Polymorphisme

c. Polymorphisme

L'héritage offre la possibilité pour une classe de s'adresser à une autre, en sollicitant de sa part un service qu'elle est capable d'exécuter de mille manières différentes, selon que ce service , nommé toujours d'une seule et même façon, se trouve redéfini dans autant de sous-classes, toutes héritant du destinataire du message. C'est la base du polymorphisme. Cela permet à notre première classe de traiter toutes ses classes interlocutrices comme une seule , et de ne modifier en rien son comportement si on ajoute une de celles-ci ou si une de celles-ci modifie sa manière de répondre aux messages de la première :
simplicité de conception, économie d'écriture et évolution sans heurt : tout **l'Orienté Objet** est là, dans le **polymorphisme**

Le polymorphisme en Java est la capacité des sous-classes d'avoir leur propre comportement et, en même temps, de partager quelques fonctionnalités avec d'autres sous-classes.

i. Surcharge de méthodes

La surcharge (surdéfinition) de méthode à l'intérieur d'une même classe consiste à définir des méthodes de même nom, mais de signatures différentes (en modifiant le nombre de paramètres, le type de paramètres ,les deux en même temps ou le type de retour). Une classe dérivée pourra à son tour surcharger une méthode d'une classe ascendante. Bien entendu, la ou les nouvelles méthodes ne deviendront utilisables que par la classe dérivée ou ses descendantes, mais pas par ses ascendantes.

Exemple 1 :

```
public class Compte {
	public double solde;
	public void calcul (){solde+=300;}
	public void afficher (){System.out.println("solde = :"+solde);}}

public class Compte_normal extends Compte {
	public void calcul (double taux){ //surcharge de la méthode calcul en
//ajoutant le paramètre taux
		super.calcul(); //appel de la méthode redefinie
		solde*=taux;} // mise a jour du solde

	public void afficher (){System.out.println("solde = :"+solde);}}

public class test {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Compte c1=new Compte();
		Compte_normal c2=new Compte_normal();
		c1.calcul();
		c2.calcul(1.07);
		c1.afficher();c2.afficher();}}
```

ii. Redéfinition de méthodes (Overriding)

On peut redéfinir une méthode de la classe de base dans une classe dérivée, comportant la même signature qu'une méthode de classe de base. L'attribut d'accès de la méthode dans une classe dérivée peut être identique a celui de la classe de base, ou moins restrictif, mais il ne peut pas être plus restrictif

- Toute méthode est déclarée comme public dans une **classe de base** doit être également comme **public** dans **la classe dérivée**.
- On ne peut pas omettre l'attribut d'accès de la classe dérivée ou le specifier comme **privée** ou **protected**.

Impératif :

On doit respecter la signature de la méthode qu'on redefinit!

- ✓ ses paramètres (nombre, type, ordre)
- ✓ son type de retour
- ✓ son attribut d'accessibilité´e : on peut élargir son accès.
- ✓ La méthode de la super-classe est accessible mais en la préfixant du mot **super**

Une méthode **package** peut être redefinie en une méthode **public**.

Une méthode **public** ne peut pas devenir **private** dans une sous-classe.

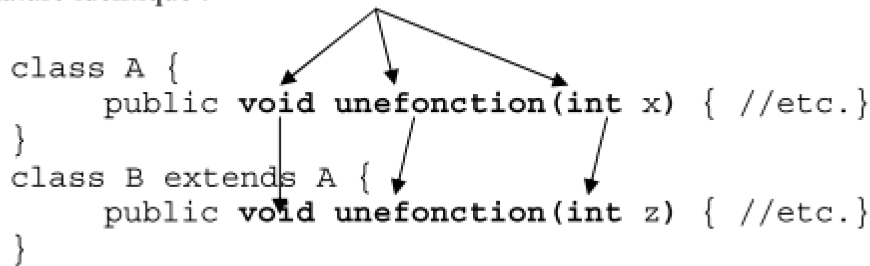
Exemple 2 :

```
public class Compte {
	public double solde;
	public void calcul (){solde+=300;}
	public void afficher (){System.out.println("solde = :"+solde);}}


public class Compte_normal extends Compte {
	public void calcul (){
		super.calcul(); //appel de la méthode redéfinie
		solde*=1.07;} // mise a jour du solde
		public void afficher (){System.out.println("solde = :"+solde);}}
public class test {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
				Compte c1=new Compte();
		Compte_normal c2=new Compte_normal();
		c1.calcul();
		c2.calcul();
		c1.afficher();
		c2.afficher();      }}
```

ii.1.Contraintes sur la redéfinition

a- signature identique :



b- droits d'accès : la redéfinition ne doit pas diminuer les droits d'accès à une méthode.

```
class A {
     private void unefonction(int x) { //etc.}
}
class B extends A {
     public void unefonction(int z) { //etc.}
}
```

Remarques 1 :

- La re-déclaration doit être strictement identique à celle de la super-classe : même nom, même paramètres et même type de retour.
- Grâce au mot-clé super, la méthode redéfinie dans la sous-classe peut réutiliser du code écrit dans la méthode de la super-classe, qui n'est plus visible autrement.
- Si la signature de la méthode change, ce n'est plus une **redéfinition** mais une **surcharge**.

Exemple 3 : redéfinition

```
//Super-classe
class graphiques {
private double  x, y ;
public String couleur ;
public void affiche () {
System.out.println ("« Le centre de l'objet = ( " + x + " , " + y + " ) " ) ; }
double surface ( ) {return (0) ;} } //fin de la classe graphiques


//classe dérivée
class cercle extends graphiques{
double rayon ;
public double surface ( ) {
return (rayon * 2* 3.14) ;} //redéfinition de «surface»
public void affiche () //redéfinition de la méthode «affiche»
{super.affiche () ; // appel de affiche de la super-classe
System.out.println (" Le rayon = " + rayon + " La couleur = " + couleur ) ;}}
```

Exemple 4 : surcharge

```
class A
{public void f (int n) { } }
class B extends A
{public void f (float n) {} } // fin de la classe B


A a ; B b ;
int n; float x;
a.f(n) ;// appelle f(int) de A
a.f(x); // Erreur, on a pas dans A f(float)
b.f(n); // appelle f(int) de A
b.f(x); // appelle f(float) de B
```

Exemple 5 : Utilisation simultanée de surcharge et de redéfinition

```
class A
{public void f (int n) {}
public void f (float n) {} }
class B extends A
{public void f (int n) {} // redefinition de la méthode f( int ) de A
public void f (double n) {} // Surcharge de la méthode f de A et de B
} // fin de la classe B


A a ; B b ;
int n ; float x; double y ;
a.f(n) ;//Appel de f(int) de A
a.f(x) ;//Appel de f(float) de A
a.f(y) ; //Erreur, il n' y a pas f(double) dans A
b.f(n) ;
//Appel de f(int) de B car f(int) est redéfinie dans B
a.f(x) ;//Appel de f(float) de A
a.f(y) ;//Appel de f(double) de B
```

Remarque 2 :

-Si une méthode d'une classe dérivée a la même signature q'une méthode d'une classe ascendante :

- Les 2 méthodes doivent avoir le même type de retour.
- Le droit d'accès de la méthode de la classe dérivée ne doit pas être moins élevé que celui de la classe ascendante.
- Une méthode statique ne peut pas être redéfinie.

Exemple 6 : polymorphisme

Pour bien exploiter le concept de l'héritage et le polymorphisme, nous allons reprendre les classes de **l'exemple 2,** tout en ajoutant la déclaration d'un tableau pour stocker les objets de la classe Compte :

```
public class test {

	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Compte c1=new Compte();
		Compte_normal c2=new Compte_normal();
		Compte c3=new Compte();
		Compte_normal c4=new Compte_normal();
		Compte t[]=new Compte[4];

		// un tableau qui va stocker des objets des types différents
		//dont chaque objet peut avoir un comportement différent des autres
		//tout en exécutant leurs méthodes correspondantes
		t[0]=c1;t[1]=c1;t[2]=c1;t[3]=c1;
		for (int i=0;i<4;i++){t[i].calcul();t[i].afficher();}     } }//chaque
objet exécute son propre méthode
```

d. Classes abstraites (utilisation et importance)

Le concept de classe abstraite se situe entre celui de classe et celui d'interface (voir section suivante). C'est une classe qu'on ne peut pas directement instancier car certaines de ses méthodes ne sont pas implémentées. Une classe abstraite peut donc contenir des variables, des méthodes implémentées et des signatures de méthode à implémenter. Une classe abstraite peut implémenter (partiellement ou totalement) des interfaces et peut hériter d'une classe ou d'une classe abstraite.

Le mot-clé **abstract** est utilisé devant le mot-clé class pour déclarer une classe abstraite, ainsi que pour la déclaration de signatures de méthodes à implémenter.

a. Intérêt

- Placer dans une classe abstraite toutes les fonctionnalités que nous souhaitons disposer lors de la création des descendants.
- Une classe abstraite sert comme classe de base pour une dérivation.

b. C'est quoi au juste?

- Une classe abstraite ne permet pas l'instanciation des objets.
- Une classe abstraite peut contenir des méthodes et des champs (qui peuvent être hérités) et une ou plusieurs méthodes abstraites.
- une méthode abstraite est une méthode dont la signature et le type de la valeur de retour sont fournis dans la classe et rien d'autre (pas le corps de la méthode i.e. définition).

c. Syntaxe

abstract class A { **// etc.**}

d. Particularités

- Une classe ayant une méthode abstraite est par défaut abstraite. Donc pas besoin de "abstract" à ce stade.
- Les classes abstraites doivent être déclarées "public" sinon pas d'héritage!
- Une classe dérivée d'une classe abstraite n'est pas obligée de redéfinir toutes les méthodes abstraites. Elle peut même n'en définir aucune. Si c'est le cas, elle reste abstraite.
- Une classe dérivée d'une classe non abstraite peut être déclarée abstraite.

Remarques 3 :

On dit alors que la classe ou la méthode est abstraite.

- Une méthode abstraite n'a que sa signature (son prototype), c'est -à-dire son type de retour suivi, de son nom, suivi de la liste de ses paramètres entre des parenthèses, suivi d'un point-virgule.
- Une méthode abstraite ne peut pas être déclarée static ou private ou final.
- Dès qu'une classe contient une méthode abstraite, elle doit elle aussi être déclarée abstraite.
- Une classe abstraite ne peut pas être instanciée. Il faudra l'étendre et définir toutes les méthodes abstraites qu'elle contient pour pouvoir l'utiliser.

Voici un petit exemple de classe abstraite :

```
public abstract
class Personne{  public abstract
String getNom() ;  public
String toString(){  return  "Mon nom est " + getNom();  } }
```

Exemple 7: classe abstraite

Imaginons que l'on souhaite attribuer deux variables,origine_x etorigine_y, à tout objet représentant une forme. Comme une interface ne peut contenir de variables, il faut transformer Forme en classe abstraite comme suit :

```
public abstract class Forme{
            private double origine_x;
            private double origine_y;
            public Forme(){
            this.origine_x=0;
            this.origine_y=0;}
            public double getOrigineX(){return this.origine_x;  }
            public double getOrigineY(){return this.origine_y;  }
            public void setOrigineX(double x){this.origine_x=x; }
            public void setOrigineY(double y){this.origine_y=y; }
            public abstract double surface();
            public abstract void affiche();{      } }
```

De plus, il faut rétablir l'héritage des classes Rectangle et Cercle vers Forme :

```
public class Rectangle extends Forme {}
public class Cercle extends Forme {}
```

e. Interfaces (utilisation et importance)

L'interface est une notion indépendante de l'héritage et donc de la classe dérivée. Une classe dérivée peut implémenter une ou plusieurs **interfaces**. Elle peut dérive d'une autre interface en utilisant **extends**, réservé pour l'héritage. En réalité, l'opération a simplement permis de concaténer les déclarations. Les droits d'accès ne sont pas pris en compte ce qui n'est pas le cas lors des héritages des classes. L'interface n'est qu'une **classe abstraite particulière.**

Une interface définit:
- ➢ Un type
- ➢ Des méthodes sans leur code (méthodes abstraites)
- ➢ Une interface ne peut pas être instanciée
- ➢ Elle est destinée à être « implémentée » par des classes

a. Intérêt

- ➢ Implémenter l'héritage multiple en java
- ➢ Une interface sert essentiellement à regrouper des constantes
- ➢ permettre qu'un ensemble de classes,
  étendant éventuellement d'autres classes, puissent implémenter une même interface

b. utilisation

- ➢ Un nom d'interface peut être utilisé comme un nom de classe
  On pourra donc mettre dans une variable dont le type est une interface n'importe quel objet implémentant l'interface en question
- ➢ Quelque soit l'objet référencé par la variable, on pourra lui appliquer une méthode déclarée dans l'interface
- ➢ Elle est destinée à être « implémentée » par des classes
  - À qui elle donnera son type
  - Qui fourniront des définitions pour les méthodes déclarées (code)
- ➢ Donner un type commun à des classes différentes pour en faire un même usage

c. Syntaxe

class MaClasse **extends** AutreClasse **implements** UneInterface, UneAutreInterface

d. Le mot réservé : **implements :**

Ce mot est employé dans l'en-tête de la déclaration d'une classe, suivi d'un nom d'interface ou de plusieurs noms d'interfaces séparés par des virgules. S'il y a une clause **extends**, la clause **implements** doit se trouver après la clause **extends**.

e. Particularités

Si une classe non abstraite possède une clause implements, elle doit définir toutes les méthode s de l'interface indiquée.

- ➢ À qui elle donnera son type
- ➢ Qui fourniront des définitions pour les méthodes déclarées (code)
- ➢ Donner un type commun à des classes différentes pour en faire un même usage
- ➢ Une interface peut contenir des attributs qui doivent être initialisés par des expressions constantes. Ces attributs sont automatiquement **final** et **static**, et sont donc des constantes.
- ➢ Une interface peut contenir des prototypes de méthodes. Celles-ci sont automatiquement publiques.
- ➢ Une interface ne contient pas de constructeur.
- ➢ Une classe quelconque peut implémenter plusieurs interface

Exemple 8:

```
//Classe abstraite Frome

public abstract class Forme {
	private String nom;
	public Forme(String x){   this.nom=x;}
	public String getNom(){   return this.nom;    }
	public void setNom(String x){this.nom=x;     } }

// l'interface Surface

public interface Surface {
	public abstract double surface(); // méthode abstraite surface
	public abstract void affiche();} // méthode abstraite affiche


// Classe Rectangle

public class Rectangle extends Forme implements Surface{
	private double longueur ;
	private double largeur;
	public Rectangle (String c,double x,double y) {
		super(c);
		this.longueur=x;
		this.largeur=y;     }

// la classe Rectangle doit implémenter les deux méthodes de l'interface Surface

	public  double surface(){return longueur*largeur;};
	public  void affiche(){System.out.println("nom : "+super.getNom()+"
longueur=  "+longueur+"    largeur =  "+largeur+"    surface=  "+surface());}
	}

//classe de test

public class Test {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		Rectangle r1=new Rectangle ("rectangle",10,30);
		r1.affiche();      }}
```