à avoir plus de souplesse d'exploitation de leurs programmes.

Il nous faudra ici deux étapes :

- La première comme mentionné ci-dessus : *Point p*

Qui va réserver un emplacement dans la mémoire pour une variable nommé p de type Point et stockera la référence (l'adresse) de cet objet. La valeur de l'objet est pour l'instant non définie.

- Dans un deuxième temps, il faut instancier (créer) notre objet, c'est un peu comme l'allocation dynamique des tableaux vu au Q1. Dans ce bout de cours, qui ne se réfère à aucun langage spécifique, nous utiliserons pour cela la norme suivante :

Création Point

Si on se réfère à p qui est notre variable instanciée, nous écrirons donc :

p :=Création point

Figure 1 : Schématisation de la création d'un objet avec deux attributs

Attention qu'il faut noter qu'une telle démarche va faire intervenir 2 choses :

- Une déclaration « classique » comme vous la connaissez (ici celle de l'objet de type *Point)* : On exploitera celle-ci pour réserver un emplacement mémoire.

- Une instruction d'instanciation qui va donner naissance à l'objet en réservant son emplacement uniquement au moment de l'exécution de l'instruction correspondante.
-

Il est évident que la valeur de p pourra changer durant l'exécution par affectation

Remarques :

- Le vocabulaire est CAPITAL : *p* est une variable de type *Point*, tandis que l'objet référencé par *p* est un objet de type *Point*.
  Pour l'instanciation, c'est par contre pareil que l'on parle d'une classe ou d'une variable (on parle toujours de la création de …).

- Certains langages possède une certaine dualité dans le fait d'en plus de posséder la possibilité d'instancier à l'exécution, a possibilité de réserver un emplacement d'objet par une déclaration.

Il est temps d'essayer d'utiliser notre objet de type *Point.*

Nous avons donc notre variable *p* qui contient notre Objet de type *Point.*

Tout d'abord, il est bon de rappeler que les attributs de nos objets sont encapsulés dans celui-ci et donc qu'il n'est pas possible d'y accéder directement. Un peu comme essayer d'accéder à la précieuse bière sans avoir d'abord ouvert la bouteille.

Ici, nous n'avons même pas encore accès à leurs noms (notre bière est dans une bouteille oui mais laquelle est-ce ?). Il faudra donc utiliser les méthodes de notre classe *Point*.

Nous avons ici prévu une méthode *Initialise* qui va nous fournir des valeurs pour les coordonnées de notre point.

Pour appeler celle-ci, rien de plus simple que l'appel d'une fonction comme au Q1 :

- Il faut préciser les valeurs des paramètres requis par la fonction : Ici les deux coordonnées cartésiennes : par exemple 5 et 8
- L'objet auquel la méthode s'applique : ici notre référence par *p.*

p.initialise (5,8)

1.2.2 Définition de notre Classe

Pour l'instant, et dans le but de pouvoir analyser ce qu'est une classe, nous avions supposé qu'elle fût déjà créée. Mais il est temps de voir comment cela se produit et comment vous pouvez créer votre propre classe.

```
classe Point
{
        //Déclaration des attributs
        //Déclaration des méthodes
}
```

Pour nos attributs, nous désirions deux entiers pour représenter les coordonnées cartésiennes de notre objet. Ceci se réalise de façon tout à fait classique :

```
entier abs // l'abscisse
entier ord //l'ordonnée
```

Ou si vous souvenez des raccourcis : *entier abs,ord*

Pour nos méthodes, cela se passe comme pour une fonction. On choisit un en-tête (ou il faut préciser les paramètres) ainsi que le type du résultat souhaité.

Remarque : Comme nous sommes indépendants du langage dans ces exemples, nous prendrons comme convenance d'écrire le mot méthode devant nos déclarations.

```
méthode initialise (entier x, entier y)
```

Dans celle-ci, il faudra attribuer des valeurs à *abs* et *ord* . Nous nous contenterons ici d'écrire :

```
abs := x // affectation de la valeur de x à l'attribut abs
```

Comme nous sommes dans le même objet, nous pouvons convenir que l'attribut est bien celui de l'objet dans lequel la méthode est en cours de traitement.

Attention qu'il est à noter que nous allons dans l'exemple suivant définir notre classe et la programme l'appelant de manière totalement séparé, or dans vos codes, il faudra apprendre à lier ceux-ci comme une bibliothèque.

Définition de la classe *Point*

```
class Point
    {
        //Attributs
        entier abs
        entier ord

        //méthodes
        méthode initialise (entier x, entier y)
        {
            abs := x
            ord := y
        }
        méthode deplace (entier dx, entier dy)
        {
           abs := abs + dx
           ord := ord + dy
        }
        méthode affiche
        {
            Ecrire "je suis un point de coordonnées :" + abs +"," + ord
        }
    }
```

Utilisation de la classe

```
Point p,r           // Deux variables de type Point
p := Création Point // Création d'un objet de type Point
p.initialise(5,8)   // Méthode d'attributions des valeurs à nos attributs
p.affiche()         // Méthode affiche sur l'objet référencé par p
p.deplace (2.0)

r:= Création Point  //Création d'un autre objet de type Point
r.initialise (4,9)
r.affiche()
p.affiche()
```

Résultat :

Je suis un point de coordonnées 5,8
Je suis un point de coordonnées 4,9
Je suis un point de coordonnées 7,8

1.3 L'encapsulation à des conséquences

1.3.1 Méthodes d'accès et altération

Nous avons vu dans les points précédents que l'encapsulation ne permettait pas d'accéder directement à nos attributs, et que nous devions passer par des méthodes permettant ces modifications.

Par exemple écrire : *p.abs* = 9 ne fonctionnera pas

Cela semble un peu draconien, et complexe mais il faut savoir qu'il est toujours possible de rajouter à une classe des méthodes pour :

- Obtenir la valeur d'un attribut donné : On parle de méthode d'accès
- Modifier la valeur d'un ou plusieurs attributs : On parle de méthodes d'altération

Rajoutons donc deux méthodes d'altération à nos coordonnées :

```
méthode fixeAbs (entier x)
{
    abs := x
}

méthode fixeOrd (entier y)
{
    ord := y
}
```

Ou encore deux méthodes d'accès :

```
entier méthode valeurAbs
{
    retourne abs
}

entier méthode valeurOrd
{
    retourne ord
}
```

Mais Monsieur alors pourquoi on encapsule ? Vous nous faites coder pour le fun, car on pouvait le faire directement avant et c'était simple ! (Quoi que moi je dois refaire l'examen mais …)

Et bien non, tout cela dépend simplement de deux notions que nous allons aborder tout de suite : L'interface d'une classe et l'implémentation d'une classe

1.3.2 Les otions d'interface, de contrat et d'implémentation

L'interface d'une classe correspond aux informations dont on doit pouvoir disposer pour savoir utiliser celle-ci. Il s'agit :

- Du nom de la classe
- De la signature (nom de la fonction et type des paramètres) et du type de résultat éventuel de chacune des méthodes

Dans notre exemple, on pourrait résumer comme ceci :

classe Point
{
    Méthode initialise (entier, entier)
    Méthode deplace (entier, entier)
    Méthode affiche
}

Le contrat d'une classe correspond à son interface et à la définition de ses méthodes.

L'implémentation d'une classe correspond à l'ensemble des instructions de la classe écrites en vue de réaliser le contrat voulu.

L'un des grands atouts de la POO c'est qu'une classe peut parfaitement modifier son implémentation, sans que ceci n'ait de conséquences sur son utilisation (si on respect le contrat bien sûr !). Nous pourrions par exemple décider que nos points sont à présent représentés par leurs coordonnées polaires et non plus cartésiennes.
Attention que ceci ne peut être fait aux dépends de la classe, nos méthodes doivent garder leur signature et signification et donc par exemple *initialise* devrait par exemple continuer à recevoir des coordonnées cartésiennes. Ceci est donc une prise de tête inutile dans cet exemple mais le contexte lui est possible !
La bonne idée ici, serait de créer des méthodes travaillant sur un type de coordonnées et d'autres sur le second.

Exercice 1 :

Ecrire une classe Point ne disposant que de 4 méthodes fixeAbs, fixeOrd, valeurAbs, valeurOrd. Réécrire le petit programme du point 1.2.2 correspondant à cette nouvelle définition.

1.3.3 L'encapsulation possède aussi ses exceptions #rebelle

Jusqu'ici nous avons présumé que, en programmation orienté objet, les attributs étaient tous encapsulés dans une classe.
Mais cela n'est pas toujours vrai, presque tous les langages permettent une certaine fluidité à ce niveau, et permettent d'accéder à nos attributs depuis l'extérieure de notre classe.
On parlera ici de *statut d'accès* (ou simplement d'accès) à un attribut qui peut être :

- Privé : c'est le cas que nous avions considéré jusqu'ici
- Publique : c'est le cas ou l'on peut directement accéder à l'attribut pour le modifier, le lire.

Pour déclarer un attribut tel que notre attribut *abs* en publique, il suffit de noter :

public entier abs // l'attribut abs n'est donc pas encapsulé

Si l'on considère qu'un objet de type *Point* est instancié, par exemple avec une variable *p* qui y fait référence, nous allons pouvoir écrire :

écrire p.abs

ou encore

p.abs := 10

Mais dans la bonne pratique, nous allons essayer de limiter au maximum les appels directs aux attributs de manière publique.

En autre, il est possible de prévoir des accès privés pour les méthodes également, par exemple, si lors de l'exécution d'une classe, nous souhaitons une méthode n'ayant rien à voir avec le contrat de la classe mais qui est utilisée par d'autres méthodes de la classe.
Il faut donc surtout retenir que :

Par défaut, les attributs d'une classe sont privés et ses méthodes publiques

Remarquons qu'il existe d'autres statut d'accès que nous verrons plus tard.

1.4 Méthode appelant une autre méthode

Pour le moment nous avions simplement appelé une méthode d'une classe en l'appliquant à un objet. Mais comme nous l'avons vu au premier quadrimestre pour les fonctions, une méthode peut également en appeler une autre.
Il faut s'avoir qu'une méthode d'une classe peut appeler une méthode d'une autre classe, mais nous laisserons pour l'instant cela de coté car nous devrions parler de composition d'objets et nous ne l'avons pas encore vu.

Nous pouvons en revanche regarder comment une méthode d'une classe peut appeler une autre méthode de cette même classe.

```
classe Point
{
        Méthode afficheAbs(….. ) // affiche l'abscisse
        Méthode afficheOrd(….. ) //affiche l'ordonnée
        Méthode affiche (….. ) //affiche les deux coordonnées
}
```

Pour ne pas complétement redéfinir la méthode *affiche*, nous allons dire que celle-ci appel les méthodes *afficheAbs* et *afficheOrd*.

```
méthode affiche
{
        afficheAbs
        afficheOrd
}
```

Comme nous somme dans une même classe, il ne faut pas spécifier l'objet auquel se rapporte ces méthodes, car par convention, il s'agit de l'objet unique ayant appelé la méthode affiche.

1.5 Les constructeurs

1.5.1 Introduction

Reprenons notre classe *Point* dotée de ses méthodes *initialise, deplace, affiche*. On remarque que jusqu'ici il faut faire appel à la méthode *initialise* pour attribuer des valeurs à nos attributs. Si par malheur on oublie et que l'on fait :

```
Point p
P := Creation Point
p.deplace(3,5)
```

Les valeurs de nos attributs ne sont pas définies avant l'appel de la méthode *deplace* et donc (même si certains langages autorisent les valeurs par défauts) nous aurons un problème car nous allons modifier des valeurs inexistantes ou inconnues.

D'une manière plus générale, dans les langages dit orienté objet, on parlera de constructeur :

- Un constructeur se présente comme une méthode particulière de la classe, portant un nom conventionnel ; Nous conviendrons d'utiliser une nomenclature connue qui est d'utiliser le nom de la classe comme nom du constructeur.
- Un constructeur peut disposer de paramètres
- Le constructeur est appelé au moment de la création de l'objet et on peut (si besoin), lui fournir un ou plusieurs paramètres.

1.5.2 Adaptation de notre classe Point

Rajoutons un constructeur à notre classe *Point*

```
classe Point
{
        methode entier Point (entier x, entier y) //constructeur (même nom
que la classe)
        {
                abs :=x
                ord := y
        }
        methode deplace (entier dx, entier dy)
        {
                abs := abs + dx
                ord := ord + dy
        }
        methode affiche
       {
                Ecrire « je suis un point de coordonnées »,abs , « », ord
       }
}
```

Lors de la création de l'objet, nous devrons prévoir les paramètres pour notre constructeur. Par convention, nous allons procéder comme suit :

p := creation Point (2,6) //Allocation de l'emplacement pour un point et initialisation de ses attributs aux valeurs 2 et 6

Remarques :

1. Il est capitale de comprendre que l'instanciation d'un objet, que l'on appel par : creation Point (2,5)
   Va réaliser 2 opérations :
   - Allocation d'un emplacement mémoire pour notre objet
   - Appel éventuel de constructeurs existants

   Ces deux opérations ne sont pas dissociables. Un constructeur ne peut aucunement, jamais, impossible être appelé directement.
2. Ici notre constructeur sert simplement à donner des valeurs par défauts, cela sera souvent le cas, mais il faut savoir qu'il peut aussi réaliser d'autres actions : allocations d'emplacements dynamiques, connexion à une DB, …

1.5.3 Surdéfinition ou surcharge d'un constructeur

Au quadrimestre 1, nous avons qu'il était possible de surcharger, surdéfinir des fonctions et bien il en pareil pour les méthodes ou le constructeur d'une classe. Pour cela, il suffit de définir plusieurs constructeurs qui se différencie par le nombre de paramètres.

```
point p,q,r
p:= creation Point  //appel constructeur 1
p.affiche()
q:= creation Point(2) //appel constructeur 2
q.affiche
r := creation Point(3,5) //appel constructeur 3
r.affiche

classe Point
{
    methode Point   //Constructeur 1 sans paramètre
   {
      abs :=0
      ord := 0
        }
    methode Point (entier x)  //Constructeur 2 avec 1 paramètre
   {
      abs :=x
      ord := 0
        }

   methode Point (entier x, entier y)  //Constructeur 3 avec 2 paramètres
   {
      abs :=x
      ord := y
   }
   methode deplace (entier dx, entier dy)
   {
      abs := abs + dx
      ord := ord + dy
   }
   methode affiche
   {
```

```
        Ecrire « je suis un point de coordonnées »,abs , « », ord
    }
    entier abs
    entier ord
}
```

Résultat :
Je suis un point de coordonnées 0 0
Je suis un point de coordonnées 2 0
Je suis un point de coordonnées 3 5

1.5.4 Appel automatique du constructeur

Une fois que l'on ajoute un ou des constructeurs à une classe, on peut s'imaginer qu'il serait utile d'empêcher d'appeler cette classe sans l'utilisation d'un constructeur créé. Et ça tombe bien, c'est ce qui est prévu dans la plupart des langages.
Imaginons la classe *Point* :

```
classe Point
{
        Point (entier x, entier y)  //unique constructeur à 2 paramètres
        {….. }
}
```

Et cette déclaration :

```
Point p
```

Alors la ligne suivante sera interdite :

```
P := creation Point //interdit
```

Nous pouvons donc conclure que l'instanciation est possible dans ce cas :

- Soit lorsque la classe ne dispose d'aucun constructeur
- Soit lorsque la classe dispose au moins d'un constructeur sans paramètres

Lorsqu'une classe dispose d'au moins un constructeur, il n'est plus possible d'instancier un objet sans qu'il fasse appel à un de ceux-ci.

Remarques :

1. Nous n'avons pas prévu de mécanisme permettant à un constructeur de fournir un résultat
2. Nous savons qu'il est possible d'initialiser des variables locales lors de leur déclaration. Il en est donc de même pour les variables locales de nos méthodes. **Par contre, nous ne prévoirons aucun mécanisme permettant d'initialiser les attributs d'un objet directement.** Cela est possible dans certains langages, mais cela interfère avec le travail du constructeur et donc par soucis de précision et de rigueur, nous n'en ferons rien.

1.5.5 Exemple et exercices

Nous allons partir d'un exemple nous permettant de :

- Surdéfinir un constructeur
- Modifier l'implémentation d'une classe en conservant le contrat

On vous demande de réfléchir à une classe Carré qui va avoir pour interface :

```
Carre (entier)    // constructeur à 1 paramètre
Carre             // constructeur sans paramètre (défaut =10)
entier methode taille    //fournit la veleur du cote
entier methode surface    //fournit la valeur de la surface
entier methode  perimetre // fournit la valeur du perimetre
methode changeCote (entier) // modifie la valeur du coté
```

Voici une première possibilité :

```
classe Carre
{
    methode Carre (entier n )
    {
        coté := n    }
    methode Carre
    {
        cote := 10     }
    methode changeCote (entier n )
    {
        cote := n    }
    entier methode surface
    {
        entier s
        s:= cote * cote
        retourne s }
    entier methode perimetre
    {
        entier p
        p := 4 * cote
        retourne p
    }
    entier cote
}
```

Mais il est possible de réfléchir un peu autrement et de voir une autre manière de réaliser l'implémentation de notre classe. Où l'on prévoit d'autres attributs comme périmètre et surface.
Les méthodes *perimetre* et *surface* vont devenir de simples méthodes d'accès aux attributs et n'ont plus à exécuter les calculs à chaque appel.
En revanche, les calculs sont effectués par les méthodes qui sont susceptibles de modifier la valeur du côté, soit ici nos constructeurs et la méthode *changeCote*.
Pour simplifier l'écriture, nous allons prévoir deux méthodes privées (*calculPerimetre* et *calculSurface*). Dont rappelons le, l'usage est réservé aux méthodes de la classe.

```
-   classe Carre
    {
       methode Carre (entier n )
       {
          cote := n
          calculPerimetre
          calculSurface    }
       methode Carre
       {
          cote := 10
          calculPerimetre
          calculSurface    }
       entier methode taille
       {
          retourne cote    }
       methode changeCote (entier n )
       {
          cote := n
          calculPerimetre
          calculSurface    }
       entier methode surface
       {
          retourne surface    }
       entier methode perimetre
       {
          retourne perimetre    }
       privé methode calculSurface
       {
          surface := cote *cote    }
       privé methode calculPerimetre {
          primetre := cote * 4    }
       entier cote
       entier surface
       entier perimetre }
```

On peut donc voir par cet exemple que tant que l'on respecte l'interface, il est possible de modifier presque à volonté l'implémentation d'une classe selon les besoins et les différents appels.

Exercice 2 :

Ecrire une classe nommée *Carac*, permettant de conserver un caractère. Elle disposera :

- D'un constructeur à un paramètre fournissant le caractère voulu
- D'un constructeur sans paramètre qui attribuera par défaut la valeur « espace » au caractère
- D'une méthode nommée *estVoyelle* fournissant la valeur *vrai* lorsque le caractère est une voyelle et *faux* lorsque celui-ci est une consonne

Composer un petit programme utilisant cette classe.

Exercice 3 :

Ecrire une classe *Rectangle* disposant de :

- De trois constructeurs : un sans paramètre dont les deux dimensions valent 1, le deuxième avec un paramètre utilisé pour les deux dimensions et enfin le troisième avec deux paramètres pour chacune des dimensions. Celles-ci seront de type *réel*
- Une méthode *perimetre* fournissant le résultat du périmètre du rectangle
- Une méthode *surface* fournissant en résultat la surface du rectangle
- Une méthode *agrandit* disposant d'un paramètre de type *réel* qui servira de multiplicateur aux dimensions du rectangle

Ecrire un petit programme utilisant cette classe.

Exercice 4 :

Ecrire une classe nommée *Reservoir*, implémentant l'interface et le « contrat » suivants :

```
methode Reservoir (entier n) // crée un réservoir de capacité maximale n
entier méthode verse (entier q) //ajoute la quantité q au réservoir si
                                  // possible sinon, on ne verse que ce qui
                                  // est possible. Fournit également en
                                  // résultat la quantité réellement ajoutée
entier methode puise (entier q) // puise la quantité q si possible sinon on
                                  // puise le reste et on fournit en résultat la
                                  // quantité réellement puisée
Entier methode jauge            // fournit le « niveau » du réservoir
```

1.6 Mode de gestion des objets

Dans le début de ce chapitre, nous avons vu au point 1.2, une méthode souple et très répandue de la « gestion des objets ». Nous pourrions résumé en disant qu'il faut dissocier ce que nous avons appelé « la variable de type objet » (celle qui contient la référence de l'objet) de l'objet à proprement parlé. Celle-ci s'apparente aux variables que nous avions rencontrées jusqu'ici au quadrimestre 1.
L'emplacement de celle-ci est géré comme pour une simple variable (la mémoire statique pour les variables du programme principal, pile pour les variables locales aux fonctions ou méthodes).
Par contre, l'objet lui, voit son emplacement alloué dynamiquement au moment de l'exécution de la commande « *Creation* ». Nous parlerons à partir de ces lignes de gestion par référence.
Certains langages objet offrent un autre mode de gestion des objets, la gestion par valeur, qui cohabite avec le mode par préférence. Ici , on va considérer que la seule déclaration d'un objet entraîne la réservation d'un emplacement mémoire correspondant (un peu comme un tableau en C).
Si notre classe comporte un constructeur, la déclaration de l'objet doit préciser des paramètres pour ce constructeur.
On peut se référer à une forme du style :

Point p (3,6) //Réserve un emplacement pour l'objet p et attribue 2 Paramètres

Nous verrons plus loin, les vrais attraits de passer par un mode ou l'autre. Les exemples ici étant trop simple pour bien comprendre.
Il faut pour l'instant simplement retenir que , dans le mode par référence, on manipule une référence à l'objet, alors que dans le second on manipule directement l'objet lui-même et donc directement les valeurs de ses attributs.

1.7 Parlons langage de programmation

Le C++, C#, Java ou encore PHP sont très proche de ce que nous venons de voir. Nous avons par exemple :

- Les attributs et les méthodes qui peuvent être privés (private) ou publique (public). Il nous est donc possible de passer outre le principe d'encapsulation. A noter que le C++ par exemple est privé par défaut alors que le PHP est lui publique
- On peut disposer de constructeurs. Dans les langages ci-dessus sauf PHP, ils portent le nom de la classe et peuvent être surdéfinis (surchargé). En PHP par contre, il ne peut pas l'être et se nomme _construct mais il peut prévoir des valeurs par défaut pour ses paramètres.

Python lui est un peu à part, il aime l'originalité. Premièrement, dans une méthode, il faut préciser le mot clé *self* (un paramètre implicite qui correspond à l'objet appelé). Deuxièmement, le suffixe *self* doit apparaitre devant les noms des attributs de la classe. Par défaut, tous les membres sont publiques en Python, mais l'on peut obtenir des membres privés en les préfixant de deux underscore __ . Le constructeur lui se définit par un nom spécifique *__init__* , il ne peut pas être sur défini mais on peut lui attribuer des valeurs par défauts.

Utilisation d'une classe

Java, C# et PHP utilisent une gestion par référence dans laquelle on utilise le mot clé *new* (au lieu de notre fameux Création) pour instancier les objets :

```
Point p ; // ($p en PHP) : p est  une référence d'un objet de type Point
…
P = new Point (5,6) ; //Crée un objet de type Point en appelant son constructeur
```

En python, on gère aussi les objets par référence mais il faut les instancier sous la forme :

```
P = Point( 5 , 7) ##crée un objet de type Point en appelant son constructeur
```

C++ par contre va gérer les objets par valeur, et la déclaration d'un objet va provoquer sa création :

```
Point p ( 4 ,4) ; // crée un objet de type Point en appelant son constructeur
```

Mais en C++, on peut également créer dynamiquement des objets en utilisant des pointeurs qui jouent alors le rôle de références :

```
Point *adp ; //adp est un pointeur contenant l'adresse d'un objet de type Point
adp = new Point (4, 4) ; //crée un objet de type Point en appelant son
constructeur
```

2. Les propriétés des objets et des méthodes

2.1 Affectation et comparaison des objets

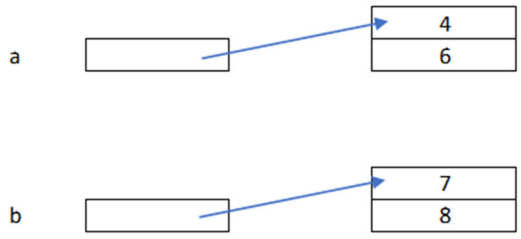
Dans le chapitre précédent, nous avons convenu que les objets étaient gérés par référence. Essayons de voir ce qu'il se passe au niveau de l'affectation et de leurs comparaisons.

2.1.1 Exemple

Imaginons partir d'une classe *Point* qui possède un constructeur) deux paramètres entiers et regardons les instructions suivantes :

Point a,b
……
a:= création Point (4,6)
b := création Point (7 ,8)

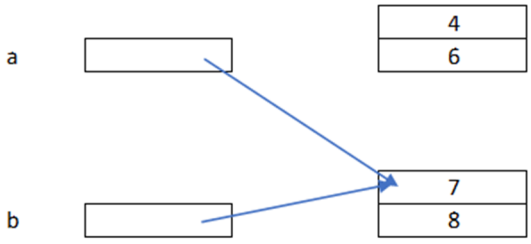
On se souvient donc que l'expression *Création Point (4,6)* va fournir une référence à un objet de type *Point* et c'est bien cette référence que nous appelons la variable de *a* de type *Point*. On obtient donc un résultat comme ceci :



On avait donc implicitement jusqu'ici accepté que l'instruction d'affectation pouvait s'appliquer à des variables d'un type. Ici, nous l'utilisons avec à droite du := une expression spécifiant le type d'objet. Mais, il est clair qu'on peut aussi se référer à une expression de variable de type objet. Comme par exemple :

a:=b

On va donc ici recopier la référence de *b* dans *a* et avoir comme situation :



On *a* donc *a* et *b* qui vont ici référencer le même objet. Et **non pas** deux objets avec les mêmes valeurs. Et donc toute modification effectuée par la variable *a* sera répercuté sur *b* .
Donc si par exemple, on essaye d'utiliser la méthode *deplace* qui existe dans la définition de notre *Point* au chapitre précédent.

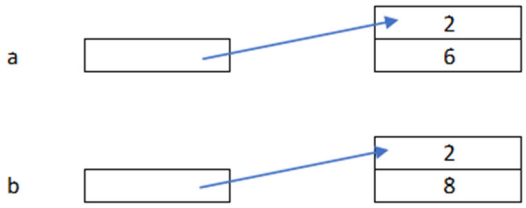
a.deplace(3,4)
b.affiche

On obtiendra donc comme affichage les valeurs : 10 et 12.

2.1.2 Deuxième exemple

Considérons l'exemple suivant :

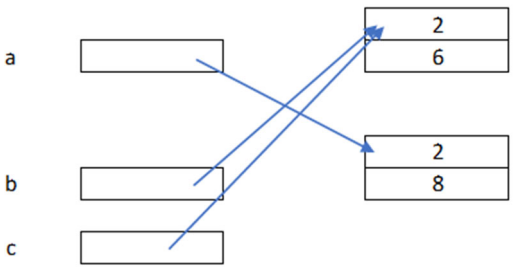
Point a,b,c
……
a:= création Point (2,6)
b := création Point (2 ,8)



Essayons de jouer avec ce que nous appelions une variable tampon au Q1. Nous aurons donc :

c := a
a := b
b := c

On obtient donc :



On a ainsi pu modifier les valeurs(les références) des variables *a* et *b* sans modifier la valeur des objets qui sont référencés.
Attention de bien comprendre qu'il n'existe que deux objets de type *Point* et trois variables de type *Point* (trois références, dont deux de celles-ci ont la même valeur)

Si l'on souhaitait changer le contenu de l'objet et non sa référence, cela sera plus complexe car il faudrait connaître les attributs de l'objet par les méthodes adéquates et par normes, ces attributs ne sont pas utilisable dans l'interface.

2.1.3 Comparaison d'objets

Nous connaissons du premier quadrimestres les comparaisons d'égalité (=) ou d'inégalité (<>) que nous appliquions à des types de bases. Il en sera pareille pour les objets. Nous conviendrons que cela se déroule sur les références des objets et non sur les valeurs de ceux-ci. L'égalité n'est donc réelle que si les références sont semblables et non pas des objets de même valeur.

Exemple :
```
Point p,q,r
p := Creation Point (2,7)
q := Creation Point (2,7)
si (p=q) alors …    // Ici, c'est faux
r := p
si (p=r) alors … // ici c'est vrai
```

Il est évidemment possible de créer une méthode appropriée qui fera la comparaison entre les valeurs des attributs eux-mêmes.

2.1.4 Cas des langages gérant les objets par valeur

Comme nous l'avons déjà expliqué, certains langages permettent de gérer nos objets par valeur ou par référence.
Si on gère ceux-ci par valeur, une simple déclaration d'un objet va alors réserver un emplacement complet pour notre objet.

Exemple : `Point a (2,6) //exemple de déclaration dans un langage par valeur`

Dans ce cas, si on l'écrit *a := b* (en considérant que *a* et *b* sont des objets de type *Point*), on va effectuer une copie de l'ensemble de attributs *b* dans ceux de *a*.
Ici, le principe d'encapsulation est quand même respecté, dans la mesure ou tous les attributs vont être recopiés dans notre nouvel objet.
Ces considérations semblables s'appliquent à la comparaison des objets. Dans ce genre de langage, la comparaison porte sur les valeurs des attributs et donc deux objets différents avec des valeurs d'attributs égales apparaîtront égaux.

```
Point a(3,5), b(3,5)
Si(a=b) ….  //ici on aura donc vrai comme réponse
```

2.2 Les objets locaux et leur durée de vie

La notion de variable locale n'est pas inconnue, la connaissance de la durée de vie d'une variable est connue selon l'endroit de sa déclaration. Une variable locale est par exemple, une variable définie dans une fonction, une boucle, …
Nous avons aussi expliqué que ce système fonctionne également dans une méthode.
Sachant ça, il est donc évident qu'on peut faire pareil avec des variables de type Objet. Prenons l'exemple de notre Objet Point.

```
fonction(…)
{
        Point p
        …..
}
```

C'est également le cas dans une méthode, mais avec une différence, on peut y distinguer deux cas, celui on l'on va déclarer une variable locale d'un type objet de la classe à laquelle elle appartient ou si elle est d'un type différent.
Ici nous n'allons pas tenir compte de ce fait, nous verrons plus tard ce qu'il implique. Un effet aura lieu au niveau de l'accès de la méthode aux attributs de l'objet correspondant.
Ce qu'il faut absolument retenir ici, c'est que nous parlons bien de variable de type Objet. C'est-à-dire qui est destinée à recevoir la référence d'un objet. L'objet(ou les objets) doit (devront) être créé(s) auparavant.

```
f(…)
{
        Point p
        ….
        p = Creation Point (…)
        ….
}
```

On peut se poser la question de la durée de vie de nos variables locales dans ce cas. Celle de la variable *p* et de l'objet référencé. Pour la variable comme toute variable locale, elle n'existera pas après la sortie de la fonction. Par contre, l'objet lui (si on est en gestion par référence bien entendu) continuera d'exister.
Mais l'objet même s'il existe, ne pourra être utilisé que si une référence reste active sur lui.
Autrement dit, comme dans notre exemple ci-dessus, l'objet *p* ne peut pas être utilisé en dehors de la fonction *f* car il est créé à l'intérieur de celle-ci. Une solution pour garder notre objet en dehors de la fonction serait par exemple de le fournir en résultat ou de faire une copie de notre objet.

En conclusion, on peut dire que la gestion par référence des objets, qui correspond à ce que l'on nomme « gestion dynamique de la mémoire », on s'affranchit totalement de la notion de localité des objets eux-mêmes ; il n'existe donc pas d'objets locaux mais des références locales.

2.3 Cas des objets transmis en paramètre

Jusqu'à ce point du cours, tous les paramètres de nos méthodes ne pouvaient passer en paramètres que des types de base ou des tableaux.
Mais il est aussi possible de passer des Objets en paramètre.

2.3.1 La transmission d'un objet en paramètre

Comme nous l'avons déjà expliqué, par convention, les paramètres d'une fonction sauf contre-indication sont transmis par valeur et les tableaux par référence. Les méthodes vont donc respecter les mêmes règles.

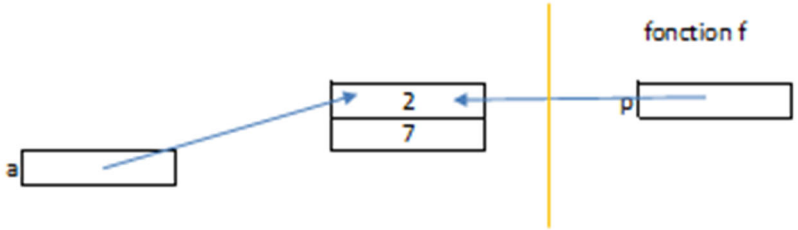
Pour ce qui est le transfère d'objet en paramètre, que cela soit pour une fonction ou pour une méthode, nous conviendrons également d'un transfère par valeur. Mais attention qu'ici on parle d'une valeur qui est en fait une référence, de sorte que la méthode (ou la fonction) reçoit en fait une copie de la référence de l'objet correspondant qu'elle peut donc modifier.

Voyons cela par un exemple pour que cela soit plus clair :

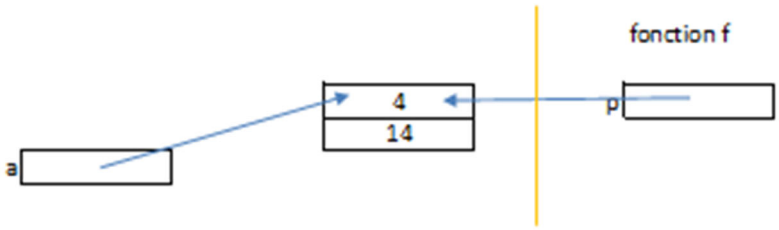
```
Point b
b := Creation Point ( 2,7)
f (b)
b.affiche
….
f (Point p)
{
        p.deplace (1,6)
}
```

➔ Résultat 3 13

Graphiquement cela donnerait :



A la fin de l'exécution de notre bout de code, l'objet référencé simultanément par *a* et *p* a vu sa valeur modifiée par l'appel de déplace.



Exercices :

Soit la classe suivante :

```
class Point
{
        methode deplace (entier dx, entier dy)
                {
                        abs =abs+dx
                        ord = ord + dy
                }
        methode valeurAbscisse { retourne abs}
        methode valeurOordonnee (retourne ord)
        entier abs,ord
}
```

Ecrire une fonction (indépendante) nommée ***grandit*** telle que l'appel de cette fonction ***grandit (p, n)*** , ***p*** étant un point et ***n*** une valeur entière, multiplie les coordonnées de ce point par la valeur ***n***.

2.3.2 L'unité d'encapsulation est la classe

Imaginons vouloir au sein de notre classe *Point* introduire une méthode nommée *coincide* (chargée de détecter la coïncidence éventuelle de deux points) qui nous retournera un résultat de type booléen. Son appel se fera obligatoirement de la manière suivante, considérant *a* comme un objet de type *Point* :
a.coincide (….)

Les « … » devront donc être remplacé par l'objet de type Point que l'on veut comparer à notre premier objet. Si on suppose que celui se nomme b :
a.coincide (b)

Ou évidemment vu que ce problème est symétrique :
b.coincide (a)

Réfléchissons maintenant à la manière d'écrire cette méthode.
L'entête pourrait être quelque chose comme :
Booléen coincide (Point pt)

Le corps quant à lui devra permettre de comparer les coordonnées de nos deux points. Comme précédemment, nous allons nous baser sur des coordonnées *x,y* . Nous allons donc avoir :

```
booléen coincide (Point pt)
{
        booléen res
        res := pt.x = x et pt.y=y
        retourne res
}
```

Mais il nous faut pour cela que notre méthode *coincide*, appelé par notre objet *a*, puisse accéder aux attributs privés de notre objet *b* qui est de la même classe. C'est en fait un processus connu qui se passe dans la plupart des langages objet et que l'on va traduire en disant :

> ***L'unité d'encapsulation est la classe et non l'objet.***

En français, nous dirons que seules les méthodes d'une classe peuvent accéder aux attributs privés de cette classe. . Cette autorisation concerna tous les objets de la classe et pas seulement l'objet impliqué dans l'appel.

```
Point a, b, c
a := creation Point (1,3)
b := creation Point (2,5)
c := creation Point (1,3)

classe Point
{
        methode Point(entier x, entier y )
        {
                abs := x
                ord := y
        }
        booleen methode coincide (Point pt)
        {
                booleen ok
                ok := pt.abs = abs et pt.ord = ord
                retourne ok
        }
        entier abs, ord
}

a=c
c=a
```

Remarque :

Il est évident qu'une méthode d'une classe reçoit en paramètre un objet d'une classe différente, elle n'aura accès qu'aux attributs ou méthodes publiques de l'objet.

2.3.3 Exemple

Jusqu'ici, nos exemples étaient un peu particuliers, ils étaient des cas d'écoles, on parlait d'une fonction ou d'une méthode mais les modifications n'avaient pas lieu. Essayons de nous référer à un exemple un peu plus concret, ou la méthode appelée va agir sur l'objet reçu en paramètre.

Nous allons introduire dans une classe *Point*, une méthode *Permute*, qui va être chargée d'échanger les coordonnées de deux points.

```
classe Point
{
        methode permute(Point p )
        {
                Point c                          //variable locale de type Point
                c := Creation Point (0,0)        //creation d'un nouvel objet
                c.abs = p.abs et c.ord = p.ord   //copie de l'objet p dans l'objet c
                p.abs := abs et ord := c.ord     //copie du point courant dans p
                abs :=c.abs et ord=c.ord         //cp de l'objet c dans l'objet courant
        }
        …
        entier abs, ord
}
```

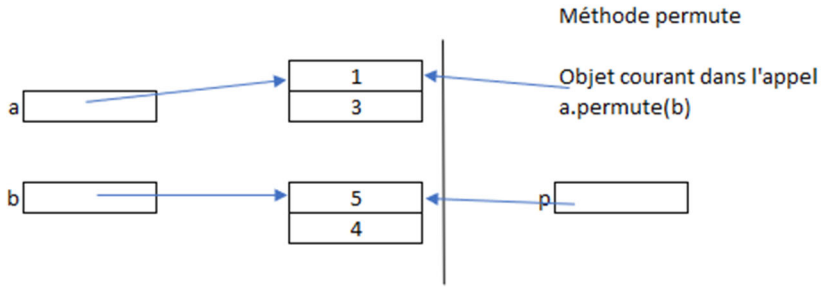
Cette méthode va recevoir en paramètre la référence à un *Point* dont elle va échanger les coordonnées avec celles du point concerné par la méthode.
Dans notre méthode *Permute*, nous avons créé un objet de type *Point* qui va permettre cette échange (il était aussi possible de le faire avec deux variables locales de type *Entier*).

```
Point a, b
….
a := creation Point (1,3)
b := creation Point (5,4)
```

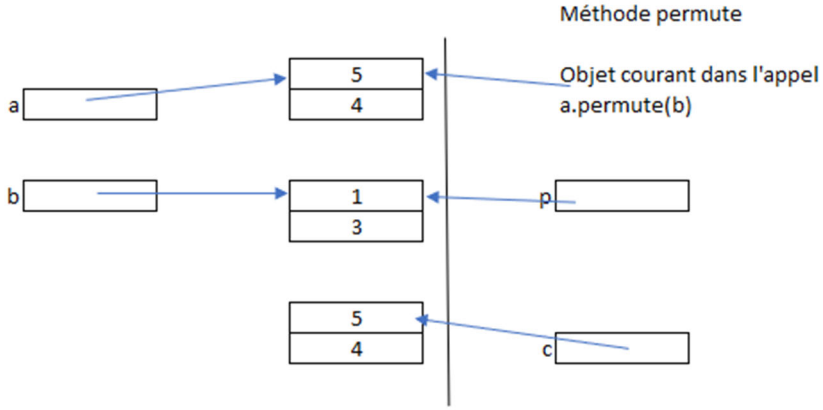
Si on appel permute :

```
a.permute(b)
```

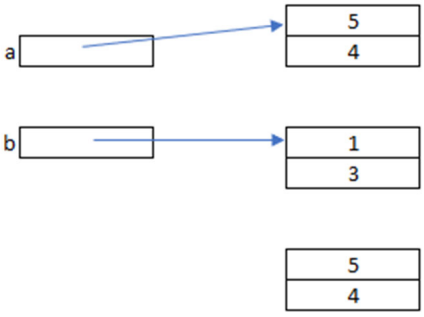
Voici le développement de nos variables de manières graphiques



A la fin de l'exécution de notre méthode (avant de faire la ligne retour) :



Une fois le retour fait, les variables locales à permute ne sont plus disponibles mais l'objet qu'on vient d'y créer lui existe toujours et donc nous avons :



Attention de bien réaliser qu'ici, ce ne sont pas les références contenues dans a et b qui ont changées, mais bien les valeurs des objets correspondants. On notera que la copie des objets est ici réalisée par une méthode de leur classe.
Il est évident que si l'on modifie l'implémentation de la *classe*, la méthode *Permute* doit être adaptée en conséquence.

Remarque :

1. Ici nous n'avons plus de référence sur l'objet crée par la méthode permute qui ne sera donc plus accessible (dans certains environnements, il est automatiquement détruit)
2. Dans les langages ou les objets sont gérés par valeur, il existe généralement deux possibilités de transmission d'un objet en paramètre d'une fonction ou d'une méthode :
   - Par référence : comme on vient de le voir
   - Par valeur : On effectue alors une copie des valeurs des attributs

Exercice :

Introduire dans une classe Point (dotée des attributs usuels abs et ord), un constructeur particulier recevant en paramètre un objet de type Point dont il recopiera la valeur des attributs dans l'objet à construire.

2.4 Un objet comme résultat

Jusqu'à présent, nous avions toujours supposé que le résultat d'une fonction était d'un type de base et qu'il était transmis par valeur, c'est-à-dire qu'on va effectuer une recopie de la valeur calculée localement dans notre fonction.
Attention de bien se souvenir, qu'une fonction ne peut pas fournir un tableau de résultat. Ces règles vont évidemment s'appliquer aux méthodes.

A partir de maintenant, nous allons convenir qu'une fonction ou une méthode peut fournir un résultat de type Objet.
Dans ce cas, il y aura aussi la recopie de la valeur concernée, mais ici on recopiera en fait la référence. On accèdera donc à l'objet fourni dans le programme.
Prenons un exemple avec notre classe *Point* qui à l'aide d'une méthode va nous donner la symétrie du point.

```
Point a, b
a := creation Point (1,3)
a.affiche
b := a.symetrique
b.affiche

classe Point {
        methode Point(entier abs, entier ord )
        {
                x := abs
                y := ord   }
        Point methode symetrie
        {
                Point res
                res := creation Point(y,x)
                retourne res  }
        methode affiche
        {
                Ecrire « Coordonnées : », x, « »,y
        }
        entier x,y  }
```

```
Coordonnées : 1 3
Coordonnées : 3 1
```

Notez bien que la variable locale *res* disparaît à la fin de l'exécution de la méthode *symetrie* . En revanche, l'objet instancié par *Creation Point (y,x)* continue lui d'exister.

Remarque :

Dans un langage où les objets sont gérés par valeur, il va exister deux possibilités de transmettre le résultat :

1. Par référence : on fonctionne comme ci-dessus ; il faut simplement faire attention à ne pas renvoyer la référence de l'objet local.
2. Par valeur : On va effectuer une copie de l'objet qui n'aura plus besoin d'exister après le traitement. (On recopie donc un objet local)

Exercice :

On va disposer d'une classe Point dotée des attributs usuels abs et ord, ainsi que d'un constructeur à deux paramètres.
Ajouter une méthode fournissant en résultat ce que nous nommerons « somme » du point courant et du point fourni en paramètre.
Nous prendrons comme résultat, que la somme des deux points est un point obtenu en additionnant les abscisses ensemble et les ordonnées ensemble.

2.5 Les Attributs et les méthodes de classe

Dans la plupart des langages objet, on va pouvoir définir des attributs qui, au lieu d'exister pour chaque objet de la classe, n'existent qu'une seule fois pour tous les objets d'une même classe.
On parlera alors d'**ATTRIBUTS DE CLASSE** (ou champs de classe). De même, on va pouvoir définir des METHODE DE CLASSE qui vont pouvoir être appelées indépendamment de tout objet de la classe.

2.5.1 Attributs de classe

Considérons d'abords la définition (assez simpliste) de classe suivante :

```
classe A
{
        entier n
        réel y
}
```

Chaque objet de type A possède donc ses propres attributs n et y.
Exemple :

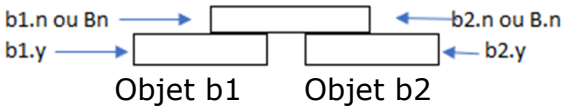
```
A a1,a2
a1 := Creation A
a2 := Creation A
```

Schématiquement on aurait :

Chaque objet *a1* et *a2* va donc bien disposer de ses propres attributs *n* et *y*. Mais (sinon ce chapitre n'aurait pas lieu d'être) il est possible que l'un ou plusieurs attributs ne puissent exister qu'en un seul exemplaire. Et cela peut importe le nombre d'objets de la classe.
Dans ce cours, nous conviendrons de faire appel au mot clé *deClasse* pour les définir.

```
classe B
{
        entier deClasse n
        réel y
}
B b1,b2
b1 := Creation B
b2 := Creation B
```

Schématiquement :



On peut se rendre compte que les notations *b1.n* et *b2.n* réfèrent le même élément. Cet attribut, existe en fait peut importe l'objet de la classe. Il sera préférable de le référer avec la notation *B.n* .

Il est évident que ces 3 notations ne sont valides que pour un attribut non privé. Il sera tout fois possible de prévoir des attributs de classe privé, mais ils ne seront alors accessibles que par des méthodes.

Si vous avez bien compris le principe d'une classe, vous pouvez convenir que à l'intérieur d'une méthode de la classe, nous atteindrons notre attribut *n* en le nommant simplement *n* et non *B.n* ou *b1.n* (Utiliser B ou b n'est pas une faute).

Dans les langages objet, nous conviendrons qu'il est possible d'initialiser notre attribut de classe dès sa déclaration. Attention que ceci n'avait pas été pris en compte pour les attributs usuels.
Un constructeur peut agir sur un attribut de classe mais il ne peut pas l'initialiser car :

- Celle-ci se ferait à chaque instanciation d'objet
- Par nature un attribut de classe doit posséder une valeur même si aucun objet de la classe n'a encore été instancié.

Exemple :

```
Cl a,b,c
écrire « En A »
a := Creation Cl
écrire « En B »
b := Creation Cl
c := Creation Cl
écrire « En C »

Classe Cl
{
        methode Cl
        {
                écrire « ++ création d'objet de type Cl »
                nb := nb+1
                écrire « Il y en a maintenant »,nb
        }
        entier deClasse nb :=0 //initialisation de l'attribut de classe nb
}
```

Résultat

```
En A
++ création d'objet de type Cl
Il y en a maintenant 1
En B
++ création d'objet de type Cl
Il y en a maintenant 2
++ création d'objet de type Cl
Il y en a maintenant 3
En C
```

2.5.2 Méthodes de classe

Comme nous venons de le voir pour les attributs de classe, qui n'existent qu'en un seul exemplaire, nous allons pouvoir créer des méthodes de classe qui auront également une indépendance de l'objet.
C'est par exemple le cas d'une méthode qui aurait pour but d'agir sur des attributs de classe.

On peut réussir à appeler cette méthode en la faisant porter artificiellement sur un objet de la classe (même si cette référence est en soit inutile).

Bien sûr, vous pouvez toujours appeler une telle méthode en la faisant porter artificiellement sur un objet de la classe (alors que la référence à un tel objet n'est pas utile). Là encore, la plupart des langages vous permettent de définir une méthode de classe. Nous conviendrons de la définir en utilisant la mention *deClasse* dans son en-tête. L'appel d'une telle méthode ne nécessitera plus le nom d'un objet particulier, mais simplement le nom de la classe correspondante.
Bien entendu, une méthode de classe ne pourra en aucun cas agir sur des attributs usuels puisque, par nature, elle n'est liée à aucun objet en particulier. Considérez cet exemple :

```
classe A{
        …
        réel x                      // attribut usuel
        entier deClasse n           // attribut de classe
        …
        méthode deClasse f          // méthode de classe
        {
                …                   // ici, on ne peut pas accéder à x, attribut usuel
                …                   // mais on peut accéder à l'attribut de classe n
        }
}
…
A.a
A.f                                 // appelle la méthode de classe f de la classe A
```

Voyons un exemple concret :

```
Cl a, b, c
écrire «En A : nb objets = », Cl.nbObj
a := Création Cl
écrire «En B : nb objets = »,Cl.nbObj
b := Création Cl
c := Création Cl
écrire «En C : nb objets = », Cl.nbObj

classe Cl
{
        méthode Cl
        {
                écrire«++ création objet de type Cl »
                nb := nb + 1
                écrire «il y en a maintenant», nb
        }
        entier methode deClasse nbObj
        {
                retourne nb
        }
        entier deClasse nb := 0
}
```

```
En A : nb objets= 0
++ création objet de type Cl
il y en a maintenant 1
En B : nb objets= 1
++ création objet de type Cl
il y en a maintenant 2
++ création objet de type Cl
il y en a maintenant 3
En C : nb objets= 3
```

Exercice :

Introduire dans une classe `Point`, comportant les attributs `abs` et `ord`, une méthode de classe permettant de tester la coïncidence de deux points fournis en paramètres.

Exercice :

Même question que dans l'exercice du point 2.4 , en faisant de somme une méthode de classe, au lieu d'une méthode usuelle.

D'une manière générale, les méthodes et attributs de classe s'avèrent pratiques pour permettre à différents objets d'une classe de disposer d'informations collectives. Nous en avons vu un exemple ci-dessus avec le comptage d'objets d'une classe. On pourrait également introduire dans une des classes *Point* déjà rencontrées deux attributs de classe destinés à contenir les coordonnées d'une origine partagée par tous les points.
Par ailleurs, les méthodes de classe peuvent également fournir des services n'ayant de signification que pour la classe même. Ce serait par exemple le cas d'une méthode fournissant l'identification d'une classe (nom de classe, numéro d'identification, nom de l'auteur...).
Enfin, on peut utiliser des méthodes de classe pour regrouper au sein d'une classe des fonctionnalités ayant un point commun et n'étant pas liées à un quelconque objet. C'est par exemple ce qui se produit avec les fonctions mathématiques dans certains langages objet ne disposant pas de la notion de fonction usuelle. Par exemple, la fonction mathématique sinus pourra se noter *Math.sin* (méthode de classe sin de la classe Math).

2.6 Des tableaux d'objets

Nous conviendrons qu'il est possible d'utiliser des tableaux d'objets. Une déclaration telle que :
tableau Point tp [3]
réservera l'emplacement pour un tableau de 3 références sur des objets de type *Point.*

Un exemple étant toujours plus parlant en voici un :

```
tableau Point tp [3]
entier i
tp[1] := Création Point(1, 2)
tp[2] := Création Point(4, 5)
tp[3] := Création Point(8, 9)

repeter pour i := 1 à 3
        tp[i].affiche

classe Point{
        methode Point (entier x, entier y)
        {
                abs := x ord := y
        }
        methode affiche
        {
                ecrire «Je suis un point de coordonnées », abs, « », ord
        }
        entier abs, ord
}
```

Je suis un point de coordonnées 1 2
Je suis un point de coordonnées 4 5
Je suis un point de coordonnées 8 9

Remarques :

1 Ici, nous n'avons considéré que des tableaux d'objets déclarés dans un programme. Nous pourrions également déclarer de tels tableaux localement à des fonctions ou à des méthodes. En revanche, l'utilisation de tableaux d'objets comme attributs d'une classe s'apparentent à la composition d'objets, étudiée dans un prochain chapitre.

2 Dans les langages gérant les objets par valeur, la déclaration d'un tableau d'objets conservera une présentation voisine de la précédente :
tableau Point tp [3]
Mais, cette fois, elle entraînera la réservation de l'emplacement mémoire pour 3 objets de type Point et elle devra provoquer l'appel du constructeur (s'il existe) pour chacun de ces objets. Dans la plupart des langages, il faudra qu'il existe un constructeur sans paramètre (ou aucun constructeur) puisque aucune information n'est prévue ici pour ce constructeur (certains langages disposent d'un mécanisme d'initialisation de tableau, à condition qu'on dispose d'un constructeur à un seul paramètre).

2.7 Les Autoréférence

Considérons l'application d'une méthode à un objet, par exemple :

a.déplace (4, 5)

Comme nous avons déjà eu l'occasion de le mentionner :

- cette méthode *déplace* doit recevoir une information lui permettant d'identifier l'objet concerné (ici *a*), afin de pouvoir agir convenablement sur lui ;
- la transmission de cette information est prise en charge automatiquement par le traducteur.

C'est ce qui permet aux instructions de la méthode d'accéder aux attributs de l'objet sans avoir besoin de préciser sur quel objet on agit.
Mais il peut arriver qu'au sein d'une méthode, on ait besoin de faire référence à l'objet dans sa globalité (et non plus à chacun de ses attributs). Ce sera par exemple le cas si l'on souhaite transmettre cet objet en paramètre d'une autre méthode. Un tel besoin pourrait apparaître dans une méthode destinée à ajouter l'objet concerné à une liste chaînée...

Les langages objet disposent d'une notation particulière de cet « objet courant ». Ici, nous conviendrons que le nom *courant* représente cet objet et qu'il s'utilise comme une variable du même type (donc comme une référence) :

```
classe A
{ .....
        public m(...) // méthode de la classe A
        { ..... // ici courant désigne l'objet ayant appelé la méthode
                // par exemple, si f est une fonction ayant un paramètre de type A
                // on pourra appeler f(courant) qui transmettra à f la référence
                // de «l'objet courant» (celui sur lequel porte m)
        }
}
```

Exemple d'utilisation des objets courants

À titre d'illustration du rôle de courant, voici une façon artificielle d'écrire la méthode *coincide* rencontrée au point 2.3.2 de ce syllabus :

```
booléen méthode coincide (Point pt)
{
        retourne pt.x = courant.x et pt.y = courant.y
}
```

Notez que l'aspect symétrique du problème apparaît plus clairement.
Ce type de notation artificielle peut s'avérer pratique dans l'écriture de certains constructeurs. Par exemple :

```
méthode Point(entier x, entier y)
{
        abs := x
        ord := y
}
```

peut aussi être écrit ainsi :

```
méthode Point(entier abs, entier ord) // notez les noms des paramètres muets ici
{
        courant.abs := abs // ici abs désigne le premier paramètre de Point
                           // l'attribut abs de l'objet courant est masqué ; mais
                           // on peut le nommer courant.abs
        courant.ord := ord
}
```

Cette démarche permet d'employer des noms de paramètres identiques à des noms d'attributs, ce qui évite parfois d'avoir à créer de nouveaux identificateurs, comme *x* et *y* ici.

2.8 Classes dites standards ou de type Chaîne

Nous savons du Q1 que les langages procéduraux disposent de « bibliothèques de fonctions ». De façon comparable, les langages objet disposent de « bibliothèques de classes ». Parmi les plus répandues, on trouve des classes permettant de manipuler des « structures de données élaborées » (vecteurs dynamiques, listes chaînées, piles, files d'attente, queues...), des dates, des heures et surtout des chaînes de caractères.

Une classe de type chaîne permet de manipuler des suites de caractères, en nombre quelconque, d'une manière beaucoup plus souple que ne le permettrait un tableau. Chaque langage propose ses propres fonctionnalités d'un tel type, avec beaucoup de disparités. C'est la raison pour laquelle nous ne l'avons pas introduit dans notre pseudo-langage de référence dans ce cours.

À simple titre indicatif, nous pouvons dire que ce type se nomme souvent *String*. Généralement, les libellés apparaissent comme des constantes de ce type, de sorte qu'ils peuvent être utilisés directement dans une affectation, par exemple :

```
String ch
ch := «bonjour»
```

En revanche, suivant les langages, l'accès à un caractère de rang donné de la chaîne pourra se faire :

- comme l'accès à un élément d'un tableau :
  ```
  écrire ch[i]   // affiche le ième caractère de la chaîne ch
  ```
- à l'aide d'une méthode particulière de la classe String, nommée souvent *charAt* :
  ```
  écrire ch.charAt(i)  // affiche le ième caractère de la chaîne ch
  ```

Dans certains langages, ces chaînes seront modifiables, soit au niveau du caractère, soit au niveau d'une « sous-chaîne » (suite de caractères de rangs donnés). Dans d'autres langages, de telles modifications ne seront pas permises. Dans tous les cas, on disposera de méthodes de recherche dans une chaîne donnée de l'occurrence d'un caractère donné ou d'une sous-chaîne donnée. La concaténation (mise bout à bout de deux chaînes) sera toujours présente, mais, là encore, elle pourra s'exprimer de manières différentes (opérateur +, méthode...).

En python cela donnerait par exemple :

```
class Point:

    def __init__(self, x=0, y=0):
        self.__abs = x;
        self.__ord = y


    def affiche(self):  ## la méthode affiche
        print("Je suis un point de coordonnées ", self.__abs, " ",
self.__ord)

    def symetrique(self):
        res = Point(-self.__abs, -self.__ord)
        return res

    def coincide(self, p):  # p est transmis par référence
        return p.__abs == self.__abs and p.__ord == self.__ord


    def permute(self, p) :   # p est transmis par référence
        c = Point (0, 0)
        c._abs = p.__abs ; c._ord = p.__ord
        p._abs = self.__abs ; p._ord = self.__ord
        self.__abs = c.__abs ; self.__ord = c.__ord


a= Point(1, 2)
b = Point(0,0)
c = Point(1, 2)
d = Point(-1, -2)
if a.coincide(c):
    print("a coincide avec c")
    print("a : ", end="")
    a.affiche()
b = a.symetrique()
print("b : ", end="")
b.affiche()

if b.coincide(d):
    print("b coincide avec d")
    a.permute (b)
    print("a ", end="")
    a.affiche()
    print("b ", end="")
    b.affiche()
```

Résultat :

```
a coincide avec c
a : Je suis un point de coordonnées  1   2
b : Je suis un point de coordonnées  -1   -2
b coincide avec d
a Je suis un point de coordonnées  0   0
b Je suis un point de coordonnées  -1   -2
```

3. La composition des objets

Jusqu'ici, nous n'avons considéré que des objets dont les attributs étaient d'un type de base ou tableau. Mais, bien entendu, une classe peut comporter des attributs qui sont eux-mêmes des variables de type classe. C'est cette possibilité que nous allons étudier ici en détail, ce qui va nous amener à considérer les problèmes qui peuvent se poser :

- au niveau des droits d'accès à ces attributs de type classe;
- dans la nature de la« relation » qui se crée entre les deux objets concernés.

Nous serons également amenés à vous présenter la distinction entre copie profonde et copie superficielle d'un objet. Enfin, nous vous montrerons comment réaliser une classe à instance unique (design pattern : singleton).

3.1 Une classe Cercle comme première approche

Supposons que l'on dispose de cette classe *Point* que, volontairement, nous avons privée de méthode *deplace* (notez que nous nous autorisons ici à placer plusieurs instructions sur une même ligne, en les séparant par un point-virgule):

```
classe Point
{
        méthode Point (entier x, entier y)
        {
                abs := x ; ord := y }
        méthode affiche
        {
                «Je suis un point de coordonnées». abs, « ». ord }

         entier abs, ord
```

Imaginons que nous souhaitions définir une classe *Cercle*, permettant de manipuler des cercles d'un plan, caractérisés par les deux coordonnées de leur centre et leur rayon (supposé de type *réel*).

Par souci de simplicité, nous supposerons que nous souhaitons que cette classe *Cercle* dispose de seulement 3 méthodes: un constructeur, une méthode affiche fournissant les caractéristiques du cercle et une méthode *deplace* permettant de déplacer le centre.
A priori, nous pouvons songer à utiliser un objet de type *Point* pour représenter le centre du cercle, de sorte que les attributs de notre classe *Cercle* pourraient être les suivants :

```
Point centre  // centre du cercle
réel rayon    // rayon du cercle
```

Voyons alors les problèmes qui peuvent se poser :

- d'une part au niveau des droits d'accès des méthodes de *Cercle* aux méthodes de *Point*

- d'autre part, au niveau des relations qui s'établissent entre les objets concernés.

3.1.1 Droits d'accès

Comment doter la classe Cercle d'une méthode affiche

Pour doter notre classe *Cercle* d'une méthode *affiche*, nous serions tentés de procéder ainsi :

```
méthode affiche
{
        écrire «coordonnées du centre », centre.abs, « », centre.ord
        écrire «rayon », rayon
}
```

Mais les attributs *abs* et *ord* appartiennent à la classe *Point* et non à la classe *Cercle*. Ce n'est pas parce qu'un objet possède un attribut de type classe que ses méthodes ont pour autant accès aux attributs privés de cette classe. Il s'agit là, en quelque sorte, d'une généralisation du principe d'encapsulation. S'il n'en allait pas ainsi, une modification de l'implémentation de la classe *Point* entraînerait une modification de l'implémentation de toutes les classes qui l'utilisent...

La seule chose que l'on puisse utiliser alors reste la méthode affiche (publique) de la classe Point, en écrivant ainsi la méthode affiche de *Cercle*

```
méthode affiche
{
        écrire «Je suis un cercle de rayon », rayon
        écrire « et de centre »
        centre.affiche
}
```

En fait, cette méthode afficherait l'information relative à un cercle de cette manière (ici, on suppose qu'il s'agit d'un cercle de coordonnées 1 et 2 et de rayon 4.5) :

```
Je suis un cercle de rayon 4.5 et de centre
Je suis un point de coordonnées 1 2
```

Certes, on trouve bien les informations souhaitées, mais leur présentation laisse quelque peu à désirer.
On déduit que, pour obtenir des résultats satisfaisants, il faudrait que la classe *Point* dispose :

- soit d'une méthode *affiche* présentant ses informations d'une manière différente ;
- soit de méthodes d'accès.

Comment ajouter une méthode déplace ?

Quant à la méthode *deplace*, il n'est pas possible de procéder ainsi :

```
méthode déplace (entier dx, entier dy)
{
        centre.abs = centre.abs + dx
        centre.ord = centre.ord + dy
}
```

Là encore, *deplace* est une méthode de la classe *Cercle*, pas de la classe *Point*. Elle n'a donc pas accès aux attributs privés de la classe *Point*.

Pour pouvoir réaliser la méthode *deplace*, il faudrait que la classe *Point* dispose :

- soit d'une méthode de déplacement d'un point,
- soit de méthodes d'accès et de méthodes d'altération.

Cet exemple montre bien qu'il est difficile de réaliser une bonne « conception » de classe, c'est-à-dire de définir le bon contrat et la bonne interface. Bien entendu, ici, l'exemple était simpliste et il serait tout à fait possible de définir une classe *Cercle* sans recourir à une classe Point. Mais, dans la pratique, les choses seront plus complexes et seul un contrat bien spécifié permettra de juger de la possibilité d'utiliser ou non une classe donnée...

Exercice :

Supposer que la classe Point précédente dispose des méthodes d'accès valeurAbs et valeurOrd ainsi que des méthodes d'altération fixeAbs et fixeOrd et écrire la méthode deplace de la classe Cercle.

Remarque :

Nous n'avons examiné que les problèmes d'accès se posant dans l'écriture des méthodes de Cercle. Il va de soi que, pour les objets de type Cercle, ceux-ci n'auront pas plus accès à un attribut(privé) de type classe qu'ils n'avaient accès à un attribut (privé) d'un type de base.

3.1.2 Les relations établies à la construction

Nous avons fait l'hypothèse que nous souhaitions doter notre classe *Cercle* d'un seul constructeur. Nous constatons que nous pouvons choisir de fournir en paramètres :

- soit les deux coordonnées du centre et le rayon,
- soit un objet de type *Point* (représentant le centre) et le rayon.

Comparons ces deux situations et leurs conséquences.

Les coordonnées en paramètres

Notre constructeur se présentera donc ainsi :

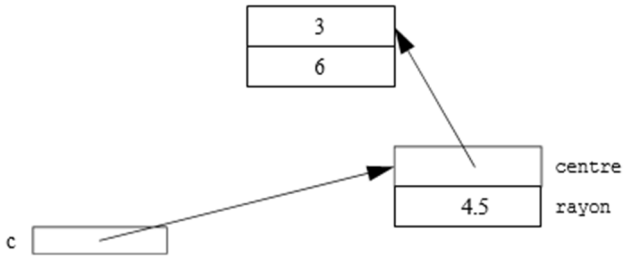
```
classe Cercle
{ .....
        methode Cercle (entier x, entier y, réel r)
        {
                centre := Creation Point (x, y) rayon := r
        }
        Point centre réel rayon
}
```

Notez bien qu'ici, le problème des droits d'accès aux attributs de type *Point* ne se pose pas car nous créons un nouvel objet de ce type à l'aide d'un constructeur à deux paramètres.

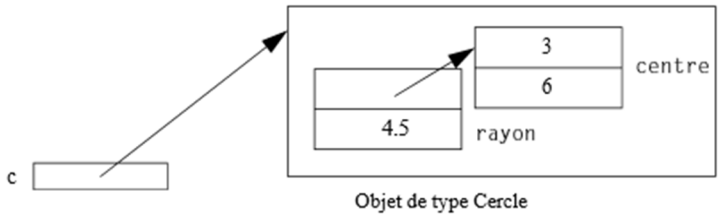
Considérons alors cet exemple d'utilisation de notre classe *Cercle*:

```
Cercle c
c := Création Cercle (3, 6, 4.5)
```

La situation peut être schématisée ainsi :



Ici, le point de coordonnées (3, 6) n'est accessible que par l'objet de type *Cercle* (référencé par c) qui en « possède » en quelque sorte la référence et qui se trouve donc être le seul à pouvoir éventuellement le modifier. Tout se passe comme si ce point faisait partie intégrante de l'objet de type *Cercle*. On pourrait rendre le lien entre ces deux objets plus explicites en schématisant ainsi la situation :



Passage d'un objet comme paramètre

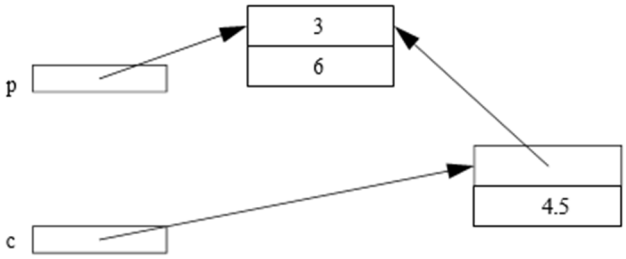
Voyons maintenant ce qui se produira si nous prévoyons de fournir un objet de type *Point* en paramètre du constructeur de la classe *Cercle*. Ce dernier se présentera donc ainsi:

```
méthode Cercle (Point pc, réel r)
{
        centre := pc
        rayon := r
}
```

Considérons alors une adaptation de l'exemple précédent d'utilisation de notre classe *Cercle* :

```
Point p Cercle c
p := Création Point (3, 6)
c := Création Cercle (p, 4.5)
```

La situation peut être schématisée ainsi :



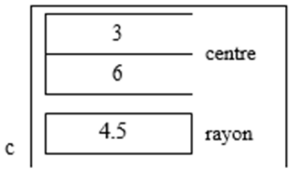
On constate que le point de coordonnées 3 et 6 est accessible indépendamment de l'objet de type *Cercle*, par la référence contenue dans *p*. Si les coordonnées de ce point étaient modifiées par le biais de cette référence (en supposant que la classe *Point* dispose des méthodes d'altération voulues), le centre du cercle s'en trouverait modifié en conséquence. En revanche, si l'on changeait la valeur de *p*, en lui faisant désigner un autre point, ceci n'aurait plus aucune incidence sur le centre du cercle...

3.1.3 Cas de la gestion par valeur

La règle évoquée reste toujours en vigueur dans le cas de la gestion des objets par valeur : les méthodes de *Cercle* n'ont pas accès aux attributs privés d'un objet de type *Point*. En revanche, les différences que nous venons d'évoquer entre les deux types de constructeurs s'estompent dans le cas de la gestion par valeur.

La première situation :

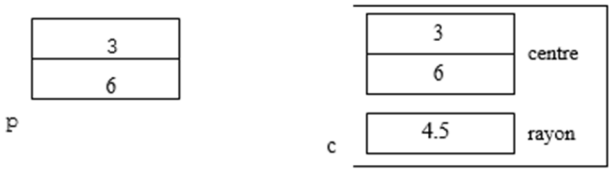
Cercle (3, 6, 4.5)



Il n'existe plus aucun objet de type *Point*, accessible à l'extérieur du cercle.
La seconde situation :

Point p (3, 6)
Cercle c (p, 4.5)

provoque la transmission par valeur du paramètre *p* , ce qui conduit à cette situation :



Ici, il subsiste un objet de type *Point*, extérieur, mais il est totalement indépendant du centre du cercle. Une modification de *p* n'aura aucune incidence sur le centre du cercle *c*.

Comme vous pouvez le constater, le mode de gestion des objets a de lourdes conséquences sur le comportement d'un programme. Sa « signification » peut s'en trouver fortement modifiée, ce qui justifie qu'on puisse parfois parler de « sémantique valeur » ou de « sémantique référence ».

Remarque :

Attention que certains langages de POO utilisant la gestion par valeur permettent aussi la gestion par référence et donc la création d'objets « hybrides ».

3.2 Un autre exemple, une autre classe : Segment

L'exemple précédent était quelque peu artificiel. Il avait surtout pour but de vous présenter les conséquences de la gestion par référence, ainsi que ses différences avec la gestion par valeur.

Nous vous proposons maintenant un exemple, toujours simple, mais plus réaliste, à savoir la création d'une classe *Segment*, permettant de manipuler des segments définis à partir de leur origine et de leur extrémité.

Nous supposerons que nous disposons de la classe *Point* suivante (notez la présentation des résultats fournis par affiche, différente de celle que nous avons utilisée jusqu'ici) :

```
classe Point
{
        methode Point (entier x, entier y)
        {
                abs := x ; ord := y }

        methode affiche { écrire «abscisse : », abs, «, ordonnée : », ord }

        methode deplace ( entier dx, entier dy)
        {
        abs := abs + dx ; ord := ord + dy }

        entier abs, ord
}
```

Nous allons chercher à doter la classe *Segment* des fonctionnalités suivantes :

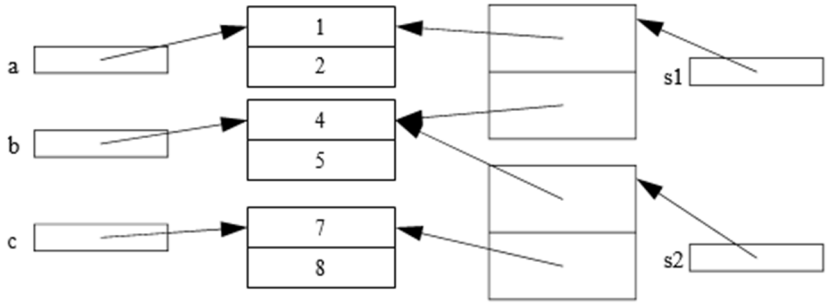
- constructeur, recevant deux points en paramètres,
- méthode *affiche* fournissant les coordonnées de l'origine et de l'extrémité. Elle pourrait se présenter ainsi :

```
classe Segment
{
        méthode Segment (Point o, Point e)
        {
        origine := o extremite := e }

        methode affiche
        {
                ecrire «-- origine    »
                origine.affiche
                ecrire «-- extrémite »
                extremite.affiche   }

        Point origine, extremite
}
```

Considérons alors cet exemple d'utilisation de *Segment* :

```
Point a, b, c
Segment s1, s2
a := Creation Point (1, 2)
b := Creation Point (4, 5)
c := Creation Point (7, 8)
s1 := Creation Segment (a, b)
s2 := Creation Segment (b, c)
s1.affiche
s2.affiche
```

Graphiquement nous aurons donc :

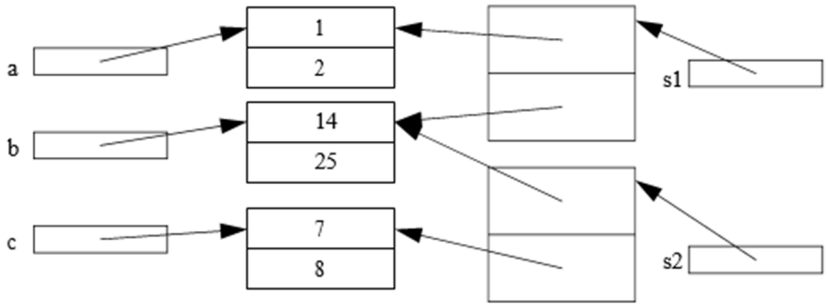


Et nos instructions appelant la méthode *affiche* donnerons :

-- origine
abscisse : 1, ordonnée : 2
-- extrémité
abscisse : 4, ordonnée : 5
-- origine
abscisse : 4, ordonnée : 5
-- extrémité
abscisse : 7, ordonnée : 8

Si maintenant, nous exécutons ces instructions:

b.deplace (10, 20)
s1.affiche
s2.affiche



La modification des coordonnées est bien répercutée sur les segments *s1* et *s2.*
Et le résultat à l'écran deviendrait :

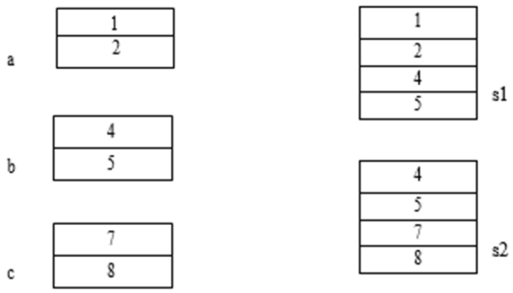
-- origine
abscisse : 1, ordonnée : 2
-- extrémité
abscisse : 14, ordonnée : 25
-- origine
abscisse : 14, ordonnée : 25
-- extrémité
abscisse : 7, ordonnée : 8

Remarque :

Dans nos langages où la gestion des objets est par valeur, nous pourrions avoir quelque chose du genre :

Point a(1, 2), b(4, 5), c(7, 8)
Segment s1(a,b), s2(b, c) s1.affiche
s2.affiche

Mais, cette fois, les attributs des points *a*, *b* et *c* auraient été recopiés dans les objets *s1* et *s2*, selon ce schéma :



La modification opérée par l'instruction *b.déplace (10, 20)* ne portera plus que sur le point *b* et elle n'aura aucune incidence sur les valeurs de *s1* qui ne représentera plus le « segment bc ».

Exercice :

Dans la classe Segment précédente, introduire une méthode nommée inverse, inversant l'origine et l'extrémité d'un segment.

Exercice :

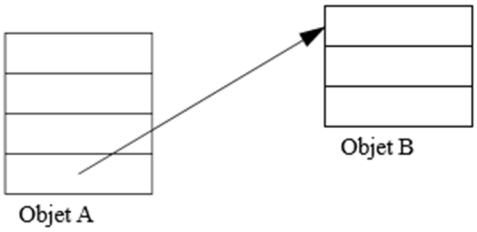
On dispose de la classe Point suivante :

```
classe Point
{
        méthode Point (entier x, entier y) { abs := x ; ord := y }
        méthode affiche { écrire «abscisse : », abs, «, ordonnée : », ord }
        méthode déplace ( entier dx, entier dy) {
                abs := abs + dx ; ord := ord + dy }
        entier abs, ord
}
```

Écrire une classe nommée Quadrilatère, permettant de manipuler des quadrilatères, déf- inis par 4 points, dotée d'un constructeur (recevant un tableau de 4 points en paramètre), d'une méthode affiche fournissant les coordonnées de chacun des 4 points du quadrilatère et d'une méthode déplace (déplaçant de la même manière les 4 points du quadrilatère).

3.3 La relation entre les objets

Nous venons de voir que définir un attribut de type classe n'est pas quelque chose d'aussi anodin que d'utiliser un attribut d'un type de base. En effet, cet attribut n'est en fait qu'une référence à un autre objet, ce que l'on peut schématiser ainsi de façon générale:



La question peut alors se poser de savoir si l'objet *B* peut exister, indépendamment de *A* ou non.
En fait, comme on l'a entrevu dans les exemples précédents, la situation diffère selon que *B* ait été créé par *A* (en général par son constructeur, mais il pourrait aussi s'agir d'une méthode) ou si sa référence a simplement été communiquée à *A* par le programme.

Dans le premier cas, lorsque l'objet *A* ne se trouve plus référencé, il en ira probablement de même de *B* (sauf si *A* en a communiqué la référence à « l'extérieur»). On traduit souvent cette situation en disant que *A* possède *B*; par exemple, on pouvait dire qu'un segment possédait deux points. Dans le second cas, en revanche, il est possible que *A* ne soit plus référencé alors que *B* l'est encore. On dit que *A* ne possède pas *B*. Dans nos deux exemples de classe *Cercle*, selon le constructeur utilisé, un cercle possédait un point dans le premier cas et pas dans le second (et nous avions illustré la première situation par un schéma montrant mieux cette appartenance).

Comme nous l'avons montré, dans les langages gérant les objets par valeur, l'existence d'un attribut de type objet conduit systématiquement à une relation de possession (quitte à ce que l'objet possédé soit en fait une copie d'un autre). D'une manière générale, cette relation de possession joue un rôle important en POO. On dit souvent qu'elle traduit la relation «a» (du verbe avoir) et nous verrons qu'elle doit être clairement différenciée de la relation« est» (du verbe être) qui sera induite par l'héritage.

Dans les méthodes de « conception objet)) telles que UML, on établit généralement une classification très fine des relations existant entre différentes classes, en distinguant notamment des liens d'association, d'agrégation ou de composition ou de simple utilisation. Mais, il est parfois difficile de retrouver l'expression de ces différentes relations dans les langages eux- mêmes, comme on a pu commencer à l'entrevoir précédemment.

3.4 Une copie profonde ou superficielle des objets

Nous avons vu que, dans les langages gérant les objets par référence, on se trouve souvent en présence de la recopie de cette référence et non pas de la recopie des attributs des objets concernés. Cela concerne à la fois les affectations et les transmissions en paramètre ou en résultat des objets.

Si on le désire, il reste toujours possible de réaliser explicitement la recopie de tous les attributs d'un objet dans un autre objet du même type. Toutefois, si les données sont convenablement encapsulées, il n'est pas possible d'y accéder directement. On peut alors songer à s'appuyer sur l'existence de méthodes d'accès et d'altérations de ces attributs privés.
Cependant, comme nous l'avons déjà évoqué, cette démarche demande la connaissance de l'implémentation effective de la classe correspondante et elle doit être reconsidérée à chaque modification de cette implémentation.

En fait, la démarche la plus réaliste consiste plutôt à prévoir dans la classe elle-même une méthode particulière destinée à fournir une copie de l'objet concerné, comme dans cet exemple:

```
classe Point
{
        methode Point(entier x. entier y ) {
                abs := x ; ord := y }

        methode déplace (entier dx. entier dy ) {
                abs := abs + dx  ord := ord + dy J

        methode copie{
                Point p         // référence locale à un objet de type Point
                p := Creation Point (abs, ord)
                retourne p  }
        …
        entier abs, ord
}
```

```
Point a, b
a := Création Point (1, 2)
a.déplace(3,5) // a a pour coordonnées 4,7
b := a.copie // b contient la référence d'un nouveau point de coordonnées 4, 7
```

Cette démarche ne présente aucune difficulté tant que la classe concernée ne contient pas d'attributs de type classe. Dans le cas contraire, il faut décider si leur copie doit porter, à son tour, sur les objets référencés plutôt que sur les références. Par exemple, avec la classe *Segment* présentée précédemment, on pourrait ne recopier que les références des points concernés ou, au contraire, recopier leur valeur.

On voit donc apparaître la distinction entre ce que l'on nomme:

- la copie superficielle d'un objet: on va simplement recopier la valeur de ses attributs, y compris ceux de type classe (qui sont donc des références);

- la copie profonde d'un objet: comme vu avant, on va recopier la valeur des attributs d'un type de base, mais pour ceux de type classe, on va créer une nouvelle référence à un autre objet de même type et de même valeur.

Comme on s'en doute, la copie profonde peut être récursive; autrement dit, un attribut de type classe peut très bien, à son tour, renfermer d'autres attributs de type classe...
On voit d'ailleurs que, pour des objets complexes, il pourra exister des niveaux intermédiaires entre copie profonde ou copie superficielle.
Dans le cas des langages gérant les objets par valeur, on notera que la copie induite par les affectations ou la transmission par valeur des paramètres correspond à une copie profonde. Mais ces langages permettent généralement de créer des objets hybrides en faisant cohabiter des attributs de type classe (gérés par valeur) et des références à des objets ; dans ce cas, on comprend que ces dernières ne sont plus soumises à une copie profonde. Généralement, le programme peut alors fournir une méthode de son cru pour gérer convenablement (plus précisément, comme on le désire) cette copie, qu'elle soit induite par une affectation ou par le passage en paramètre.

Exercice :

Doter la classe Segment présentée au point 3.2, d'une méthode de copie superficielle et d'une méthode de copie profonde. Pour réaliser cette seconde méthode, on prévoira d'adapter la classe Point en la dotant d'une méthode de copie.

3.5 La classe singleton comme bonus (optionnel)

Jusqu'ici, nous avons rencontré des situations dans lesquelles une classe *A* contenait un attribut de type classe *B*. Mais il est également possible qu'une classe *A* renferme un attribut de ce même type *A*.

Supposons par exemple qu'on l'on souhaite réaliser une classe nommée *Singleton* ne permettant d'instancier qu'un seul objet de sa classe.
Plus précisément, on souhaite que chaque tentative d'instanciation, à partir de la seconde, fournisse toujours la référence à un seul et unique objet. Compte tenu de la manière dont fonctionne le constructeur et du fait qu'il ne fournit pas de résultat, on voit qu'il n'est pas possible d'opérer cette gestion d'objet unique dans le constructeur lui-même. Les demandes d'instanciation doivent donc être adressées à une autre méthode de la classe *Singleton* que nous nommerons *creeInstance* et qui doit:

- instancier un objet lors de son premier appel ;
- fournir en résultat la référence de cet objet unique, et ceci aussi bien pour le premier appel que pour les suivants.

On voit que la méthode *creeInstance* ne peut être qu'une méthode de classe puisqu'on ne peut pas l'appliquer à un objet existant. D'autre part, la référence de l'instance unique devra être conservée dans un attribut de type *Singleton*. Cet attribut devra être accessible à la méthode de classe *creeInstance*; il devra donc s'agir d'un attribut de classe.

Voici ce que pourrait être le « canevas » de cette classe :

```
classe Singleton{
        Singleton méthode deClasse creeInstance
        {       // crée une instance (Création Singleton) s'il y a lieu, de référence s

                Retourne s  // fournit la référence de l'instance unique
        }
               // autres méthodes de la classe

        Singleton deClasse s  // référence à l'unique objet instancié
        // autres attributs de la classe }
```

Pour interdire l'instanciation directe par le programme, il ne suffit pas de ne pas prévoir de constructeur pour cette classe Singleton (revoyez éventuellement les règles concernant l'appel des constructeurs au paragraphe 1.5.4).
Il faut (artificiellement) la doter d'au moins un constructeur privé, dont la seule existence interdira l'instanciation directe d'un objet. Ce constructeur pourra éventuellement être vide ou, s'il possède un intérêt, être utilisé par la méthode *creeinstance*.

Enfin, pour gérer convenablement l'unicité de l'instance, nous utilisons un attribut de classe nommé créé, de type booléen, initialisé à faux, et mis à vrai dès qu'un objet a été instancié, afin de ne plus en créer d'autres. Rappelons qu'ici, nous avons convenu que tout appel, à partir du second, de la méthode *Creeinstance*, renvoie la référence à l'objet déjà créé. Voici ce que pourrait être le schéma de notre classe :

```
classe Singleton
{
        prive methode Singleton  //constructeur privé
        {  //initialisation des attributs
        }
        Singleton méthode deClasse creeInstance
        {
         si non créé alors  {
                s := Creation Singleton
                cree := vrai
                }
        retourne s

        // autres méthodes de la classe

        Singleton deClasse s        // référence à l'unique objet instancié booléen
        deClasse créé := faux
        // autres attributs de la classe
}
```

Remarque :

- Ce canevas ne serait pas utilisable dans un langage gérant les objets par valeur, puisqu'un objet de classe Singleton contiendrait, non plus une référence vers un objet de ce type, mais un objet (complet), ce qui, bien entendu, n'est pas possible: un objet ne peut se contenir lui-même. Dans un tel cas, il faut, si le langage le permet, utiliser un pointeur ou une référence.

- Nous avons considéré ici une situation particulière : un objet de type A contient une référence à lui-même (et, de plus, il s'agit d'un attribut de classe). D'une manière générale, un objet al de type A pourrait très bien contenir une référence (de classe ou non) à un autre objet b de type A. Ce serait le cas, par exemple, dans une « liste chaînée » d'objets, dans laquelle chaque objet contiendrait la référence de l'objet suivant de la liste. Là encore, ces situations poseront problème en cas de gestion par valeur: si a1, a2 et a3 sont de type A, un objet a1 ne peut pas contenir un objet a2 qui, à sont tour, contiendrait un objet a3...

Exercice :

En s'inspirant de l'exemple précédent, écrire un canevas pour une classe nommée Doubleton, n'autorisant pas plus de deux instanciations. On utilisera également une méthode creeInstance, dont les deux premiers appels fourniront un objet de type Doubleton. Les appels suivants fourniront à tour de rôle la référence au premier ou au second objet.

Exemple :

Prenons des exemples dans des langages qui gère les objets par référence.

En python :

```
class Point :
    def __init__(self, x,y):
        self._abs = x
        self._ord = y
    def deplace(self,dx, dy):
        self._abs += dx
        self._ord += dy
    def affiche(self) :
        print ("abscisse ",self._abs, " ordonnée ",self._ord)
class Segment :
    def __init__(self,o,e ):
        self._origine =o
        self._extremite = e
    def affiche (self) :
        print ("-- origine ", end="")
        self._origine.affiche()
        print ("--extrémité", end="")
        self._extremite.affiche()

a= Point (1,2)
b = Point (4,5)
c = Point (7,8)
s1 = Segment (a, b)
s2 = Segment (b, c)
print ("Segment s1 (ab)")
s1.affiche()
print ("Segment s2 (be)")
s2.affiche ()
b.deplace(10, 20)
print ("on deplace b")
print ("Segment s1 (ab)")
s1.affiche()
print ("Segment s2 (be)")
s2.affiche ()
```

```
Segment sl (ab)
-- origine abscisse  1  ordonnée  2
--extrémitéabscisse  4  ordonnée  5
Segment s2 (be)
-- origine abscisse  4  ordonnée  5
--extrémitéabscisse  7  ordonnée  8
on deplace b
Segment sl (ab)
-- origine abscisse  1  ordonnée  2
--extrémitéabscisse  14  ordonnée  25
Segment s2 (be)
-- origine abscisse  14  ordonnée  25
--extrémitéabscisse  7  ordonnée  8
```

En java :

```
public class TstSegment
{ public static voici main(String args[J)
    {
        Point a= new Point (1, 2), b = new Point (4, 5), c = new Point (7, 8);
        Segment sl = new Segment (a, b), s2 = new Segment (b, c) ;
        System.out.println("Segment sl (ab)") ; sl.affiche() ;
        System.out.println("Segment s2 (be)") ; s2.affiche() ;
        b.deplace(10, 20); System.out.println("on deplace b");
        System.out.println("Segment sl (ab)"); sl.affiche();
        System.out.println("Segment s2 (be)") ; s2.affiche() ;
    }
}
class Point
(
    public Point (int x. int y){ abs = x ; ord = y ;}
    public voici deplace (int dx,int dy) { abs += dx ; ord += dy ; }
    public voici affiche()
    { System.out.println("abscisse : "+ abs + ". ordonnée : "+  ord);
    }
    private int abs, ord ;
class Segment
{
    public Segment (Point o, Point e) { origine= o ; extremite = e ;}
    public void affiche()
    {
        System.out.print("-- origine    ") ; origine.affiche();
        System.out.print ( "-- extrémité "); extremite.affiche();
    private Point origine, extremite ;
}
```

4. La POO et la notion d'héritage.

En POO, l'héritage est fondamental. Il va offrir la possibilité de « réutiliser» les « composants logiciels » que sont les classes, en offrant l'opportunité de définir une nouvelle classe, dite classe dérivée, depuis une classe existante dite classe de base. Celle-ci « hérite» d'emblée des fonctionnalités de la classe de base (attributs et méthodes) qu'elle pourra modifier ou compléter à volonté, sans qu'il soit nécessaire de remettre en question la classe de base.

Cette technique va offrir la possibilité de développer de nouveaux outils en se fondant sur un certain acquis, ce qui justifie le terme d'héritage. Comme on peut s'y attendre, il sera possible de développer à partir d'une classe de base, autant de classes dérivées qu'on le désire. De même, une classe dérivée va donc pouvoir à son tour devenir une classe de base pour une nouvelle classe dérivée.

Nous commencerons par vous présenter cette notion d'héritage et nous verrons son incidence sur les droits d'accès aux attributs et méthodes d'une classe dérivée, d'une part pour les objets, d'autre part pour les méthodes d'une telle classe. Puis nous ferons le point sur la construction et l'initialisation des objets dérivés. Nous montrerons ensuite comment une classe dérivée peut redéfinir une méthode d'une classe de base, en en proposant une nouvelle implémentation.

4.1 Comprendre le terme d'héritage

Imaginons repartir d'une classe point (sans constructeur dans ce cas) :

```
classe Point  {
        méthode initialise (entier x, entier y) { abs := x ; ord := y }

        méthode déplace (entier dx, entier dy) {
                abs := abs + dx ; ord := ord + dy ; }

        méthode affiche { écrire «Je suis en », x, « », y }

        entier abs, ord  }
```

Imaginons avoir besoin d'une classe *Pointcol*, destinée à manipuler des points colorés d'un plan. Une telle classe peut donc disposer des mêmes fonctionnalités que notre classe Point, auxquelles on va pouvoir adjoindre, par exemple, une méthode nommée colore, qui sera chargée de définir la couleur. Nous pouvons donc la définir en disant qu'elle «hérite» de *Point* et en précisant les attributs et méthodes supplémentaires. Nous procéderons comme suit :

```
classe PointCol deriveDe Point
{       // définition des méthodes supplémentaires
        //déclaration des attributs supplémentaires
}
```

Si nous prévoyons, outre la méthode *colore*, un attribut nommé *couleur*, de type entier, destiné à représenter la couleur d'un point, voici comment on peut écrire la définition de la classe *Pointcol* :

```
classe Pointcol deriveDe Point
{
        methode colore(entier c) I
        {
                couleur := c }

         entier couleur
}
```

Disposant de cette classe, nous pouvons déclarer des variables de type *Pointcol* et créer des objets de ce type de manière usuelle, par exemple :

```
Pointcol pc ;
…
pc := Création Pointcol
```

Un objet de type *Pointcol* peut alors faire appel :

- aux méthodes publiques de *Pointcol*, ici *colore* ;
- mais aussi aux méthodes publiques de *Point* : *initialise*, *déplace* et *affiche*.

Notez que, bien que cela soit peu recommandé, il est théoriquement possible de prévoir certains attributs publics. Dans ce cas, un objet de type *Pointcol* aura également accès aux attributs publics de *Point*. Pour simplifier le discours, nous parlerons de membre pour désigner indifféremment un attribut ou une méthode. D'une manière générale, nous pouvons dire que:

Un objet d'une classe dérivée accède aux membres publics de sa classe de base.

Tout se passe comme si ces membres publics de la classe de base étaient finalement définis dans la classe dérivée.

En revanche, un objet d'une classe dérivée n'a pas accès aux membres (attributs et méthodes) privés de la classe de base. C'est bien ce à quoi on peut s'attendre, dans la mesure où l'on ne voit pas pourquoi un objet de la classe dérivée aurait plus de droits qu'un objet de la classe de base.

Par exemple, avec :

```
Pointcol pc
```

on pourra utiliser l'appel *pc.déplace* ou *pc.affiche*, mais on ne pourra (heureusement) pas accéder à *pc. abs* ou *pc.ord*.

Essayons avec un petit programme illustrant les possibilités avec des classes rudimentaire mais précises. Ici, nous allons créer à la fois un objet de type *Pointcol* et un objet de type *Point*.

```
// classe de base
classe Point {
        methode initialise(entier x, entier y) { abs := x ; ord := y }

        methode déplace (entier dx, entier dy) {
                abs := abs + dx ; ord := ord + dy }

        methode affiche { écrire «Je suis en», x,«»,y }

        entier abs, ord }

// classe dérivée
classe Pointcol deriveDe Point {
        methode colore (entier c) {couleur := c }

        entier couleur }

// Programme utilisant Pointcol et Point Pointcol pc
Point p
pc := Creation Pointcol
pc.affiche
pc.initialise(3, 5)
pc.colore(3)
pc.affiche
pc.déplace ( 2, -1)
pc.affiche
p := Creation Point
p.initialise (6. 9)
p.affiche
```

Résultat :

Je suis en 0 0
Je suis en 3 5
Je suis en 5 4
Je suis en 6 9

Remarques :

1 Une classe de base peut aussi se nommer « super-classe », une classe ascendante, une classe parente, une classe mère ou tout simplement une classe. De même, une classe dérivée peut aussi se nommer « sous-classe », une classe descendante, une classe fille ou tout simplement une classe. Enfin, on peut parler indifféremment d'héritage ou de dérivation, ou, plus rarement de sous-classement.

2 Nous avons déjà dit qu'un programme et les classes qu'il utilise pouvaient être fournies de façon séparée. Il en va bien sûr de même pour une classe et une classe dérivée. Bien entendu, il faudra faire en sorte que les informations nécessaires (par exemple définition de la classe de base et définition de la classe dérivée) soient :

- ou bien connues du traducteur de langage ;
- ou bien convenablement rassemblées au moment de l'exécution.

La démarche correspondante dépendra étroitement du langage et de l'environnement concernés. Dans tous les cas, on pourra créer, pour une classe de base donnée, autant de classes dérivées qu'on le souhaitera.

4.2 Les droits d'accès de la classe dérivée à sa classe de base

Nous venons de voir les possibilités pour un objet d'une classe dérivée d'accéder uniquement aux membres publics de sa classe de base.
Cependant, nous n'avons pas parlé de quel usage une méthode d'une classe dérivée peut avoir des membres (publics ou privés) d'une classe de base. Essayons de bien distinguer les membres privés et publiques pour ce qu'il en est.

4.2.1 Impossible pour une sous-classe d'accéder aux membres de la classe mère

En fait, les méthodes des classes dérivées n'ont pas plus de privilèges sur leurs classes de base que les méthodes des autres classes.

Une méthode d'une classe dérivée n'a pas accès aux membres (attribut ou méthodes) privés de sa classe de base.

Cette règle peut sembler restrictive, mais sans elle, créer une sous-classe impliquerait un viol de l'encapsulation à coup sûr. En d'autres termes, une modification de l'implémentation d'une classe mère va plus que probablement impliquer une modification de l'implémentation de la classe enfant.

Si l'on considère la classe *Pointcol* précédente, elle ne dispose pour l'instant que d'une méthode *affiche*, héritée de *Point* qui, bien entendu, ne fournit pas la couleur. On peut chercher à la doter d'une nouvelle méthode nommée par exemple *affichec*, fournissant à la fois les coordonnées du point coloré et sa couleur. Il ne sera pas possible de procéder ainsi:

```
méthode affichec // méthode affichant les coordonnees et la couleur
{
        écrire <<Je suis en », abs, « », ord     // NON : abs et ord sont privés
        écrire« et ma couleur est : », couleur // OK
}
```

Remarque : Dans certains langages on va pouvoir affaiblir cette contrainte mais nous y reviendrons.

4.2.2 La sous-classe accède aux membres publics

Une méthode d'une classe dérivée a accès aux membres publics de sa classe de base.

Donc si on réécrit la méthode *affichec*, on va pouvoir se baser la méthode *affiche* de *Point* :

```
methode affichec
{
        affiche()
        ecrire« et ma couleur est », couleur }
```

On va donc dire que l'appel *affiche* dans la méthode *affichec* est en fait équivalent à : *courant.affiche*

Ce qui correspond à dire qu'il applique la méthode *affiche* à l'objet (de type *Pointcol*) ayant appelé la méthode *affichec*.

On peut donc procéder de même pour définir dans *Pointcol* une nouvelle méthode d'initialisation nommée *initialisec*, chargée d'attribuer les coordonnées et la couleur à un point coloré :

```
methode initialisec (entier x, entier y, entier c)
{
        initialise (x, y)
        couleur := c
 }
```

```
classe Point
{
        methode initialise (entier x, entier y) {
                abs := x ; ord := y }

        methode deplace (entier dx, entier dy) {
                abs := abs + dx ; ord := ord + dy ; }

        methode affiche {
        ecrire «Je suis en », x, « », y }

        entier abs, ord
}

classe Pointcol deriveDe Point
{
        methode colore (entier c) {
                couleur := c }

        methode affichec{
                affiche()
                ecrire « et ma couleur est : », couleur }

        methode initialisec (entier x, entier y, entier c){
                initialise (x, y) couleur := c }
                entier couleur }

Pointcol pc1, pc2

pc1 := Creation Pointcol
pc1.initialise (3, 5)
pc1.colore (3)
pc1.affiche   // attention, ici affiche pc1.affichec      // et ici affichec
pc2 := Creation Pointcol
pc2.initialisec(5, 8, 2)
pc2.affichec
pc2.deplace (1, -3)
pc2.affichec
```

```
Je suis en 3 5
Je suis en 3 5 et ma couleur est : 3
Je suis en 5 8 et ma couleur est : 2
Je suis en 6 5 et ma couleur est : 2
```

Exercice :

On dispose de la classe suivante :

```
classe Point
        {
        methode initialise(entier x, entier y) {abs := x ; ord := y }
        methode deplace (entier dx, entier dy) { abs := abs + dx ; ord := ord + dy
}
        entier méthode valeurAbs {
                retourne abs }
        entier methode valeurOrd {
                retourne ord }
        entier abs, ord
```

Écrire une classe PointA, dérivée de Point, disposant d'une méthode affiche fournissant les coordonnées d'un point. Écrire un petit programme utilisant les deux classes Point et Point A. Que se passerait-il si la classe Point ne disposait pas des méthodes va1eurAbs et valeurOrd?

Exercice :

On dispose de la classe suivante:

```
classe Point
{
        methode fixePoint (entier x, entier y) { abs := x ; ord := y }
        methode déplace (entier dx, entier dy) { abs := abs + dx ; ord := ord + dy
}
        methode affCoord {écrire «Coordonnées : », abs, « », ord }
        entier abs, ord
```

Écrire une classe PointNom, dérivée de Point, permettant de manipuler des points définis par deux coordonnées entières et un nom (caractère) et disposant des méthodes suivantes :

- fixePoint Nom pour définir les coordonnées et le nom ;
- fixeNom pour définir (ou modifier) seulement le nom d'un tel objet;
- affCoordNom pour affiche les coordonnées et le nom.

Écrire un petit programme utilisant cette classe.

4.3 L'héritage et les constructeurs

Jusqu'ici, nous avons considéré des classes sans constructeur, aussi bien pour la classe de base que pour la classe dérivée. En pratique, la plupart des classes disposeront d'au moins un constructeur. Voyons quel va en être l'incidence sur l'héritage.

Supposons que notre classe *Point* dispose d'un constructeur à deux paramètres, jouant le même rôle que la méthode *initialise* :

```
classe Point
{
        methode Point(entier x, entier y) {
                1 abs := x ; ord := y }
        methode déplace (entier dx, entier dy) {
                abs := abs + dx ord := ord + dy }
        methode affiche {
                écrire <<Je suis en », abs,« », ord }
        entier abs, ord
}
```

Essayons maintenant d'imaginer que nous voulons simplement doter notre classe enfant *Pointcol* d'un constructeur avec trois paramètres :

```
Classe Pointcol derivede Point
{
        methode Pointcol (entier x, entier y, entier, c)
        {}
}
```

Quel doit alors être le travail du constructeur de Pointcol ?
Va-t-il pouvoir s'appuyer sur l'existence d'un constructeur de Point?

En fait, dans tous les langages, le constructeur de Pointcol pourra exploiter l'existence d'un constructeur de Point. Cependant, dans certains cas, le mécanisme sera « implicite», alors que, dans d'autres, il devra être« explicité».

Nous allons convener que:

Le constructeur d'une classe dérivée, s'il existe, doit prendre en charge la construction de la totalité de l'objet, mais il peut appeler le constructeur de la classe de base, en le désignant par le terme base.

On aura donc :

```
methode PointCol (entier x, entier y , entier c ) {
        Base(x,y) //appel du constructeur de classe de base avec 2 paramètres
        Couleur :=c  }
```

Réadaptons le programme du point 4.2.2 avec cette nouveauté (remplacement des *initialise* par des constructeurs :

```
classe Point {
        methode Point (entier x, entier y) {
                abs := x ; ord := y }
        methode déplace (entier dx, entier dy) {
                abs := abs + dx
                ord := ord + dy }
        methode affiche {
                ecrire «Je suis en », x, « », y }
        entier abs, ord  }

classe Pointcol deriveDe Point {
        methode colore (entier c){
                couleur := c }
        methode affichec{
                affiche()
                ecrire « et ma couleur est : », couleur }
        methode Pointcol (entier x, entier y, entier c){
                base (x, y)
                couleur := c }
        entier couleur }

Pointcol pc1, pc2
pc1 := Création Pointcol(3, 5, 3)
pc1.affiche   // attention, ici affiche
pc1.affichec  // et ici affichec
pc2 := Création Pointcol(5, 8, 2)
pc2.deplace (1, -3)
pc2.affichec
```

Résultat :

```
Je suis en 3 5
Je suis en 3 5
        et ma couleur est : 3
Je suis en 5 8
        et ma couleur est : 2
Je suis en 6 5
        et ma couleur est : 2
```

Remarques :

- Avec nos hypothèses (la prise en charge de la totalité de la construction de l'objet dérivé par son constructeur), si Pointcol ne disposait d'aucun constructeur, aucun constructeur de Point ne pourrait être appelé. On notera qu'ici cette contrainte est logique puisqu'on ne voit pas comment on pourrait fournir des paramètres au constructeur de Point. Les choses seraient toutefois moins évidentes si Point disposait d'un constructeur sans para- mètre. Ici, toujours avec nos hypothèses, ce constructeur ne pourrait pas non plus être appelé. Rappelons qu'il ne s'agit que d'une hypothèse qui se trouvera vérifiée dans certains langages et pas dans d'autres (certains pouvant prévoir l'appel automatique d'un constructeur sans paramètre).
- Ne confondez pas l'appel par super du constructeur de la classe de base avec un appel direct d'un constructeur (lequel, rappelons-le est interdit). Le premier se contente de compléter l'instanciation d'un nouvel objet, alors que le second, s'il était autorisé, appliquerait un constructeur à un objet déjà instancié.

Exercice :

On dispose de la classe Point suivante (disposant cette fois d'un constructeur)

```
classe Point
{
        methode Point (entier x, entier y){
                abs := x ; ord := y }
        methode affCoord { écrire «Coordonnées : », abs,« », ord }
        entier abs, ord
}
```

Écrire une classe Point Nom, dérivée de Point, permettant de manipuler des points définis par leurs coordonnées entières et un nom (caractère), et disposant des méthodes suivantes :

- constructeur pour définir les coordonnées et le nom ;
- affCoordNom pour afficher les coordonnées et le nom.

Écrire un petit programme utilisant cette classe.

4.4 L'héritage et la composition , qui, quoi, comment ?

Ci-dessus nous avons vu, comment l'héritage permet de réutiliser une classe existante.
Mais, comme nous avons pu le montrer dans le chapitre précédent, il en va déjà ainsi avec la relation de composition. Ici, nous vous proposons d'examiner les différences entre ces deux possibilités en les appliquant à une même classe *Cercle*, définie de deux façons différentes, à savoir:

- Par héritage d'une classe *Point* ;
- En utilisant un attribut de type *Point*, comme au chapitre précédent

Nous supposerons donc que nous disposons de cette classe *Point*, dotée ici uniquement d'un constructeur et d'une méthode *deplace* :

```
classe Point{
        methode Point (entier x, entier y){
                abs :=  x ; ord := y }
        methode deplace(entier dx, entier dy) {
                abs := abs + dx ; ord := ord + dy }
        entier abs, ord
}
```

Créons une classe dérivée nommée *Cercled*, dotée uniquement d'un constructeur (qui appelle celui de la classe de base pour initialiser les coordonnées du centre) :

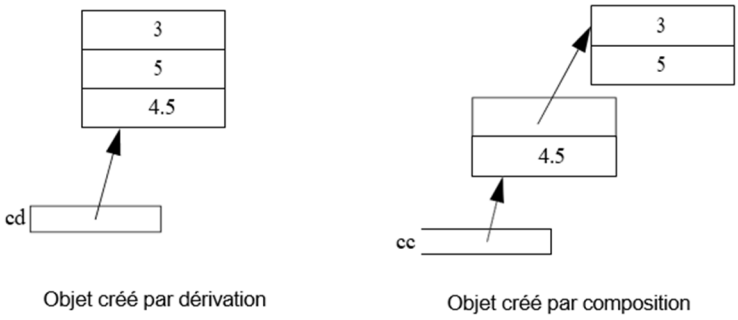
```
classe Cercled deriveDe Point
{
        methode Cercled (entier x, entier y, réel r) {
                base (x. y)
                rayon := r }
        réel rayon
}
```

Parrallèlement, nous allons créer une classe nommée *Cerclec* utilisant un attribut de type *Point* :

```
classe Cerclec
{
        methode Cerclec (entier x, entier y, réel r) {
                centre := Création Point(x, y) }
        Point centre
        réel rayon
}
```

On constate qu'ici, entre les deux formes de constructeur possibles (coordonnées du centre, ou objet de type *Point*), nous avons choisi la deuxième.
Considérons maintenant un objet de chaque type :

Cercled cd
Cerclec cc
cd := Création Cercled (3, 5. 4.5)
cc := Création Cerclec (3, 5, 4.5)
La situation est donc :



Objet créé par dérivation Objet créé par composition

Dans le cas de la classe dérivée Cercled, on voit qu'on a été amené à ne créer qu'un seul objet renfermant l'ensemble des attributs d'un cercle. En revanche, dans le cas de la classe composée Cerclec, on constate qu'on est en présence de relations entre deux objets qui seront plus ou moins« dépendants», suivant la manière dont s'est opérée la construction (ici, on a affaire à une relation de possession).

D'autres différences apparaissent également au niveau de l'utilisation des objets correspondants.
À l'objet cd, on peut, sans aucun problème appliquer la méthode déplace, héritée de sa classe de base :

cd.deplace(4, 5) // OK

En revanche, ce n'est plus possible avec cc

cc.deplace(4, 5) // impossible. déplace n'est pas une méthode de Cerclec

Il faudrait appliquer cette méthode à l'attribut *cercle* de *cc* (*cc.cercle.deplace(...)*), ce qui ici n'est pas possible puisque cet attribut est privé.

Déjà sur cet exemple, on voit apparaître la principale différence entre héritage et composition :

- **l'héritage** conduit à une relation de type« est»: ici *cd* est un cercle, mais comme il hérite de *Point*, c'est aussi un *Point* et on peut lui appliquer directement les méthodes de *Point*;

- la **composition** peut conduire à une relation de type« a» : ici, *cc* a un centre de type *Point*, mais ce n'est pas un objet de type *Point*: on ne peut donc pas lui appliquer les méthodes de la classe *Point*. On pourrait espérer les appliquer à son centre (de type *Point*), mais comme l'attribut correspondant est privé, ce n'est pas possible.

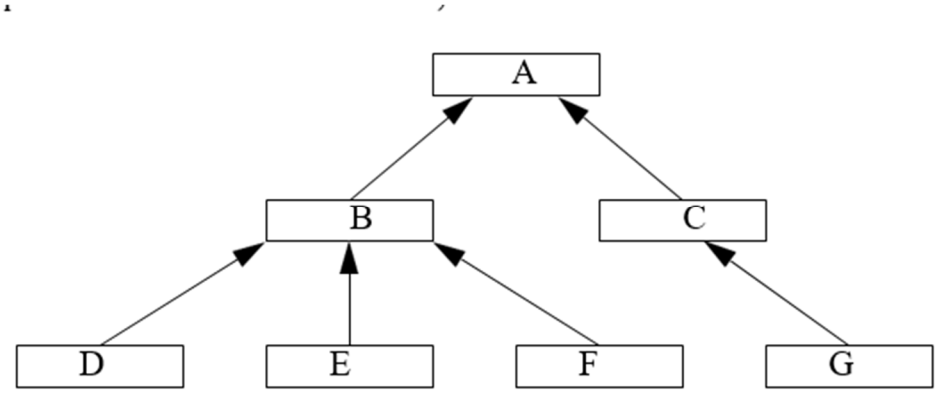
Nous verrons que cette relation «est» sera à la base de ce que l'on nomme le **polymorphisme**.

4.5 Un peu de math avec la dérivé d'une dérivé d'une dérivé…

Un peu comme en math, un peu comme dans la vie, tout ne se passe que sur deux niveaux, il est évidemment possible de créer des classes enfante d'une classe déjà elle-même enfant d'une autre et donc :

- d'une même classe peuvent dériver plusieurs classes différentes
- les notions classe de base ou dérivé sont relatives puisqu'une classe dérivée peut-être de base pour son enfant à elle.

On peut donc se retrouver avec un schéma qui pourrait par exemple être comme celui-ci-dessous (attention qu'ici on fait partir les flèches des enfants vers les parents mais il est possible de voir l'inverse) :



D est dérivée de B, elle-même dérivée de A. On dit souvent que D dérive de A. On dit aussi que D est une sous-classe de A ou que A est une super-classe de D. Parfois, on dit que D est une descendante de A ou encore que A est une ascendante de D. Naturellement, D est aussi une descendante de B. Lorsqu'on a besoin d'être plus précis, on dit alors que D est une descendante directe de B.

Notez que, nous sommes toujours partis d'une classe existante, à partir de laquelle nous avons défini une classe dérivée. Mais, dans la « conception » d'un logiciel, il pourra aussi arriver que l'on définisse une classe de base dans le but de regrouper des fonctionnalités communes à d'autres classes (par exemple, ici, A pour B et C).

4.6 La redéfinition d'une méthode

Pour le moment l'héritage ne servait qu'à rajouter des attributs/méthodes à une classe existante, sans pour autant remettre ce qui existait en cause.
Voyons un peu maintenant ce qu'est la redéfinition de méthode : Dans une classe dérivée, nous allons définir une nouvelle implémentation d'une méthode qui est définie dans la classe parente. C'est ce qui sera la base du polymorphisme.

4.6.1 La notion de redéfinition

Nous avons vu qu'un objet d'une classe dérivée peut accéder à toutes les méthodes publiques de sa classe de base.

```
classe Point{
        …
        methode affiche {
        ecrire <<Je suis en ». abs.« ». ord }
        entier abs, ord
}
classe Pointcol deriveDe Point
{
        …  // ici , on suppose qu'aucune méthode ne se nomme affiche entier
couleur
}
Point p
Pointcol pc
```

L'appel *p.affiche* fournit tout naturellement les coordonnées de l'objet *p* de type *Point*. L'appel *pc.affiche* fournit également les coordonnées de l'objet *pc* de type *Pointcol*, mais bien entendu, il n'a aucune raison d'en fournir la couleur.

C'est la raison pour laquelle, dans l'exemple dupoint 4.2.2, nous avions introduit dans la classe *Pointcol* une méthode *affichec* affichant à la fois les coordonnées et la couleur d'un objet de type *Pointcol*.

Or, manifestement, ces deux méthodes *affiche* et *affichec* font un travail semblable: elles affichent les valeurs des données d'un objet de leur classe. Dans ces conditions, il paraît logique de chercher à leur attribuer le même nom.
Cette possibilité existe dans la plupart des langages objet ; elle se nomme **redéfinition de méthode**. Elle permet à une classe dérivée de redéfinir une méthode de sa classe de base, en en proposant une nouvelle définition.

Nous pouvons donc définir dans *Pointcol* une méthode affiche reprenant la définition actuelle de *affichec*. Comme les coordonnées de la classe *Point* sont encapsulées et que cette dernière ne dispose pas de méthode d'accès, nous devons utiliser dans cette méthode affiche de *Pointcol* la méthode *affiche* de *Point*. Dans ce cas, un petit problème se pose ; en effet, nous pourrions être

tentés d'écrire ainsi notre nouvelle méthode (en changeant simplement l'en-tête *affichec* en *affiche*):

```
classe Pointal deriveDe Point {
        methode affiche {
                affiche
                ecrire« et ma couleur est», couleur}
        … }
```

Or, l'appel *affiche* provoquerait un appel de ... cette même méthode *affiche* de *Pointcol*, lequel provoquerait à son tour un appel de cette même méthode. Nous serions en présence d'appels **récursifs** qui, ici, ne se termineraient jamais. Il faut donc préciser qu'on souhaite appeler non pas la méthode affiche de la classe *Pointcol*, mais la méthode affiche de sa classe de **base**. Nous conviendrons de noter cela à l'aide du mot base, déjà rencontré:

```
methode affiche{
                base.affiche // appel de la méthode affiche de la classe de base
                ecrire« et ma couleur est»,couleur }
```

Dans ces conditions, l'appel *pc.affiche* entraînera bien l'appel de *affiche* de *Pointcol*, laquelle, comme nous l'espérons, appellera *affiche* de *Point*, avant d'afficher la couleur.

Exemple plus complet :

```
classe Point{
        methode Point(entier x, entier y) {
                abs := x ; ord := y }

        methode déplace (entier dx, entier dy){
                abs := abs + dx ; ord := ord + dy }

        methode affiche {
                ecrire «Je suis en », abs, « », ord
                entier abs, ord }

classe Pointcol deriveDe Point {
        methode Pointcol(entier x, entier y, entier c) {
                 base (x, y)
                couleur := c }

        methode affiche{
                base.affiche
        ecrire« et ma couleur est », couleur }

        entier couleur }
```

```
Pointcol pc
pc := Création (Pointcol(3,5, 3)
pc.affiche     //ici, il s'agit de affiche de Pointcol
pc.deplace (1. -3)
pc.affiche()
```

Résultat :

```
Je suis en 3 5
        et ma couleur est 3
Je suis en 4 2
        et ma couleur est 3
```

Remarques:

1. Une redéfinition de méthode n'implique pas obligatoirement un appel de la méthode de la classe de base et donc le mot clé base.
2. La redéfinition n'implique que les objets de la classe dérivée, la clase de base continue de fonctionner normalement.

4.6.2 La redéfinition de manière générale

Ici, nous avons considéré un exemple simple : la méthode affiche ne possédait aucun paramètre et aucun résultat.
Examinons maintenant des situations plus générales :

```
classe Point{
        .....
        methode dzplace (entier dx, entier dy) {       }
        .....}

classe Pointcol deriveDe Point{
        .....
        methode deplace (entier dx, entier dy) {       }
        .....}
```

Ici, la méthode déplace de *Pointcol* redéfinit la méthode parente. Elle est tout naturellement utilisée pour un objet de ce type, mais pas pour un objet de type *Point* :

```
Point p Pointcol pc
.....
p.déplace (3, 5)     // appelle la méthode déplace de Point
pc.déplace (4, 8) // appelle la méthode déplace de Pointcol
```

En revanche, considérons cette situation (sans trop nous interroger sur la signification du déplacement d'un point coloré !) :

```
classe Point{
        .....
        methode deplace (entier dx, entier dy) {      }
        .....}

classe Pointcol deriveDe Point{
         .....
        methode deplace (entier dx, entier dy, entier dc) {  }
        .....}

Pointcol pc
pc.déplace (3, 5)    // méthode appelée ?
```

Cette fois, l'instruction *pc.déplace(3, 5)* ne peut plus appeler la méthode déplace de *Pointcol* puisqu'elle ne prévoit pas le bon nombre de paramètres. On peut alors se demander si elle va appeller la méthode déplace de *Point*.
Autrement dit, *Pointcol* dispose- t-elle de deux méthodes déplace (comme si l'on avait affaire à une surdéfinition).
La réponse va malheureusement dépendre du langage concerné et elle tient à la manière dont ce langage va faire cohabiter ces notions (différentes) de:

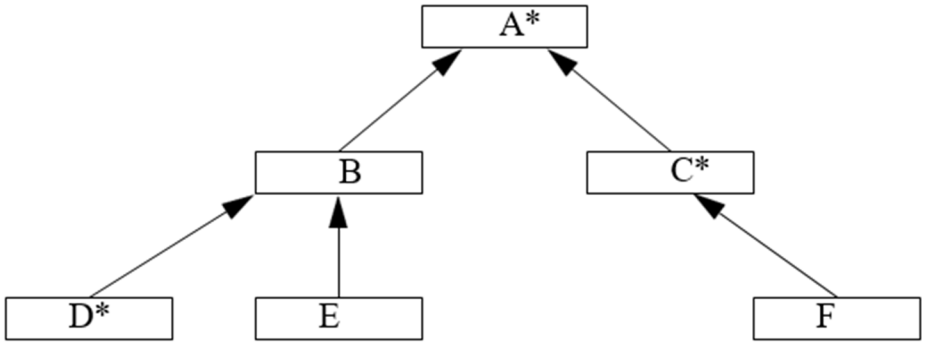
- **surdéfinition**: plusieurs méthodes accessibles, identifiées par leur signature
- **redéfinition**: une seule méthode accessible à un objet de type donné : une redéfinition masque une méthode de même nom d'un ascendant...

Ici, nous ne chercherons pas à examiner de telles situations en détail (ce qui peut devenir très complexe, et parfois un peu éloigné d'une bonne pratique de la programmation orientée objet). Nous nous contenterons simplement de définir précisément ce qu'est la situation de redéfinition (laquelle, encore une fois, sera fondamentale dans la mise en œuvre du polymorphisme), en convenant que:

Pour qu'il y ait redéfinition d'une méthode d'une classe par une classe dérivée, il faut que la méthode possède la même signature (nom de fonction et type des paramètres) et le même type de résultat dans la classe dérivée que dans la classe de base.

4.6.3 Redéfinition d'une méthode et dérivations successives

On peut rencontrer des arborescences d'héritage aussi complexes qu'on le veut. Comme on peut s'y attendre, la redéfinition d'une méthode s'applique à une classe et à toutes ses descendantes jusqu'à ce qu'éventuellement l'une d'entre elles redéfinisse à nouveau la méthode. Considérons par exemple l'arborescence suivante, dans laquelle la présence d'un astérisque(*) signale la définition ou la redéfinition d'une méthode *m* :



Dans ces conditions, l'appel de la méthode *m* conduira, pour chaque classe, à l'appel de la méthode indiquée en regard :

classe A: méthode m de A,
classe B: méthode m de A,
classe C: méthode m de C,
classe D: méthode m de D,
classe E: méthode m de A,
classe F: méthode m de C.

Exercice :
On dispose de la classe suivante:

```
classe Point
{
        methode Point(entier x, entier y){
                abs := x ord := y }
        methode affiche {
                ecrire «Coordonnées : », abs,« », ord }
        entier abs, ord
}
```

Écrire une classe nommée Point Nom, dérivée de Point, permettant de manipuler des points définis par leurs coordonnées et un nom (caractère), disposant des méthodes suivantes:

- constructeur pour définir les coordonnées et le nom ;
- affiche pour afficher les coordonnées et le nom.

4.7 Les droits d'accès et l'héritage

Nous avons déjà vu que la plupart des langages permettaient de s'affranchir plus ou moins du principe d'encapsulation en déclarant des attributs publics. Souvent, il est également possible de privilégier les classes dérivées par rapport aux autres, en permettant à leurs méthodes d'accéder à des attributs théoriquement encapsulés. La plupart du temps, cette« gestion» des droits d'accès se fait à l'aide d'un statut intermédiaire (nommé souvent protégé) entre le statut public et le statut privé.
Les membres protégés de la classe de base :

- sont accessibles aux méthodes de la classe dérivée (comme s'ils étaient publics) et non accessibles aux objets de cette classe (comme s'ils étaient privés);
- conservent leur statut protégé dans les classes dérivées.

Certains langages permettent également de réduire les droits d'accès au moment de la définition de la classe dérivée.

Au niveau des langages les plus connus :

Syntaxe de la dérivation et droits d'accès

C#

Les membres d'une classe disposent de trois statuts : private, protected et public. Le statut protected correspond au statut protégé qui a été présenté précédemment. C# dispose également d'unstatut internal, concernant les classes d'un même fichier. Les dérivations se notent ainsi:

```
class B : A
```

Java

Les membres d'une classe disposent également de trois statuts: private, protected et public. Mais, curieusement, le statut protected s'applique non seulement aux classes dérivées, mais aussi à toutes les classes d'un même ensemble nommé« paquetage». Cette confusion lui fait perdre de l'intérêt et, en pratique, il est peu utilisé. Les dérivations se notent ainsi :

```
class B extends A
```

PHP

Là encore, les membres d'une classe disposent des trois statuts : pri vat e, protect ed et public. Le statut protected ne concerne que les classes dérivées. Les dérivations se notent ainsi : `class Pointcol extends Point`

Python

Nous avons vu que les membres d'une classe ne pouvaient être que publics (par défaut) ou privés (pour ceux dont le nom commence par_). Tout naturellement, seuls les membres privés de la classe de base sont accessibles à la classe dérivée. Les dérivations se notent ainsi :

```
class Pointcol(Point) :
```

C++

Les membres d'une classe disposent de trois statuts: private, protected et public. Le statut protected correspond au statut protégé présenté précédemment. Mais les dérivations peuvent être également public, protected ou pri vate (ce qui fait 9 combinaisons possibles pour un membre donné). Le premier mode de dérivation (public) correspond à ce qui a été présenté dans ce chapitre; les autres vont dans le sens d'une réduction des droits d'accès de la classe dérivée.

La dérivation s'écrit comme dans cet exemple:

```
class Pointcol : public Point // ici Pointcol dérive «publiquement» de Point
```

Gestion des constructeurs

En Java et en Python, le constructeur de l'objet dérivé prend en charge l'ensemble de sa construction, mais il peut appeler le constructeur de la classe de base par `super (...)` en Java ou par `NomClasse-base.__init__(…)` en Python.

PHP fonctionne comme Java, mais l'appel du constructeur de la classe de base est « préfixé » par

`parent` :

```
public function __construct (x, y, c)
{ parent::__construct (x, y)    // appel du constructeur de la classe de base
  .....
}
```

En C++, la construction d'un objet d'une classe dérivée entraîne automatiquement le constructeur de la classe de base. S'il est nécessaire de lui communiquer des paramètres, il existe une syntaxe particulière de l'en-tête du constructeur de la classe dérivée comme dans :

```
Pointcol (int x, int y, int c) : Point (x, y)   // constructeur de Pointcol
{ ..... }
```

C# fonctionnne comme C++, en remplaçant simplement le nom du constructeur de la classe de base par

`base` :

```
Pointcol (int x, int y, int c) : base (x, y)   // constructeur de Pointcol
{ ..... }
```

Redéfinition de méthodes

Elle se déroule comme nous l'avons indiqué. Toutefois, en C#, il est nécessaire d'ajouter le terme new dans l'en-tête de la méthode pour indiquer au compilateur qu'il s'agit effectivement d'une « nouvelle définition ». Cette contrainte évite de redéfinir une méthode par erreur.

Exemple de codes :

En python :

```
class Point :
    def __init__ (self, x, y):
        self.__x = x
        self.__y = y

    def deplace(self, dx, dy):
        self.__x += dx
        self.__y += dy

    def affiche(self):
        print("Je suis en", self.__x, self.__y)

class Pointcol(Point) :

    def __init__(self,x,y, couleur):
        Point.__init__(self,x,y)
        self.__couleur = couleur

    def affiche(self):
        Point.affiche(self)
        print( "et ma couleur est :" , self.__couleur )

pc = Pointcol(3, 5, 3)
pc.affiche()
pc.deplace(1, -3)
pc.affiche()
```

Résultat :

```
Je suis en 3 5
et ma couleur est : 3
Je suis en 4 2
et ma couleur est : 3
```

En Java :

```java
class Point
{
    public Point (int x, int y) {
        this.x = x ; this.y = y ; }
    public void deplace (int dx, int dy) {
        x += dx ; y += dy ; }
    public void affiche () {
        System.out.println ("Je suis en " + x + " " + y) ; }
    private int x, y ;
}
class Pointcol extends Point
{
    public Pointcol (int x, int y, int couleur){
        super (x, y) ;  // obligatoirement comme première instruction this.couleur = couleur ;
        }
public void affiche () {
    super.affiche() ; System.out.println (" et ma couleur est : " + couleur) ; }
    private int couleur ; }

public class TstPcol
{
    public static void main (String args[])
    {
        Pointcol pc = new Pointcol(3, 5, (byte)3) ;
        pc.affiche() ; // ici, il s'agit de affiche de Pointcol
        pc.deplace (1, -3) ; // et deplace de Point
        pc.affiche() ;
    }
}
```

5. Le polymorphisme

Comme on a déjà pu le dire auparavant, le polymorphisme est un concept très puissant de la POO qui va compléter l'héritage. Il permet de manipuler des objets sans en connaître (tout à fait) le type. On pourra par exemple en construire un tableau d'objets (donc en fait de références à des objets), les uns étant de type *Point*, les autres étant de type *Pointcol* (dérivé de *Point*) et appeler la méthode affiche pour chacun des objets du tableau. Chaque objet réagira en fonction de son propre type.

Mais il ne s'agira pas de traiter ainsi n'importe quel objet. Nous montrerons que le polymorphisme exploite la relation «est» induite par l'héritage en appliquant la règle suivante :
un point coloré est aussi un point, on peut donc bien le traiter comme un point, la réciproque étant bien sûr fausse.

Débutons par présenter les deux notions de base du polymorphisme que sont la compatibilité par affectation et la ligature dynamique. Nous en examinerons ensuite les conséquences dans différentes situations. Nous verrons enfin quelles sont les limites de ce polymorphisme, ce qui nous amènera, au passage, à parler de valeurs de retour covariantes.

5.1 Les bases

Parlons d'abord de ce qui va constituer les bases du polymorphisme :
- la compatibilité par affectation,
- la ligature dynamique.

5.1.1 La compatibilité par affectation

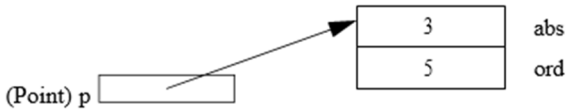
Considérons cette situation dans laquelle les classes *Point* et *Pointcol* sont censées disposer chacune d'une méthode affiche, ainsi que des constructeurs habituels (respectivement à deux et trois paramètres) :

```
classe Point{
        methode Point (entier x, entier y) {     }
        methode affiche (   }
}
classe Pointcol deriveDe Point{
        methode Pointcol(entier x, entier y, entier c) {}
        methode affiche {   }
}
```

Avec ces instructions :

```
Point p ;
p := Création Point (3, 5)
```

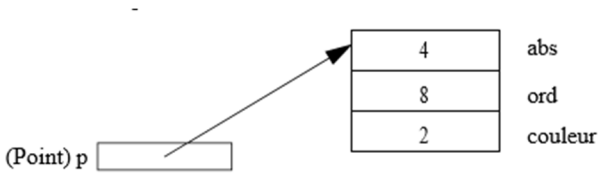
On peut donc schématiser la situation :



Mais, il est également permis d'affecter à *p*, non seulement la référence à un objet de type *Point*, mais aussi celle d'un objet de type *Pointcol* :

```
p = Création(Pointcol(4. 8. 2))   //  p de type Point contient la référence
                                  // à un objet de type Pointcol
```

La situation est:



En revanche, l'affection qui suit est totalement illégale :

```
Pointcol pc
…
pc := Creation (Point(…)) //interdit
```

On dira que :

On peut affecter à une variable objet non seulement la référence à un objet du type correspondant, mais aussi une référence à un objet d'un type dérivé.

On parle souvent de « compatibilité par affectation» entre un type classe et un type ascendant. Notez que les affectations légales sont celles qui exploitent la relation « est» induite par l'héritage.
On peut dire qu'un point coloré est un point, mais on ne peut pas dire qu'un point est un point coloré.
Voyons comment la ligature dynamique permet d'aller plus loin dans l'exploitation de cette relation, en autorisant d'appliquer à un point coloré les opérations applicables à un point.

5.1.2 La ligature dynamique

Considérons :

```
Point p
p := Création Point (3,5)
p.affiche                    // appelle la méthode affiche de la classe Point
p := Création Poincol ( 4,8,2)
p.affiche                    // appelle la méthode affiche de la classe Pointcol
```

Dans la dernière instruction, la variable *p* est de type *Point*, alors que l'objet référencé par *p* est de type *Pointcol*. L'instruction *p.affiche* appelle alors la méthode affiche de la classe *Pointcol*. Autrement dit, elle se fonde, non pas sur le type de la variable *p*, mais bel et bien sur le type effectif de l'objet référencé par *p* au moment de l'appel (ce type pouvant évoluer au fil de l'exécution). Ce choix d'une méthode au moment de l'exécution, basé sur la nature exacte de l'objet référencé porte généralement le nom de ligature dynamique (ou encore de liaison dynamique).

5.1.3 Résumé

On peut traduire le polymorphisme par :

- la compatibilité par affectation entre un type classe et un type ascendant;
- la ligature dynamique des méthodes.

Le polymorphisme permet d'obtenir un comportement adapté à chaque type d'objet, sans avoir besoin de tester sa nature de quelque façon que ce soit. La richesse de cette technique amène parfois à dire que l'instruction **si** (if) est à la POO ce que l'instruction *goto* est à la programmation structurée. Autrement dit, le bon usage de la POO (et du polymorphisme) permet parfois d'éviter des instructions de test, de même que le bon usage de la programmation structurée permettait d'éviter l'instruction *goto*. →Et oui le goto on l'évite en structurée et en POO on sera donc précis sur nos appels et méthodes.

Ce que vous connaissez sous le nom de « programmation procédurale», basée sur l'utilisation des procédures et des structures fondamentales (choix et répétitions), est aussi appelé « programmation structurée». Avant la généralisation de ce type de programmation, certains langages utilisaient des instructions de branchement conditionnel ou inconditionnel, désignés souvent par *goto*.

5.1.4 Attention à la gestion par valeur !

Le polymorphisme se fonde donc sur des références, en distinguant la nature de la référence d'une part, celle de l'objet référencé d'autre part.
Dans les langages gérant les objets par valeur, le polymorphisme n'existe donc pas vraiment.
Certes, on y trouve une compatibilité d'affectation, laquelle recopie simplement une partie d'un des objets dans l'autre :

```
Point p (3,8)
Pointcol pc (4, 9, 3)
…
p := pc  // autorisé : recopie seulement les attributs abs et ord de pc dans p
pc := p  // interdit
```

Quant à la ligature dynamique, elle n'a plus d'existence possible: quoi qu'il arrive, ici, *p* désignera toujours un objet de type *Point* et *pc* un objet de type *Pointcol*. On peut donc dire que *p.affiche* appellera toujours la méthode *affiche* de la classe *Point*.

Remarque : Lorsqu'un langage gère les objets par valeur, il ne s'agit généralement que d'une option par défaut qui se trouve complétée par un mécanisme de gestion par référence qu'il suffit alors de mettre convenablement en œuvre pour retrouver les possibilités du polymorphisme (exemple C++)

5.1.5 Exemple

```
Point p
Pointcol pc
p := Création p.affiche
Point (3, 5) // appelle affiche de Point
pc := Création Pointcol (4, 8, 2)
P := pc        // p de type Point, référence un objet de type Pointcol
p.affiche      // on appelle affiche de Pointcol
p := Création Point(5,7) //préférence a nouveau un objet de type Point
p.affiche      //on appelle affiche de Point

classe Point{
        methode Point(entier x, entier y) {
                abs := x ; ord := y}

        methode deplace(entier dx, entier dyl {
                abs := abs + dx ; ord :=ord + dy}

        methode affiche {
                ecrire «Je suis en », abs, « », ord }

        entier abs, ord
}

classs Pointcol deriveDe Point {

        methode Pointcol (entier x, entier y, entier c){
                base (x, y)
                couleur := c }

        methode affiche {
                base.affiche
                ecrire « et ma couleur est : », couleur }

        entier couleur }
```

Résultat :

Je suis en 3 5
Je suis en 4 8
et ma couleur est : 2
Je suis en 5 7

5.1.6 Exemple 2

Ici nous allons voir un exemple qui exploite les possibilités de polymorphisme pour créer un tableau « hétérogènes » d'objets.

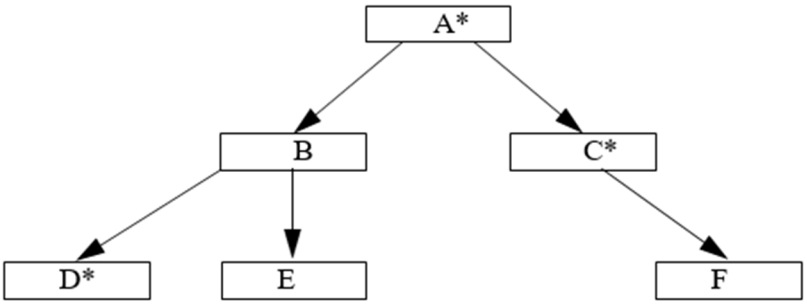
```
tableau Point [4] tabPts
entier i
tabPts [1] := Création Point (0, 2)
tabPts [2] := Création Pointcol (1, 5,3)
tabPts [3] := Création Pointcol(2,8, 9)
tabPts [4] := Création Point (1, 2)

répéter pour i := 1 à 4
        tabPts[iJ.affiche()

classe Point {
        methode Point (entier x, entier y) {
                abs := x ord := y }

        methode affiche {
                ecrire «Je suis en »,  abs,« », ord }

        entier abs, ord
}

classe Pointcol deriveDe Point {
        methode Pointcol (entier x, entier y, entier c) {
                base (x, y)
                couleur := c

        methode affiche {
                base.affiche
                ecrire « et ma couleur est : », couleur }

        entier couleur }
```

Résultat :

```
Je suis en 0 2
Je suis en 1 5
        et ma couleur est : 3
Je suis en 2 8
        et ma couleur est : 9
Je suis en 1 2
```

5.2 Cas avec plusieurs classes

Reprenons notre exemple général et plaçons un astérisque aux classes définissant ou redéfinissant la méthode *m* :



Avec ces déclarations :

```
A a ; B b ; C c ; D d ; E e ; F f ;
```

les affectations suivantes sont autorisés :

```
a := b ; a := c ; a := d ; a := e ; a := f ;
b := d ; b := e ;
c := f ;
```

En revanche, celles-ci ne le sont pas :

```
b := a ; // erreur : A ne descend pas de B
d := c ; // erreur : C ne descend pas de D
c := d ; // erreur : D ne descend pas de C
```

Voici quelques exemples précisant la méthode m appelée par l'instruction a.m, selon la nature de l'objet effectivement référencé par a (de type A):

- a référence un objet de type A : méthode m de A ;
- a référence un objet de type B : méthode m de A ;
- a référence un objet de type C : méthode m de C ;
- a référence un objet de type D : méthode m de D ;
- a référence un objet de type E : méthode m de A ;
- a référence un objet de type F : méthode m de C.

Voici quelques autres exemples correspondant cette fois à l'appel b.m :

- b référence un objet de type B : méthode m de A ;
- b référence un objet de type D : méthode m de D ;
- b référence un objet de type E : méthode m de A.

La variable contenant la référence à un objet de l'une des classes est toujours de type A, alors que, auparavant, elle était du type de l'objet référencé.

5.3 Situation spécifique au polymorphisme

Dans les exemples précédents, les méthodes affiche de *Point* et de *Pointcol* se contentaient d'afficher les valeurs des attributs concernés, sans préciser la nature exacte de l'objet. Nous pourrions par exemple souhaiter que l'information se présente ainsi pour un objet de type *Point* :

```
Je suis un point
        Mes coordonnées sont : 0 2
```

et ainsi pour un objet de type *Pointcol* :

```
Je suis un point colore de couleur 3
        Mes coordonnées sont : 1 5
```

On peut considérer que l'information affichée par chaque classe se décompose en deux parties :

- une première partie spécifique à la classe dérivée (ici *Pointcol*),
- une seconde partie commune correspondant à la partie héritée de *Point* : les valeurs des coordonnées.

D'une manière générale, ce point de vue pourrait s'appliquer à toute classe descendant de *Point*. Dans ces conditions, plutôt que de laisser chaque classe descendant de *Point* redéfinir la méthode affiche, on peut définir la méthode affiche de la classe *Point* de manière qu'elle:

- affiche les coordonnées (action commune à toutes les classes) ;

- fasse appel à une autre méthode (nommée par exemple identifie) ayant pour vocation d'afficher les informations spécifiques à chaque objet. Ce faisant, nous supposons que chaque descendante de Point redéfinira identifie de façon appropriée (mais elle n'aura plus à prendre en charge l'affichage des coordonnées).

Cette démarche nous conduit à définir la classe *Point* de la façon suivante :

```
class Point {
        methode Point(int x, int y){
                abs := x ord := y }

        methode affiche {
                identifie
                ecrire« Mes coordonnees sont : » x, «»,y

        methode identifie {
                ecrire «Je suis un point» }
        entier abs, ord }
```

Dérivons en une classe *Pointcol* en redéfinissant de façon appropriée la méthode *identifie* :

```
classe Pointcol deriveDe Point {

        methode Pointcol(entier x, entier y, entier c) {
                base (x,y) couleur := c }

        methode identifie {
                ecrire «Je suis un point coloré de couleur», couleur }

        entier couleur }
```

On peut maintenant considerer la suite d'instructions suivantes :

```
Pointcol pc
pc := Création Pointcol (8,6,2)
pc.affiche
```

L'instruction *pc.affiche* entraîne l'appel de la méthode *affiche* de la classe *Point* (puisque cette méthode n'est pas redéfinie dans *Pointcol*).
Mais dans la méthode *affiche* de *Point*, l'instruction identifie appelle la méthode *identifie* de la classe correspondant à l'objet effectivement concerné (autrement dit, celui de référence courant). Comme ici, il s'agit d'un objet de type *Pointcol*, il y aura bien appel de la méthode *identifie* de *Pointcol*.
On pourrait tirer la même conclusion si on avait utilisé la suite d'instructions suivante :

```
Point p
p = Creation Pointcol(8, 6, 2)
p.affiche
```

Ici aussi, c'est le type d'objet référencé par *p* qui intervient dans le choix de la méthode *affiche*.

Voici un exemple complet d'un petit programme utilisant les classes *Point* et *Pointcol* avec un tableau hétérogène :

```
tableau Point [4] tabPts
tabPts [1] := Création Point (0, 2)
tabPts [2] := Création Pointcol (1, 5, 3)
tabPts [3] := Création Pointcol (2, 8,9)
tabPts [4] := Création Point (1, 2)
répéter pour i := 1 à 4
        tabPts[i].affiche()

classe Point {
        methode Point (entier x, entier y) {
                abs := x ; ord := y }
        methode affiche () {
                identifie()
        ecrire « Mes coordonnées sont : », abs, « », ord }

        methode identifie () {
                ecrire «Je suis un point » }
        entier abs, ord }

classe Pointcol deriveDe Point {
        methode Pointcol (entier x, entier y, entier c) {
                base (x, y) ; couleur := c }

        methode identifie {
        ecrire «Je suis un point coloré de couleur », couleur }
        entier couleur }
```

Résultat :

```
Je suis un point
        Mes coordonnées sont : 0 2
Je suis un point coloré de couleur 3
        Mes coordonnées sont : 1 5
Je suis un point coloré de couleur 9
        Mes coordonnées sont : 2 8
Je suis un point
        Mes coordonnées sont : 1 2
```

Cet exemple peut se généraliser à plusieurs classes dérivées de *Point*. En revanche, les choses seraient moins claires en ce qui concerne la généralisation à des classes dérivées de *Pointcol*. Quoi qu'il en soit, il faut surtout considérer que l'on a affaire ici à un « exemple d'école» présentant un appel d'une méthode qui n'est pas encore définie. Nous verrons d'ailleurs que les classes abstraites et les interfaces permettent de formaliser cet aspect.

5.4 Les limites du polymorphisme et de l'héritage

Le polymorphisme se base donc sur la redéfinition des méthodes. Par exemple, *p.affiche* appelle la méthode *affiche* de *Point* ou la méthode *affiche* de *Pointcol*. Cette redéfinition va donc utiliser la signature de la méthode concernée et, en particulier, dans certains langages, elle peut interférer avec la notion de surdéfinition et aboutir ainsi à des situations parfois très complexes (dont on a toutefois rarement besoin pour faire du « bon polymorphisme»!).
Sans entrer dans ces détails, nous allons quand même attirer votre attention sur deux points :

- les limitations du polymorphisme: nous vous présenterons une situation dans laquelle on aurait aimé qu'il s'applique

- l'existence de ce que l'on nomme dans certains langages des « valeurs de retour covariantes».

5.4.1 Les limites

Le polymorphisme se base en même temps sur le type de la référence concernée et sur celui de l'objet référencé.
Revenons sur le rôle exact de chacun de ces éléments:

- le type de la référence définit la signature de la méthode à appeler et le type de son résultat ;

- le type de l'objet référencé provoque la recherche (en remontant éventuellement dans la « hiérarchie d'héritage») d'une méthode ayant la signature et le type de résultat voulus.

Cela peut vous paraître quasi évident maintenant. Pourtant, considérez la situation suivante dans laquelle :

- la classe *Point* dispose d'une méthode identique fournissant la valeur vraie lorsque le point fourni en paramètre a les mêmes coordonnées que le point courant :

```
Point pl, p2
…
pl.identique(p2) // vrai si pl et p2 ont mêmes coordonnées
```

- la classe *Pointcol*, dérivée de *Point*, définit une autre méthode nommée *identique* pour prendre en compte non seulement l'égalité des oordonnées, mais aussi celle de la couleur:

```
Pointcol pc1, pc2
…
pc1.identique(pc2) // vrai si pc1 et pc2 ont mêmes coordonnées et même couleur
```

Soient les déclarations suivantes:

```
Point pl. p2
P1 := Création Pointcol C(1,2, 5)
p2 := Création Pointcol (1, 2, 8)
```

Considérons cette expression :

```
p1.identique(p2) // est vrai
```

Elle a pour valeur vrai alors que nos deux points colorés n'ont pas la même couleur. L'explication réside tout simplement dans la bonne application des règles relatives au polymorphisme. En effet, à la rencontre de cette expression *p1.identique (p2)*, on se fonde sur le type de pl pour en déduire que l'en-tête de la méthode *identique* à appeler est de la forme *booléenne identique* (*Point*). La ligature dynamique tient compte du type de l'objet réellement référencé par *p1* (ici *Pointcol*) pour définir la classe à partir de laquelle se fera la recherche de la méthode voulue. Mais comme dans *Pointcol*, la méthode identique n'a pas la signature voulue, on poursuit la recherche dans les classes ascendantes et, finalement, on utilise la méthode *identifie* de *Point*. D'où le résultat constaté. Bien entendu, ici, la méthode *identifie* de *Pointcol* n'était pas une redéfinition de la méhtode *identifie* de *Point*.

5.4.2 Les retour covariants

Ci-dessus, nous aurions souhaité obtenir un certain comportement qui, malheureusement, n'entrait pas dans le cadre du polymorphisme. Considérons une classe *Point*, dotée d'une méthode de copie nommée *clone*, fournissant en résultat une référence sur un objet de type *Point* :

```
classe Point {
        Point methode clone {
                Point p
                …
                retourne p }
        … }
```

Et maintenant notre classe *Pointcol*, dotée d'une méthode similaire :

```
classe Pointcol deriveDe Point {
        Pointcol methode clone{
                Pointcol pc
                …
                retourne pc }
        … }
```

Imaginons la situation de polymorphisme habituelle :

Point p
Pointcol pc
…
p := Création Pointcol(...)

On va maintenant regarder ce qui se passe si on s'intéresse à l'appel *p.clone*. La référence *p* est de type *Point* et on trouve dans *Point* une méthode *clone* fournissant un résultat de type *Point*.
C'est justement cette signature (vide) et ce type de résultat (*Point*) que l'on se base pour explorer la hiérachie d'héritage, à partir de *Point* pour effectuer la ligature dynamique voulue, basée ici sur le type de l'objet référencé par *p*, c'est-à-dire plus clairement Pointcol.
On trouve bien dans *Pointcol* une méthode *clone* une bonne signature, mais son résultat n'a pas le type voulu.
Les règles strictes du polymorphisme telles que nous les avons exposées conduiraient, ici encore, à appeler la méthode *clone* de *Point*.
Cependant, on peut se rendre facilement compte qu'il s'agit d'un cas particulier qui implique que:

- la méthode *clone* de *Point*, appliquée à un objet de type *Point*, renvoyait une référence à un *Point*;
- la méthode *clone* de *Pointcol*, appliquée à un objet de type *Pointcol* renvoie une référence à un *Pointcol* ...

Cette situation paraît presque naturelle, au point que beaucoup de langages autorisent une dérogation à la règle concernant le type du résultat, connue souvent sous le nom de « valeurs de retour covariantes». Elle consiste à considérer que, dans la recherche de la ligature dynamique, on peut tolérer un écart sur le type du résultat lorsque celui-ci est du type de la classe de la référence concernée.

En résumé:

- soit *p* est une référence à un objet de type *T* et *m* une méthode de *T* renvoyant une référence de type *T*;

- soit *T'* une classe dérivée de *T*, redéfinissant la méthode *m*, de manière à ce qu'elle renvoie une référence de type *T '* .

Dans ces conditions, si à un moment donnée *p* contient la référence d'un objet de type *T'*, l'appel *p.m* peut être résolu dynamiquement par l'appel de la méthode *m* de *T'*.

5.5 Coté langage

Java, PHP et Python

En Java, PHP et Python, la gestion des objets est réalisée par référence et le polymorphisme est« natif». Autrement dit, on se trouve dans la situation décrite ici.

C#

En C#, où les objets sont gérés par référence, on retrouve bien la compatibilité d'affectation nécessaire au polymorphisme. En revanche, il faut préciser les méthodes qu'on souhaite voir soumises à la ligature dynamique, à l'aide du mot virtual (on parle alors de« méthodes virtuelles»). De plus, les méthodes virtuelles redéfinies doivent être qualifiées par override (et non plus par new). On peut donc faire cohabiter au sein d'une même classe des méthodes à ligature dynamique avec des méthodes à ligature statique.

C++

En C++, lorsque, comme c'est le cas par défaut, les objets sont gérés par valeur, il n'y a pas de polymorphisme véritable, mais simplement une compatibilité d'affectation (de peu d'intérêt), comme dans:

```
Point p (3, 8) ;
Pointcol  pc (4, 9, 3) ;

p = pc ;   // autorisé : recopie seulement les attributs de
pc dans p pc := p   // interdit
```

Pour pouvoir appliquer le polymorphisme en C++, il faut:

- manipuler les objets par pointeur, ce qui fournit la compatibilité d'affectation nécessaire ;
- déclarer explicitement à l'aide du mot virtual, les méthodes qu'on souhaite voir soumises à la ligature dynamique.

Comme en C#, une même classe peut mêler des méthodes soumises à la ligature dynamique avec des méthodes soumises à une ligature statique.

On notera bien que les méthodes virtuelles sont sans effet sur les objets gérés par valeur.

En Java :

```java
class Point
{
        public Point (int x, int y){
            abs = x ; ord = y ; }

    public void affiche () {
        identifie() ;
        System.out.println (" Mes coordonnees sont : " + abs + " " + ord) ; }

    public void identifie (){
        System.out.println ("Je suis un point ") ;}

    private int abs, ord ; }

class Pointcol extends Point {
    public Pointcol (int x, int y, int c) {
            super (x, y) ;
            couleur = c ;}
    public void identifie () {
        System.out.println ("Je suis un point colore de couleur " + couleur) ; }

    private int couleur ; }

public class TabHeter {
    public static void main (String args[]) {
        Point [] tabPts = new Point [4] ;
        tabPts [0] = new Point (0, 2) ;
        tabPts [1] = new Pointcol (1, 5, (byte)3) ;
        tabPts [2] = new Pointcol (2, 8, (byte)9) ;
        tabPts [3] = new Point (1, 2) ;

        for (int i=0 ; i< tabPts.length ; i++) {
            tabPts[i].affiche() ; }

        }
    }
```

En python :

```python
class Point :
    def __init__ (self, x, y) :
        self.__abs = x ; self.__ord = y

    def affiche (self) :
        self.identifie()
        print ( "Mes coordonnees sont ", self.__abs, " ", self.__ord)
    def identifie (self) :
        print ("Je suis un point ")
class Pointcol (Point) :
    def __init__ (self, x,y, c) :
        Point.__init__ (self, x, y)
        self.__couleur = c

    def identifie (self) :
        print ("Je suis un point colore de couleur", self.__couleur)

tabPts = [Point (0, 2), Pointcol (0, 5,3), Pointcol(2, 8, 9),Point (1, 2)]
for i in range(len(tabPts)) :
    tabPts[i].affiche()
```

6. Abstraits, interfaces ou héritages multiples ?

6.1 Les classes abstraites ou méthodes retardées

6.1.1 Les classes abstraites

Quand on commence à paramétré de l'héritage ou du polymorphisme, on peut être amené à créer une classe simplement destinée à servir de classe de base pour d'autres classes, et en aucun cas à donner naissance à des objets. Bon nombre de langages permettent alors de «formaliser» cela dans la notion de classe abstraite. Nous conviendrons qu'une telle classe se déclare ainsi :

```
classe abstraite A
{       // attributs et méthodes }
```

Nous n'avons donc plus la possibilité d'instancier des objets de type A. Une expression telle que:

```
Création A(...)       // erreur
```

sera rejetée par votre compilateur/interpréteur.

En revanche, il reste possible de déclarer des variables de type A (du moins dans les langages gérant les objets par référence (dans les autres, la seule déclaration entraînerait une instanciation interdite...)). À cette variable, on pourra par exemple affecter la référence d'un objet d'une classe dérivée de A :

```
classe A1 deriveDe A{
        .....
        A a
        a := Création A(...) // interdit
        a := Création A1(...) // OK    }
```

6.1.2 Les méthodes retardées

Les méthodes retardées sont aussi appelées méthodes abstraites. Jusqu'ici, les classes abstraites ne disposaient que de méthodes classiques et bien ce n'est pas tout. Il s'agit de méthodes dont on ne fournit que la signature et le type de résultat. Comme elles ne sont pas définies dans la classe abstraite, elles devront l'être dans les classes dérivées pour que celle-ci puisse instancier l'objet.

```
classe abstraite B {
        entier methode f (réel x) {
                . . . . . }       / / définition d'une méthode «usuelle»
        caractère methode retardee g (entier) // méthode retardée : on ne fournit
                                              // que sa signature et le type du résultat }
```

On pourra alors instancier des objets de type B1 :

```
B b
B1 b1
b1 := Création B1(...)      // OK
b := Création B1(...)       // OK
b := Création B(...)        // erreur
```

Bien entendu, il reste possible de définir comme abstraite une classe dérivée d'une classe abstraite. Dans ce cas, la redéfinition des méthodes retardées n'est pas obligatoire. Elle devra simplement être faite dans toute classe dérivée ultérieure qu'on souhaite pouvoir instancier :

```
classe abstraite B2 deriveDe B
{ ..... // ici, on n'est pas obligé de définir la méthode g }
```

6.1.3 C'est joli tout ça mais ça sert à quoi ?

Faire appel aux classes abstraites et aux méthodes retardées facilite largement la conception des logiciels. En effet, on peut placer dans une classe abstraite toutes les fonctionnalités dont on souhaite disposer pour toutes ses descendantes :

- soit sous forme d'une implémentation complète de méthodes (non retardées) et d'attributs (privés ou non) lorsqu'ils sont communs à toutes ses descendantes,
- soit sous forme d'interfaces (signature + type du résultat) de méthodes retardées dont on est alors certain qu'elles existeront dans toute classe dérivée instanciable.

C'est cette certitude de la présence de certaines méthodes qui permet d'exploiter le polymorphisme, et ce dès la conception de la classe abstraite, alors même qu'aucune classe dérivée n'a peut-être encore été créée. Notamment, on peut très bien écrire des canevas recourant à des méthodes abstraites. Par exemple, si vous avez défini :

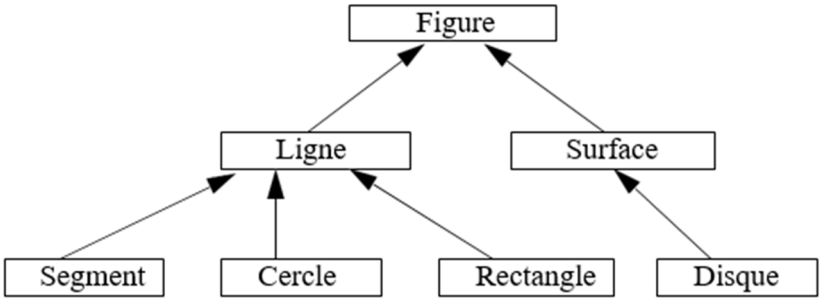
```
classe abstraite X {
        methode retardee f()      // ici, f n'est pas encore définie
        ..... }
```

vous pourrez écrire une méthode (d'une classe quelconque), disposant d'un paramètre de type X, telle que :

```
méthode algo (X x)// en paramètre, x de type X {
.....
x.f()   // appel correct car on est sûr que tout objet
.....   // d'une classe dérivée de X disposera bien d'une méthode f }
```

Bien entendu, tout appel de la méthode algo devra utiliser, en paramètre effectif, une référence à un objet d'un type dérivé de X.

Ces possibilités pourront par exemple être employées dans la réalisation d'un programme manipulant des figures géométriques, se basant sur une « hiérarchie d'héritage » ressemblant à ceci :



Les classes *Ligne* et *Surface* pourraient éventuellement rester abstraites, tout en introduisant de nouvelles méthodes retardées, par exemple *styleTrait* pour *Ligne* ou *motif-Peinture* pour *Surface*. Les classes dérivées de *Ligne* devraient, quant à elles, redéfinir convenablement les méthodes *dessine* et *styleTrait* ; celles dérivées de *Surface* devraient redéfinir *dessine* et *motifPeinture*.

6.1.4 Exemple

Ici un exemple d'une classe abstraite nommée *Affichable* qui va être dotée d'une méthode *affiche*. Également deux classes *Entier* et *Réel* qui dérivent de cette classe. Notre programme utilisera un tableau hétérogène d'objets de type *Affichable* qu'il va pouvoir remplir en instanciant des objets de type *Entier* et *Réel*.

```
classe abstraite Affichable {
        methode retardée affiche }

classe Entier deriveDe Affichable {
         methode Entier(entier n) {
                valeur := n }

        methode affiche {
                ecrire «Je suis un entier de valeur», valeur }
        entier valeur

classe Reel deriveDe Affichable {
        methode Reel (reel x) {
                valeur := x }
        methode affiche {
                ecrire "Je suis un réel de valeur ", valeur }
        reel valeur}
```

```
entier i
tableau Affichable [3] tab
tab [1] := Creation Entier (25)
tab [2] := Creation Réel (1.25)
tab [3] := Creation Entier (42)
repeter pour i := 1 a 3
        tab[i].affiche
```

```
Je suis un entier de valeur 25
Je suis un flottant de valeur 1.25
Je suis un entier de valeur 42
```

Remarque :

Nous avons vu qu'une classe abstraite pouvait comporter, en plus de méthodes retardées, des attributs ou des implémentations effectives de méthodes, ce qui n'est pas le cas de notre précédent exemple (où de surcroît, il n'y a qu'une seule méthode). Nous verrons dans le prochain paragraphe que lorsqu'une classe abstraite ne contient que des méthodes retardées, une « interface » peut jouer un rôle voisin (mais non identique) et nous montrerons comment transformer dans ce sens l'exemple précédent.

6.2 Les Interfaces en POO

Nous avons déjà vu que, dans la réalisation d'une classe, on pouvait distinguer théoriquement son «interface» de son «implémentation ». Rappelons que l'interface correspond à l'ensemble des signatures des méthodes et leur éventuel résultat.
Bon nombre de langages vont permettre de« formaliser» cette notion d'interface en offrant :

- un mécanisme de définition d'interfaces;
- un mécanisme obligeant une classe à implémenter une interface donnée.

Nous allons voir que, dans ce cas, qu'il s'agit plus que d'une simple aide à la programmation car :

- une même classe pourra implémenter plusieurs interfaces
- les interfaces pourront jouer un rôle voisin de celui des classes abstraites et participer au polymorphisme (on parler alors de« polymorphisme d'interfaces»).

6.2.1 Définition

Une interface ne contient que des signatures de méthodes (et le type de leur éventuel résultat). Nous conviendrons que cette notation :

```
interface I {
        methode f (entier)
        reel methode g (entier, réel) }
```

Va définir une interface nommée *I*, qui va disposer de deux méthodes *f* et *g* ayant une signature identique.
On remarque que contrairement à une classe abstraite, une interface ne va contenir ni définition de méthodes, ni attributs. Elle ne va contenir que des méthodes retardées de sorte que ce point ne soit pas nécessaire à mentionner.

Attention de bien faire la différence, une interface ne s'utilise pas comme une classe et n'est pas une simple classe abstraite ne comportant que des méthodes abstraites.

6.2.2 L'implémentation d'une interface

Lors de la définition d'une classe (ici *A*), nous convenons que nous pouvons préciser qu'elle « implémente » une interface donnée (ici *I*), de cette façon :

```
classe A implemente I
{ ..... }
```

Cela signifie qu'on s'engage à ce que *A* définisse convenablement les méthodes présentes dans *I*. On a donc la garantie (grâce aux vérifications opérées par le traducteur) que *A* dispose des méthodes *f* et *g*, avec les bonnes signatures et le bon type de résultat.
On notera bien que le fait qu'une classe implémente une interface donnée ne l'empêche nullement de disposer d'autres méthodes. Autrement dit, *A* n'est pas nécessairement une implémentation de *I*. On sait simplement que *A* dispose au minimum des méthodes prévues dans l'interface *I*.
D'ailleurs, une même classe peut implémenter plusieurs interfaces, comme dans cet exemple:

```
interface I1 {
        methode f }

interface I2 {
        entier methode g (caractère)
        methode h (entier) }

classe A implemente I1, I2
{       // A doit définir au moins les méthodes f det I1, g et h de I2 }
```

6.2.3 Les variables qui ont un type interface et le polymorphisme

Comme pour les variables du type d'une classe abstraite, on peut définir des variables d'un type interface :

```
interface I { …}
.....
I i      // i est une référence à un objet quelconque dont la classe implémente
         // l'interface I
```

Il est évident qu'on ne pourra pas affecter à *i* une référence à quelque chose de type *I* car on ne peut pas instancier une interface (pas plus qu'on ne pouvait instancier une classe abstraite). En revanche, on pourra affecter à *i* n'importe quelle référence à un objet d'une classe implémentant l'interface *I* :

```
classe A implemente I {   }
.....
i := Creation A(...) // OK
```

De plus, à travers *i*, on pourra manipuler des objets de classes quelconques, non obligatoirement liées par héritage, pour peu que ces classes implémentent l'interface *I*. Bien entendu, ces « manipulations » devront concerner exclusivement des méthodes prévues dans *I* :

```
interface I {
        methode f  }

classe A implemente I {
        methode f { ..... }  // on implémente I
        methode g { ..... } // une autre méthode indépendante de I }
I i
i := Création A(...) // OK
i.f      // OK : f appartient à I
i.g      // erreur : f n'appartient pas à I
```

On retrouve là en fait les mêmes restrictions que dans le polymorphisme basé sur l'héritage.
Par ailleurs, comme on l'a déjà dit, toute classe peut implémenter plusieurs interfaces et donc, suivant les cas, « profiter » du polymorphisme de l'une ou de l'autre. Voyez cet exemple :

```
Interface I1 {…}
Interface I2 {…}
classe A implémente I1    {…}
classe B implémente I2    {…}
classe C implémente I1, I2 {…}
```

```
I1 i1   // i1 pourra «désigner» des objets de type A ou C auquel on applique
        // les méthodes de I1

I2 i2   // i2 pourra «désigner» des objets de type B ou C auquel on applique
        // les méthodes de I2
```

6.2.4 Exemple complet

Voici un exemple simple illustrant le polymorphisme d'interfaces. Une interface Affichable comporte une méthode affiche. Deux classes nommées Entier et Réel implémentent cette interface (aucun lien d'héritage n'existe entre elles). On crée un tableau hétérogène de références de type Affichable qu'on remplit en instanciant des objets de type Entier ou Réel. En fait, il s'agit d'une transposition du programme 6.1.4 qui utilisait des classes abstraites (transposition qui n'est possible que parce que notre classe abstraite ne contenait rien d'autre que des méthodes retardées).

```
interface Affichable {
        methode affiche }

classe Entier implemente Affichable {
        methode Entier (entier n) {
                valeur := n }
        methode affiche {
                ecrire «Je suis un entier de valeur », valeur }

        entier valeur }

classe Reel implemente Affichable {
        methode Reel (reel x) {
                 valeur := x }
        methode affiche {
                ecrire «Je suis un réel de valeur », valeur }
        reel valeur }

tableau Affichable [3] tab entier i
tab [1] := Creation Entier (25)
tab [2] := Creation Réel (1.25)
tab [3] := Creation Entier (42)

repeter pour i := 1 à 3
        tab[i].affiche
```

Résultat :

Je suis un entier de valeur 25
Je suis un réel de valeur 1.25
Je suis un entier de valeur 42

6.2.5 Les interfaces et les classes dérivés

La notion d'interface est totalement indépendante de l'héritage. Une classe dérivée peut, à son tour implémenter une interface ou plusieurs :

```
interface I {
        methode f (entier)
                methode g }
classe A {    }
classe B deriveDe A implemente I {
        // les méthodes f et g doivent soit être déjà définies dans A
        // soit définies dans B }
```

On peut même rencontrer cette situation, dans laquelle on est certain que B implémente les deux interfaces I1 et I2 :

```
interface I1 {…}
interface I2 {…}
classe A implemente I1 {…}
classe B deriveDe A implemente I2 { .....}    // ici, B implémente I1 (par
                                              // héritage de A), et I2
```

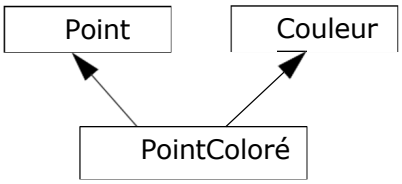
6.3 l'héritage multiple

Jusqu'ici, nous n'avons considéré que ce que l'on nomme l'héritage simple dans lequel une classe dérive d'une seule classe de base. Certains langages offrent la possibilité dite d'héritage multiple, dans laquelle un classe peut hériter simultanément de plusieurs classes de base. L'héritage multiple reste cependant peu usité car :

- peu de langages en disposent;
- il entraîne des difficultés de conception des logiciels. Il est, en effet, plus facile de

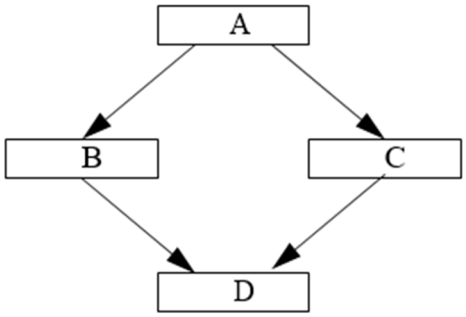
«structurer» un ensemble de classes selon un« arbre» (cas de l'héritage simple) que selon un« graphe orienté sans circuit » (cas de l'héritage multiple).

Ici, nous nous contenterons de le présenter succinctement.
L'héritage multiple permet donc de gérer des dépendances telles que :



Ici, la classe *PointColoré* hérite à la fois de la classe *Point* et de la classe *Couleur*.

Bien entendu, il faut prévoir un mécanisme d'appel des constructeurs, ce qui ne présente pas de difficultés. En revanche, certaines situations peuvent poser problème. Considérons par exemple cette situation :



Ici, on voit que, en quelque sorte, *D* hérite deux fois de *A*. Dans ces conditions, les méthodes et les attributs de *A* apparaissent deux foix dans *D*. En ce qui concerne les méthodes, cela est manifestement inutile, mais sans importance puisqu'elles ne sont pas dupliquées. En revanche, en ce qui concerne les attributs, il faudra disposer d'un mécanisme permettant de choisir de les dupliquer ou non. Dans le cas où on les duplique, il faudra être capable d'identifier ceux obtenus par l'intermédiaire de *B* de ceux obtenus par l'intermédiaire de *C*. Conceptuellement, on voit que les choses ne sont pas particulièrement claires.

6.4 Coté langage

Classes abstraites et méthodes retardées

Java, C# et PHP disposent des notions de classes abstraites et de méthodes retardées (qu'ils nomment toutefois abstraites), telles que nous les avons exposées ici.

En revanche, C++ ne dispose que de la notion de méthode virtuelle pure. Il s'agit d'une méthode déclarée virtuelle, dont le corps est remplacé par la mention «=0», comme dans:

```
class A
{ public
    virtual void f () = 0       // méthode virtuelle pure
```

Dans ce cas, il est impossible d'instancier des objets de type A qui se comporte donc comme une classe abstraite. Dans une classe dérivée de

A, la méthode f devra, soit être définie, soit déclarée à nouveau virtuelle pure.

En Python, ces notions n'existent pas de façon native. Dans les versions les plus récentes, elles peuvent être mise en place par l'intermédiaire d'un «module complémentaire»(abc).

Interfaces

Java, C# et PHP disposent de la notion d'interface. Java et PHP utilisent le mot i mp1ements (là où nous avons utilisé i mp1émente). En revanche, C# utilise la même notation que pour l'héritage: si la classe A implémente l'interface I, on écrira :

```
class A : I
```

C++ et Python ne disposent pas de la notion d'interface. En revanche, ils autorisent l'héritage multiple. Lorsqu'une classe hérite plusieurs fois d'une autre, les attributs sont dupliqués. En C++, on peut éviter cette duplication en prévoyant des« dérivations virtuelles».

Exemples langages

Java

On dispose à la fois de la notion de classe abstraite (mot abstract) et de celle de méthode retardée (même mot abstract).

```
abstract class Affichable {
    abstract public void affiche() ; }

class Entier extends Affichable {
    public Entier (int n) {
        valeur = n ; }

    public void affiche() {
        System.out.println ("Je suis un entier de valeur " + valeur) ; }

 private int valeur ; }

class Reel extends Affichable {
    public Reel (float x) {
        valeur = x ; }

    public void affiche() {
        System.out.println ("Je suis un réel de valeur " + valeur) ; }

 private float valeur ; }

public class TableauHeter {
    public static void main (String[] args) {
        Affichable [] tab ;
        tab = new Affichable [3] ;
        tab [0] = new Entier (25) ;
        tab [1] = new Reel (1.25f) ;
        tab [2] = new Entier (42) ;
        int i ;
        for (i=0 ; i<3 ; i++)
            tab[i].affiche() ; }
    }
```

Interfaces

Voici, écrit en Java (attention que C++ ne dispose pas de la notion d'interface). En Java,C# et PHP, une interface se déclare à l'aide du mot `interface`.

```
interface Affichable {
    void affiche() ;}

class Entier implements Affichable {
    public Entier (int n) {
        valeur= n ; }
    public void affiche(){
        System.out.println("Je suis un entier de valeur " + valeur); }
    private int valeur ;}

class Reel implements Affichable {
    public Reel (float x) {
        valeur= x ; }

    public void affiche(){
        System.out.println("Je suis un réel de valeur"+ valeur);}

    private float valeur; }

public class Tabhet4 {
    public static void main (String[] args) {
        Affichable[] tab ;
        tab = new Affichable [3] ;
        tab [0] = new Entier (25) ;
        tab [1] = new Reel (1.25f) ;
        tab [2] = new Entier (42);
        int i ;
        for (i=0 ; i<3 ; i++)
            tab[i].affiche() ; }}
```