LE POLYMORPHISME

Héritage et polymorphisme

- La notion de polymorphisme
  - C'est un concept très puissant en P.O.O, qui complète l'héritage.
  - Il permet de manipuler des objets sans en connaitre tout à fait le type.
    - Exemple: créer un tableau d'objets, les uns étant de type Point et les autres de type PointCol et appeler la méthode afficher(). Chaque objet réagira selon son type.
  - Le polymorphisme exploite la relation *est un (*en anglais *is a)* qui dit qu'un point coloré est aussi un point et peut donc être traité comme un point. La réciproque est fausse.

Héritage et polymorphisme

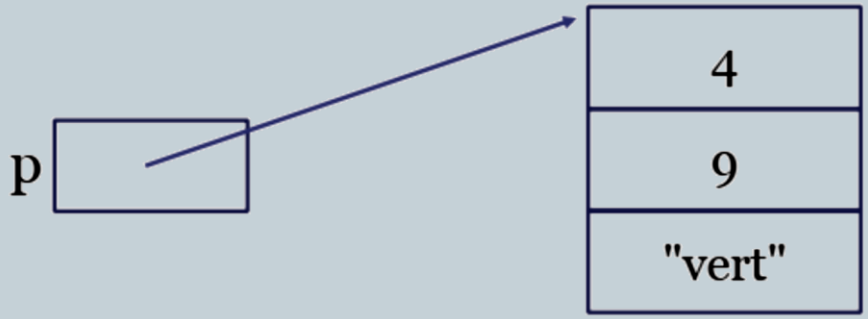
- Les bases du polymorphisme
  - Soient les classes Point et PointCol suivantes

```
class Point
{public Point (int x, int y) {....} // constructeur
public void afficher(){...} // affiche les coordonnées
}
class PointCol
{ public PointCol(int x, int y, String couleur) // constructeur
public void afficher() {...} // affiche les coordonnées et la couleur
}
```

Héritage et polymorphisme

- Les bases du polymorphisme
  - Point p;
  - p = new Point(3,5)

  - Java autorise cette affectation
  - p = new PointCol (4,9,"Vert");



Règle 1: Java permet à une variable objet l'affectation d'une référence à un objet d'un type dérivé

Héritage et polymorphisme

- Les bases du polymorphisme
  - Point p = new Point (3,5);
  - p.afficher(); // appelle la méthode afficher de la classe Point
  - p = new PointCol (4,9,"Vert");
  - p.afficher(); // appelle la méthode afficher de la clase PointCol
  - La deuxième instruction p. afficher() se base non pas sur le type de la variable p mais sur le type effectif de l'objet référencé par p au moment de l'appel car celui-ci peut évoluer dans le temps.
  - Ce choix qui se fait au moment de l'exécution et non pas au moment de la compilation s'appelle ligature dynamique (ou liaison dynamique).

Règle 2: Le choix de la méthode appelée se fait selon le type effectif de l'objet référencé au moment de l'exécution

Héritage et polymorphisme

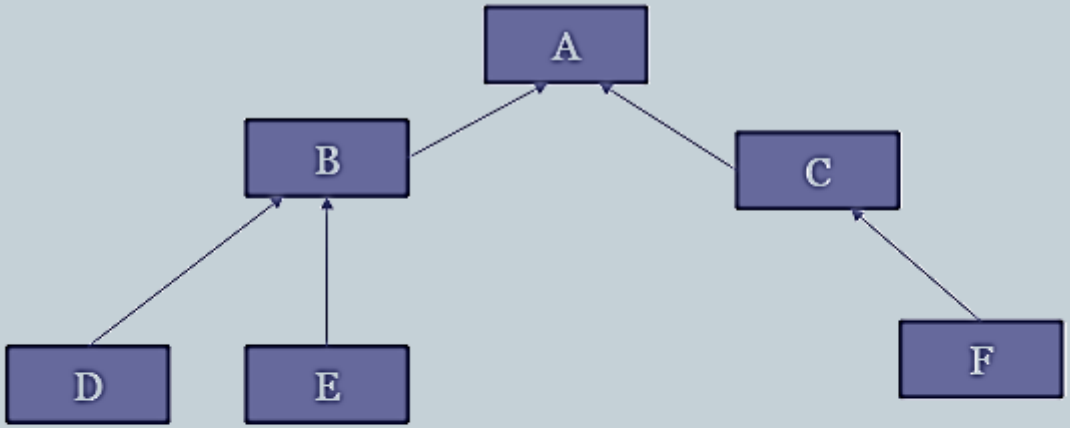
- Les bases du polymorphisme
  - En résumé:

Le polymorphisme en java se traduit par :

- La compatibilité par affectation entre un type classe et un type ascendant
- La ligature dynamique des méthodes

Héritage et polymorphisme

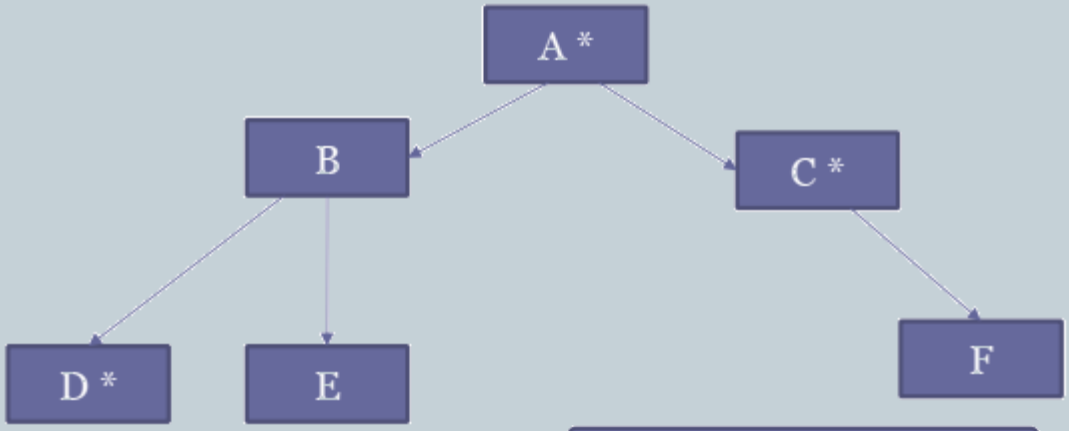
- Généralisation du polymorphisme à plusieurs classes
  - A a; B b; C c; D d; E e; F f;



a = b;	OUI	a = e;	OUI	b = e;	OUI
a = c;	OUI	b = a;	NON	c = f;	OUI
c = d;	NON	a = f;	OUI	d = c;	NON
a = d;	OUI	b = d;	OUI		

Héritage et polymorphisme

- Généralisation du polymorphisme à plusieurs classes
  - Soit une fonction f définie/redéfinie dans les classes comportant une étoile. Quelle est la méthode appelée dans les cas suivants?



a référence un objet de type A	// méthode f de A
a référence un objet de type B	// méthode f de A
a référence un objet de type C	// méthode f de C
a référence un objet de type D	// méthode f de D
a référence un objet de type E	// méthode f de A
a référence un objet de type F	// méthode f de C

Héritage et polymorphisme

- Polymorphisme, redéfinition et surdéfinition

```
class A {
public void f(float x) {…}
}
class B extends A {
public void f(float x) {…} //redéfinition de f de B
public void f (int n) {…} //surdéfiniton de f pour A et B
…….
}
A a = new A();
B b = new B();
int n;
```

```
a.f(n);   // appelle f (float) de A
b.f(n);   // appelle f (int) de B
a = b;    // a contient une référence vers un objet de type B
a.f(n);   // appelle f(float) de B
```

Héritage et polymorphisme

- Polymorphisme, redéfinition et surdéfinition

Explication:

- Bien que **a.f(n)** et **b.f(n)** appliquent toutes les deux une méthode **f** à un objet de type **B**, elles n'appellent pas la même méthode.
- Pour l'instruction **a.f(n)**, *le compilateur* recherche la meilleure méthode (selon les règles de surdéfinition) parmi toutes les méthodes de la classe **A** ou ses ascendantes. Ici, c'est la méthode void **f(float x)** de **A**.
- Une fois la bonne méthode trouvée, sa signature et son type de retour sont définitivement figés.
- *Lors de l'exécution*, on se base sur le type de l'objet référencé par **a** pour rechercher une méthode ayant la signature et la valeur de retour voulus. On aboutit alors à la méthode **f(float)** de **B**, et ce malgré l'existence dans **B** d'une méthode plus appropriée au type de l'argument effectif.
- Donc, malgré la ligature dynamique, le polymorphisme se fonde sur une signature et un type de retour définis ***à la compilation*** (*et qui ne seront pas remis en questions lors de l'exécution*)

Héritage et polymorphisme

- Polymorphisme, redéfinition et surdéfinition

L'utilisation simultanée de la surdéfinition et de la redéfinition peuvent conduire à des situations complexes.
Il est donc conseillé de les utiliser prudemment et de ne pas en abuser.

Héritage et polymorphisme

- Conversion des arguments effectifs: cas d'arguments de type objet
  - Cas 1.Méthode non surdéfinie et non redéfinie

```
class A {
public void indentite(){
System.out.println(" Objet de type A ");}
}
class B extends A{
// pas de redéfinition de la méthode identité
}
class Util {
static void f(A a) //objet de type A en argument
{ a.identite();}
}
A a = new A(...); B b = new B(...);
```

```
Util.f(a); //
Util.f(b);//
```

Affiche « Objet de type A »

Affiche « Objet de type A »

Héritage et polymorphisme

- Conversion des arguments effectifs: cas d'arguments de type objet
  - Cas 2.Méthode non surdéfinie et redéfinie

```
class A {
public void indentite (){
System.out.println(" Objet de type A ");}
}
class B extends A {
public void indentite () {
System.out.println(" Objet de type B ");}
}
class Util{
static void f(A a) {
a.identite();}
}
B b = new B();
```

```
Util.f(b);//
```

Affiche « Objet de type B »

Héritage et polymorphisme

- Conversion des arguments effectifs: cas d'arguments de type objet
  - Cas 3.Méthode surdéfinie (exemple simple)

```
class A {.....}
class B extends A {.....}
class Util
{
static void f(int n, B b) {....}
static void f(float x, A a){....}
}
...
A a = new A();
B b = new B();
int n; float x;
```

```
Util.f(n, b);   //sans conversion: Appel de f(int, B)
Util.f(x, a);   //sans conversion: Appel de f(float,A)
Util.f(n, a);   //conversion de n en float: Appel de f(float,A)
Util.f(x, b);   //conversion de b en A : Appel de f(float,A)
```

Héritage et polymorphisme

- Conversion des arguments effectifs: cas d'arguments de type objet
  - Cas 3.Méthode surdéfinie (Exemple moins trivial)

```
class A {.....}
class B extends A {.....}
class Util
{
static void f(int p, A a) {....}
static void f(float x, B b){....}
}
...
A a = new A();
B b = new B();
int n; float x;
```

```
Util.f(n, a);   //sans conversion: Appel de f(int, A)
Util.f(x, b);   //sans conversion: Appel de f(float,B)
Util.f(n, b);   //Erreur de compilation: Ambiguité car deux possibilités existent
Util.f(x, a);   //Erreur de compilationcar aucune méthode ne convient, impossible de convertir A en B ni float en int
```

- Les règles de polymorphisme en java

  - L'abus des possibilités de redéfinition et surdéfinition peuvent conduire à des situations complexes(comme on a pu le voir), voici un récapitulatif des différentes règles:

    - Compatibilité. Il existe une conversion implicite d'une référence à un objet de classe C en une référence à un objet d'une classe ascendante (aussi bien dans les affectations que dans les arguments effectifs de méthodes)

- Les règles de polymorphisme en java

  - L'abus des possibilités de redéfinition et surdéfinition peuvent conduire à des situations complexes(comme on a pu le voir), voici un récapitulatif des différentes règles:

    - Ligature dynamique. Dans un appel de la forme x.f(...) où x est supposé de type C, le choix de f est déterminé de la manière suivante:

      - A la compilation: On détermine dans la classe C ou ses ascendante la signature de la meilleure méthode f convenant à l'appel, ce qui définit également le type de retour.

- Les règles de polymorphisme en java

  - L'abus des possibilités de redéfinition et surdéfinition peuvent conduire à des situations complexes(comme on a pu le voir), voici un récapitulatif des différentes règles:

    - Ligature dynamique. Dans un appel de la forme x.f(...) où x est supposé de type C, le choix de f est déterminé de la manière suivante:

      - l'exécution: On recherche la méthode f de signature et de type de retour voulus, à partir de la classe correspondant au type effectif de l'objet référencé par x (Il peut être de type C ou descendant). Si cette classe ne comporte pas de type approprié, on remonte dans la hiérarchie jusqu'à en trouver une.

Héritage et polymorphisme

- Conversion explicite de référence
  - Il n'est pas possible d'affecter à une référence à un objet de type T , une référence à un objet d'un type ascendant.
  - Exemple:

```
class Point{…}
class PointCol extends Point {…}
…
PointCol pc;
pc = new Point (…)// erreur de compilation
Considérons maintenant cette situation
Point p; PointCol pc1, pc2;
pc1 = new PointCol (…);
…
p= pc1; // p contient la référence à un objet de type PoinCol
…
pc2 = p; // Erreur de compilation même si p contient une référence à //un objet de type PoinCol
```

Héritage et polymorphisme

- Conversion explicite de référence
  - Il faudrait faire une conversion en utilisant l'opérateur de cast comme pour les types primitifs.

Point p; PointCol pc1, pc2;
pc1 = new pointCol (…);
…
p= pc1; // p contient la référence à un objet de type PoinCol
…
pc2 = (Pointcol) p; // accepté par le compilateur mais sera vérifié à l'exécution. Si p ne contient pas une référence à un objet de type PointCol ou dérivé, l'exécution sera arrêtée

Héritage et polymorphisme

- La super-classe Object
  - La classe Object est le sommet de la hiérarchie de toutes les classes java. Toutes les classes dérivent de la classe Object (qu'elles soit prédéfinies dans java ou créées)
  - Si nous définissons une classe Point

```
class Point
{…
}
```

  - Ceci veut dire implicitement

```
class Point extends Object
{…
}
```

Héritage et polymorphisme

- Utilisation d'une référence de type Object

  - Etant données les possibilités offertes par le polymorphisme, une variable de type Object peut être utilisée pour référencer un objet de type quelconque. Ceci peut être utilisé pour manipuler des objet dont on ne connait pas le type exact au moment où on l'utilise.

  - Exemple:

```
Point p = new Point();
Object o;
...
o= p;
o.deplacer (); // erreur de compilation
```

```
((Point ) o). deplacer (); //OK
```

Héritage et polymorphisme

- Utilisation de méthodes de la classe Object
  - La classe Object comporte quelques méthodes qu'on peut soit utiliser telles quelles soit les redéfinir. Les plus importantes sont :
    - toString : fournit une chaine contenant le nom de la classe à laquelle appartient l'objet ainsi que l'adresse de l'objet en hexadecimal

```
Point p = new Point (3,5);
System.out.println(" p = " + p.toString() ); //
affiche par exemple a = Point@fc17aedf
```

    - equals: compare les adresse de deux objets.

```
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
...
```

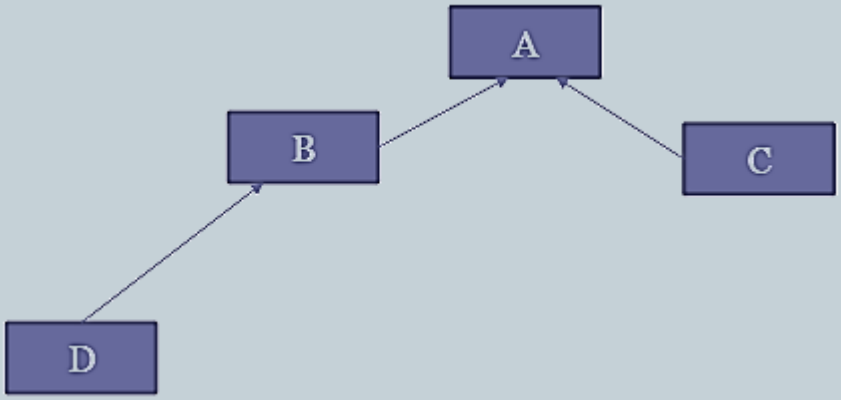
p1.equals(p2) retourne la valeur ....? false

- Les membres protégés
  - Nous avons vu les droits d'accès publique (avec le mot clé public) et privé (avec le mot clé private).
  - Il existe un troisième droit d'accès dit protégé (en utilisant le mot clé protected).
  - Un membre déclaré protected est accessible à ses classes dérivées.

Héritage et polymorphisme

- Les membres protégés

  - Exemple:
    class A
    {....
    protected int n ;
    }
    B accède à n de A
    D accède à n de B et de A
    C accède à n de A
    Mais C n'accède pas à n de B car aucun lien de dérivation ne relie C à B



Héritage et polymorphisme

- Héritage et tableaux

  - Même si nous considérons les tableaux comme étant des objets, il n'est pas possible de définir leur classe et par conséquent ils ne bénéficient que d'une partie des propriétés des objets.

  1. Un tableau peut être considéré comme appartenant à une classe dérivées d'Object

```
Object o;
o = new int[5]; // correct
o = new float [3]; // correct
```

  2. Il n'est pas possible de dériver une classe d'un tableau

```
class Test extends int []; // erreur
```

Héritage et polymorphisme

- Polymorphisme et tableaux
  - Le polymorphisme peut s'appliquer à des tableaux d'objets. Si B dérive de A, un tableau d'objets de type B est compatible avec un tableau d'objets de type A

```
class B extends A {.....}
A ta[];
B tb[];
.....
ta = tb; // OK car B dérive de A
tb = ta; // erreur
```

Mais cette propriété ne peut pas s'appliquer aux types primitifs

```
int ti [], float tf[];
....
ti = tf; // erreur ( évident car float n'est pas compatible avec int)
tf = ti; // erreur bien que int soit compatible avec float
```

Héritage et polymorphisme

- Classes et méthodes finales
  - Lorsque le mot clé final est appliqué à une variable ou à des champs d'une classe, il interdit la modification de leur valeur.

```
final int n = 23;
...
n=n+5; // erreur
```

  - Le mot clé final peut s'appliquer à une classe ou à une méthode mais avec une signification totalement différente
  - Une méthode déclarée finale ne peut pas être redéfinie dans une classe dérivée
  - Une classe déclarée finale ne peut pas être dérivée