CHAPITRE 5 :

LE POLYMORPHISME

Objectifs spécifiques

1. comprendre Le concept de polymorphisme à travers des exemples
2. Comprendre la conversion des arguments en liaison avec le polymorphisme
3. Comprendre la conversion explicite des références en rapport avec le polymorphisme

Eléments de contenu

I. Concept de polymorphisme

II. Exemple et interprétation

III. Conversion des arguments effectifs

IV. Les conversion explicites des références

Volume Horaire :

Cours : 3 heures

Travaux Pratiques : 3 heures

5.1 Concept de polymorphisme

✈ Le polymorphisme est un concept extrêmement puissant en POO, il permet de manipuler des objets sans en connaître tout à fait le type tout en se basant sur la relation d'héritage.

✈ Le polymorphisme en Java, se traduit par :

- La compatibilité entre un type dérivée et un type ascendant.
- La ligature dynamique des méthodes, dans le sens où il permet d'obtenir le comportement adapté à chaque type d'objet, sans avoir besoin de tester sa nature de quelque façon que ce soit.

✈ Le polymorphisme se base sur cette affirmation : un objet a comme type non seulement sa classe mais aussi n'importe quelle classe dérivée.

5.2 Exemples et interprétations

Exemple 1

```
//Classe de base
class graphique {
```

```
private int x, y;
public graphique ( int x, int y) { this.x = x ; this.y = y ;}
public void identifie () {
System.out.println (« Je suis une forme géometrique ») ;}
public void affiche () {
this.identifie() ;
System.out.println («  Le centre de l'objet se trouve dans :   » + x
+ «  et   » + y) ;}
double surface () {return (0) ;}
} // fin de la classe graphique
//Classe dérivée1
class Cercle extends graphique {
private double rayon =1 ;
public Cercle ( int x, int y, double r) {super(x,y) ;
rayon = r ;}
public void identifie () {
System.out.println (« Je suis un cercle  ») ;}
double surface ()
{return ( rayon * 2* 3.14) ;} }
//Classe dérivée2
class Rectangle extends graphique {
private int larg, longueur ;
Rectangle ( int x, int y, int l1, int l2)
{super (x,y) ;
longueur = l1 ; larg = l2 ;}
double surface () {return (longueur*largeur) ;}
public void identifie() {
System.out.println(« Je suis un rectangle ») ;}
}// fin de la classe Rectangle
//Classe de test
class test_poly {
public static void main (String [] args) {
graphique g = new graphique (3,7);
g.identifie ();
g= new Cercle (4,8,10) ;
// compatibilité entre le type de la  classe et de la classe dérivéé
g.identifie() ;
```

```
g= new Rectangle (7,9,10,3) ;
// compatibilité entre le type de la  classe et de la classe dérivéé
g.identifie () ;
}
}
```

Résultat de l'exécution

```
Je suis une forme géométrique
Je suis un cercle
Je suis un rectangle
```

Interprétation

Le même identificateur « g » est initialisé dans la 1ère instruction avec une référence de type « graphique », puis on a changé la référence de cette variable dans l'instruction3 en lui affectant une référence de type « Cercle », puis dans l'instruction 5, on a changé sa référence avec une référence de classe dérivée « Rectangle ».

Ces affectations confirment la compatibilité entre un type d'une classe de base et la référence d'une classe dérivée.

L'autre point qui semble plus important c'est la ligature dynamique des méthodes, dans le sens où le résultat de la méthode « identifie » a changé dans chaque appel selon le type effectif de la variable « g ».

Exemple 2 : Tableau des objets

Dans cet exemple, on va exploiter les possibilités de polymorphisme pour créer un tableau hétérogène d'objets, c'est à dires dans lequel les éléments peuvent être de types différents.

```
class test_poly2 {
public static void main (String [] args) {
graphique [] tab = new graphique [6];
tab[0] = new graphique (3,2);
tab[1] = new Cercle (10,7,3) ;
tab [2] = new Rectangle (4,7,8,6) ;
tab [3] = new graphique (8,10);
tab[4] = new Cercle (8,5,3) ;
tab[5 = new Rectangle (10,17,3,8) ;
for (i=0 ; i <=5 ; i++) {tab[i].affiche();}
}
}
```

Résultat de l'exécution

```
Je suis une forme géométrique   Le centre de l'objet se trouve dans :   3  et   2
Je suis un cercle    Le centre de l'objet se trouve dans :   10  et   7
Je suis un rectangle   Le centre de l'objet se trouve dans :   4  et   7
Je suis une forme géométrique   Le centre de l'objet se trouve dans :   8  et   10
```

Je suis un cercle Le centre de l'objet se trouve dans : 8 et 5

Je suis un rectangle Le centre de l'objet se trouve dans : 10 et 17

Exemple 3

```
class test_poly3 {
public static void main (String [] args) {
String c ;
graphique  d;
int c=  Integer.parseInt(args [0]) ;
switch (c)  {
case 0 : d=new Cercle (10,10,10) ; break;
case  1 : d=new Rectangle (10,10,10,10) ; break;
case  2 : d= new graphique (10,10); break;
default: d= new graphique (0,0);
}
d.identifie () ;
System.out.println (« Surface = » + d.surface()) ;
}
}
```

Résultat de l'exécution :

Le message affiché est en fonction de l'argument introduit dans la commande d'exécution.

Le gros avantage du polymorphisme est de pouvoir référencer des objets sans connaître à la compilation véritablement leur classe et de pouvoir par la suite à l'exécution lancer le code approprié à cet objet. La liaison entre l'identificateur de la méthode polymorphe et son code est déterminée à l'exécution et non à la compilation, on parle alors de liaison dynamique.

Remarque :

8. Le polymorphisme se base essentiellement sur le concept d'héritage et la redéfinition des méthodes. Toutefois, si dans le même contexte d'héritage, on a à la fois une redéfinition et une surcharge de la même méthode, les règles de polymorphisme deviennent de plus en plus compliquées.
9. La notion de polymorphisme peut être généralisée dans le cas de dérivations successives.

5.3 Conversion des arguments effectifs

La conversion d'un type dérivé en un type de base est aussi une conversion implicite légale, elle va donc être utilisée pour les arguments effectifs d'une méthode.

Exemple1

```
class  A
{public void identifie()
{System.out.println (« Objet de type A ») ; }
}
```

```
class B extends A
{// pas de redéfinition de identitie ici
}
class Test
{ static void f(A a ) {
        a.identitie(); }
}
…..
A a = new A();
B b = new B ();
Test.f(a);
// OK Appel usuel; il affiche "objet de type A"
Test.f(b) ;
/*OK  une référence à un objet de type B est compatible avec une
référence à un objet de type A. L'appel a.identifie affiche : « Objet
de type A » */
```

Exemple2

On va juste modifier le corps de la classe B par une redéfinition de la méthode « identifie »

```
class  B {
public void identifie () {
System.out.println (« Objet de type B ») ;}
}
….
A a = new A(); B b = new B ();
Test.f (b) ;
/*OK  une référence à un objet de type B est compatible avec une
référence à un objet de type A. L'appel a.identifie affiche : « Objet
de type B »*/
```

Cet exemple, montre que comme dans les situations d'affectation, la conversion implicite d'un type dérivé dans un type de base n'est pas dégradante puisque grâce au polymorphisme, c'est bien le type de l'objet référencé qui intervient.

Remarque :

Dans le cas où la méthode f est surdefinie, il y a encore des règles plus compliquées lors de la conversion implicite des types de paramètres.

5.4 Les conversions explicites de références

Compatibilité entre référence à un objet d'un type donné et une référence à un objet d'un type ascendant. Dans le sens inverse, la compatibilité n'est pas implicite.

Exemple :

```
Class graphique {…….}
Class Cercle {……..}
…..
graphique g,g1 ;
Cercle c,c1 ;
C = new graphique (….) ; //Erreur de compilation
g= new Cercle (…) //Juste
g1 = new graphique (….) ; //évident
c1= new Cercle(…) ; //évident
c1 = g1 //faux  (Conversion implicite illégale)
g1= c1 ; //juste ; (conversion implicite légale)
```

Il faut réaliser une conversion explicite c1 = (Cercle) g1 ;

Toutefois cette syntaxe qui est acceptée à la compilation peut être refusée à l'exécution et le traitement peut être arrêtée si il n 'y a pas compatibilité effective entre les types.

TD2

Exercice 1: Gestion de comptes en banque

Une banque gère un ensemble de comptes de différents types : comptes courant, comptes d'épargne… Les comportements et les caractéristiques liés à un compte sont à un niveau abstrait fort similaires : chacun possède un propriétaire, un solde, un numéro, un taux d'intérêt : on y peut déposer de l'argent, retirer de l'argent (s'il en reste suffisamment)

La banque propose également un système d'emprunts. Chaque emprunt est caractérisé par le montant total à rembourser, la valeur des mensualités et le compte duquel les mensualités doivent être débitées

Le but de cet exercice consiste à créer une banque et à tenir à jour l'état des comptes en fonction des retraits, des intérêts et des remboursements d'emprunts sur chaque compte.

Exercice 2

Un fichier est caractérisé par : son nom, son extension, son emplacement et sa taille sur disque (en kilo octets).

Les fichiers sont enregistrés dans des supports magnétiques pouvant être des disquettes ou des disques durs

Une disquette est caractérisée par son nom logique, la taille d'un secteur en kilo octets, le nombre de secteurs par pistes et le nombre de pistes par surface.

Un disque dur est caractérisé par son nom logique, la taille d'un secteur en kilo octets, le nombre de secteurs par piste, le nombre de pistes par surfaces et le nombre de surfaces.

Afin de pouvoir gérer son contenu, on suppose que chaque disque contient deux tableaux

- Le premier contient toutes les caractéristiques d'un fichier donné
- Le second contient pour chaque fichier son adresse physique (les adresses physiques sont attribuées automatiquement par le système)

On se propose alors de développer les classes **Fichier**, **Disquette** et **DisqueDur** :

La classe **Fichier** doit contenir :

- Un constructeur paramétré qui accepte comme arguments une liste de valeurs permettant l'initialisation des différents attributs de l'objet en question;
- Des mutateurs permettant la modification des valeurs des attributs d'un objet;
- Des accesseurs permettant le renvoi des valeurs des attributs d'un objet;

Chacune des deux classes **Disquette** et **DisqueDur** doit contenir :

- Un constructeur paramétré qui accepte comme arguments une liste de valeurs permettant l'initialisation des différents attributs de l'objet en question;

- Une méthode ***getEspaceDisque()*** permettant de calculer et de retourner la taille totale du disque concerné (en kilo octets)
- Une méthode ***getEspaceLibre()*** permettant de calculer et de retourner l'espace libre dans un disque (en kilo octets)
- Une méthode ***ajouterFichier()*** permettant d'ajouter un fichier donné (ajouter un fichier revient tout simplement à l'insérer à la fin du tableau si la taille de disque suffit, on s'occupe pas de l'adresse ou il sera stocké)
- Une méthode ***explorerExtension()*** permettent d'afficher tous les fichiers (nom, extension, emplacement et taille) ayant comme extension celle qui est introduite comme paramètre.

Dans le but d'expérimenter le comportement de toutes les classes déjà développées on se propose d'ajouter une nouvelle classe nommée **Application** basée principalement sur la méthode ***main ().*** Dans cette méthode principale on doit :

- Construire les fichiers suivants :
  - (Nom : " examen " ; Extension : " doc " ; Taille 2 kilo ; Emplacement : " C:\word ")
  - (Nom : " bulletin " ; Extension : " xls " ; Taille 1 kilo ; Emplacement : " C:\excel ")
  - (Nom : " nature " ; Extension : " gif "; Taille 25 kilo ; Emplacement : " C : ")
- construire le disque dur suivant :

  (Nom logique : " C : "; Taille d'un secteur : 0.9375 kilo octet ; Nombre de surfaces: 16 ; Nombre de piste par surface : 1024 ; Nombre de secteurs par pistes : 520)

- ajouter tous les fichiers crées à ce disque
- afficher les noms des fichiers ayant comme extension « doc ».