Polymorphisme, la classe Object, les package et la visibilité en Java.

Ce document n'est pas un polycopié, c'est seulement un résumé des principales notions abordées en cours, il vient en complément du cours d'amphi et ne contient pas toutes les notions et les illustrations qui ont été présentées.

Polymorphisme.

Le polymorphisme est le résultant de :

1. **L'héritage de type** qui permet d'unifier les types d'une hiérarchie de classes concrètes différentes. L'héritage de type fournit une interface fonctionnelle commune à toutes ces classes concrètes permettant ainsi l'évocation de leur service.
2. **Les liaisons dynamiques** qui assure la préservation du comportement d'un objet indépendamment du type qui déclare.

Pour java on parle de polymorphisme simple car les liaisons dynamiques choisissent le comportement d'un service redéfinit en fonction du type réel de l'objet. On dit simple car le type réel des paramètres n'est pris en compte (on parle de polymorphisme multiple lorsque le type réel des paramètres est considéré par l'algorithme de sélection).

Le polymorphisme simple permet une factorisation de code au niveau des clients.

Par exemple, considérons l'interface suivante:

```
public interface Forme implements Cloneable
{
        .......
        public void dessiner();
        public void translater(double dx, double dy);
        public Object Clone();
        .....
}
```

Elle permet de manipuler toutes les formes en dehors de leur instanciation concrète. On peut donc écrire une classe Dessin qui seraient la suivante avec une implémentation naïve.

```
Public class Dessin
{
	private Forme [] dessin;
	private int indiceCourant;

	public Dessin() {
		dessin = new Forme[10]; // On crée un tableau de référence à des formes en utilisant
					// Forme qui est la racine de la hiérarchie de classe.
		indiceCourant = -1;
	}

	public void addForme(Forme f){
		if(indiceCourant >=9) // agrandir dessin et recopier les formes.
		dessin[++indiceCourant] = f.clone();// permet de se protéger des effets de bords.
							// on utilise un service commun à toutes les formes
							// mais qui dépend du type réel de la forme
	}

	public void translater(double dx, double dy){
		for(int i=0; i <= indiceCourant; i++)
		  dessin[i].translater(dx,dy); // Ce code est applicable pour toutes les formes.
						// On utilise le service translater spécifique au type réel
	}

	public void dessiner() {
		for(int i=0; i <= indiceCourant; i++)
		  dessin[i].translater(dx,dy); // Ce code est applicable pour toutes les formes.
						// On utilise le service dessiner spécifique au type réel
	}
}
```

Sur cet exemple, la classe Dessin offre une factorisation de code pour la manipulation des formes, puisqu'il est indépendant des implémentations de Forme.

Par exemple la classe Carre.

```
public class  Carre implements Forme
{
	private Point2D centre;
	private double coté;

	public Carre(Point2D p, double c){
		centre = p.clone();
		coté = c;
	}
```

```
	public void dessiner()
	{
		System.out.println(« Je suis une Forme Centre » + centre.toString());
		System.out.println(« Je suis un carré »);
	}
	public void translater(double dx, double dy);
	{
		centre.translater(dx,dy);
	}
	public Object Clone();
	{
		Carre tmp = (Carre) super.clone();
		tmp.centre = centre.clone();
	}
}
```

On peut maintenant ajouter des carrés dans un dessin en faisant dessin.addForme(new Carre(p,10)) par exemple. Lorsque on évoque le service dessin.dessiner(), on utilisera le service dessiner du carre ajouter. Cet exemple illustre, l'indépendance entre le code de la classe dessin et la réalisation concrète de Forme qui est faîte par la classe Carre. Maintenant, si nous fournissons une nouvelle réalisation de l'interface Forme en écrivant la classe Cercle, nous pourrons mettre des Cercles et Carrés dans un dessin tout en préservant leur comportement spécifique grâce au polymorphisme. Le code de la classe Cercle est alors le suivant.

```
public class  Cercle implements Forme
{
	private Point2D centre;
	private double rayon;

	public Carre(Point2D p, double r){
		centre = p.clone();
		rayon= r;
	}
	public void dessiner()
	{
		System.out.println(« Je suis une Forme Centre » + centre.toString());
		System.out.println(« Je suis un Cercle»);
	}
	public void translater(double dx, double dy);
```

```
        {
                centre.translater(dx,dy);
        }
        public Object Clone();
        {
                Cercle tmp = (Cercle) super.clone();
                tmp.centre = centre.clone();
                return tmp;
        }
}
```

On peut donc maintenant ajouter à la fois des Cercles et des Carrés dans un dessin. En effet, un dessin est un ensemble de Formes et les carrés et les cercles sont des Formes on peut donc les insérer dans un dessin. Par exemple,

```
Dessin dessin = new Dessin();
dessin.addForme(new Carre(p, 10));
dessin.addForme(new Cercle(q,20));
dessin.dessine(); // On dessinera un cercle et un carre même si ils sont considérés comme des formes,
                  // il conserveront leur comportement spécifique.
```

Le DownCast

En java, il est possible de revenir du type déclaré vers un sous type du type réel, cela consiste à utiliser le down cast. Par exemple, si on crée une poule qui est déclarée comme un animal

```
Animal a = new Poule();
```

Le type réel de m est Poule son type déclaré est Animal. On peut revenir vers le type Poule en utilisant le down cast.

```
Poule p = (Poule) a;
```

Ou bien encore la considérer comme un Mammifère.

```
Mamifère m = (Mammifère) a;
```

Dans le cas, où le type réel de l'objet n'est pas compatible avec celui du down cast, une exception sera levée. Le down cast permet donc de revenir vers un sous type du type réel en bénéficiant ainsi des services présents dans l'interface fonctionnelle publique du sous type.

La Classe Object.

Il existe en Java, une unique racine pour l'héritage qui est appelé la classe Object. Toute classe hérite du type et du code de la classe Object. Comme pour exister une interface doit être implémenté par une classe, on peut considérer que toute interface hérite de l'interface fonctionnelle de Object.

La classe Object permet donc de présenter le comportement minimal de tout objet ainsi que le code par défaut pour ce comportement minimal. Nous ne nous intéresserons qu' à certains des services

présentés par Object.

```
Public class Object
{
        public boolean equals(Object o){.....}
        public String toString() {.....}
        protected Object clone(){.....}
}
```

Le service **equals** sert à comparer si deux objets sont égaux. Cela veut dire si ils sont dans le même état. Par défaut, le code de equals dans la classe object est le suivant:

```
public boolean equals(Object o)
{
        return this == o;
}
```

Dans ce cas deux objets sont égaux si ils référencent le même objet physique. C'est à dire que deux objets sont égaux si il s'agit du même objet. On peut avoir un besoin moins restrictif de l'égalité de deux objets. Par exemple, deux objets de type Point peuvent être deux objets différents mais à un moment donné ils peuvent avoir les mêmes coordonnées dans un repère. Dans ce cas, on peut considérer que deux points sont égaux si ils ont les mêmes coordonnées. Le code pour la classe Point2D est alors le suivant:

```
public boolean equals(Object o) {
        Point2D tmp =  (Point2D) o;
        return this.getX() == tmp.getX() && this.getY() == tmp.getY();
}
```

Ou aussi pour une formeCentre

```
public boolean equals(Object o) {
        FormeCentre tmp =  (FormeCentre) o;
        return tmp.centre.equals(o.centre);
}
```

Le service **toString() p**ermet d'afficher des informations sur un objet, par défaut il affiche la classe et la clef d'indentification de l'objet (clef de hashage). Si on veut par exemple afficher des informations complémentaires le code peut alors être :.

```
public String toString()
{
        String addInfo =  « Des informations complémentaires »;
        return super.toString() + addInfo;
}
```

Enfin le dernier service est clone(). Ce service est protected et il doit retourner un nouvel objet

identique à l'objet à dupliquer. Par défaut, si on veut dupliquer un objet, il faut explicitement la redéfinir dans la classe et la déclarer comme publique. Par exemple,

```
public Object clone()
{
        return super.clone();
}
```

Le comportement par défaut de clone au niveau de object, consiste à allouer une place mémoire équivalente à celle définit par le type réel de l'objet. Puis ensuite, il effectue une copie bit à bit de l'objet à recopier vers l'objet copie. Pour les types primitifs cela ne pose pas de problème par exemple pour un objet de type Point2D. Le comportement suivant est identique à celui de clone.

```
public Object clone()
{
        Point2D tmp = (Point2D) super.clone();
        tmp.x = getX();
        tmp.y = getY();
        return tmp;
}
```

Il est inutile dans ce cas, de redéfinir la méthode clone car celle de objet est suffisante. Par contre, si l'on doit non pas partager une référence mais un nouvel objet dans le même état, il est nécessaire de redéfinir clone c'est par exemple le cas pour une FormeCentre qui contient une référence à un Point2D.

```
Public Object clone()
{
        FormeCentre tmp = (FormeCentre) super.clone();
        tmp.centre = this.centre.clone();
        return tmp;
}
```

Factorisation de Code.

En dehors du polymorphisme, on peut considérer que les carrés et les cercles sont des formes centrées. Et donc créer une classe abstraite FormeCentré qui factorise le code de Carre et de Cercle.

```
public abstract class  FormeCentre implements Forme
{
        protected Point2D centre;
        protected FormeCentre(Point2D p){
                centre = p.clone();
        }
        public String toString() // Adaptation du comportement de Object.
```

```
	{
		System.out.println(« Je suis une Forme Centre » + centre.toString());
	}

	public void dessiner()
	{
		System.out.println(toString());
	}
	public void translater(double dx, double dy);
	{
		centre.translater(dx,dy);
	}
	public Object clone() // D éclaré protected par Object
	{
		FormeCentre tmp = (FormeCentre) super.clone();
		tmp.centre = centre.clone();
		return tmp;
	}
	public boolean equals(Object o) // Déclaré public par Object
	{
		FormeCentre tmp = (FormeCentre) o;
		return tmp.centre.equals(O.centre);
	}
}
```

Cette classe redéfini le comportement par défaut de Object en l'adaptant à ses spécificités. Il faut changer le comportement de equals et clone car une FormeCentre contient un point. Deux formes centrées sont égales si leurs centres sont égaux. Et lorsqu'on duplique une forme on duplique aussi son centre. Concernant la classe Carré par exemple, il est inutile de redéfinir le comportement de clone car celui de FormeCentre est suffisant puisque le cote qui est un type primitif sera copié au moment du code de object, par contre celui de FormeCentre doit être redéfini car le centre est partagé par valeur et non par référence. Pour ce qui est de equals, deux carrés sont égaux si ils ont le même coté et sur leur forme centrée sont égales, le code est alors :

```
public abstract class Carre extends  FormeCentre
{
	protected double cote;
	protected Carre(Point2D p, double c){
		super(p);
		cote = c;
	}
```

```
	public String toString() // Adaptation du comportement de Object.
	{
	String s = super.toString() + « je suis un carre de cote » + new Double(cote).toString();
		return s;
	}

	public void dessiner()// on utilise le code de FormeCentre et on le spécialise.
	{
		super.toString();
		System.out.println(toString());
	}



	// public void translater(double dx, double dy) inutile de la redéfinir translater un carre
	                                                consiste à translater son centre.

	//public Object clone()  inutile de la redéfinir

	public boolean equals(Object o) // Déclaré public par Object
	{
		Carre tmp = (Carre) o;
		return super.equals(o) && tmp.cote == cote;
	}
}
```

La notion de package.

Il existe en Java, une notion modulaire supplémentaire qui est celle de package. On package est un ensemble de classe qui ont une sémantique commune. Le package permet de regrouper des classes publiques qui ont un lien entre elles. Par exemple, le package FormeGeo, peut regrouper toutes les classes qui ont un lien avec les formes géométriques. Dans un package, il y a deux types d'informations,

- Les informations publiques: ceux sont des classes ou des interfaces qui sont visibles de l'extérieur du package.
- Les informations locales aux paquetages: ceux des classes ou des interfaces qui ne sont visibles qu'à l'intérieur du package, elles servent en principe à l'implémentation des informations publiques du package.

Pour déclarer l'appartenance des informations d'un fichier suffixé « .java » à un package on utilise l'instruction

package NOMPACKAGE;

Cette instruction doit être la première ligne du fichier. Toutes les classes ou interfaces déclarées dans ce fichier appartiendront au package NOMPACKAGE. Parmi celle ci, une ou aucune seront déclarés publiques, dans ce cas la classe ou l'interface publique devront avoir le même nom que le fichier « .java ». Les autres classes ou interfaces seront alors des entités locales au package et ne seront pas visibles à l'extérieur du package.

Pour pourvoir accéder aux interfaces ou classes publiques d'un package, il faut déclarer son utilisation en utilisant l'instruction

import NOMPACKAGE.*;

Dans ce cas, toutes les classes ou interfaces publiques déclarées dans NOMPACKAGE seront accessible depuis ce fichier, les entités locales à NOMPACKAGE ne seront pas accessibles.

ATTENTION: Un package est représenté par un ensemble de répertoires du système de fichiers. Pour pouvoir accéder à un package, il faut que le/les répertoires qui contiennent les entités du package soit présents dans la variable d'environnement CLASSPATH.

Le nom d'un package représente un chemin dans le système de fichier à partir d'un point d'entrée dans la variable CLASSPATH, mais il n'existe pas la notion de sous package. Par exemple, FormeGeo.Coniques n'est pas un sous package de FormeGeo, alors que le répertoire associé Coniques est un sous répertoire de FormeGeo.

Comme les classes ou interfaces locales à un package servent à l'implémentation des classes ou interfaces publiques du package, il existe en java un accès privilégiés aux entités ou services entre les entités d'un même package. Le préfixe vide permet un accès privilégié aux entités d'un package. Par exemple,

```
package Nom;
class X
{
        int var;
}
```

La variable var sera accessible en lecture écriture à toute les entités du package Nom;

La visibilité en Java.

Le tableau suivant résume en fonction des mots clefs l'accès aux entités d'une classe (variable ou service) le mot OUI indique la possibilité de l'accès:

	Même classe	Sous classe	Même package	Extérieur
public	OUI	OUI	OUI	OUI
protected	OUI	OUI	OUI	NON
rien	OUI	NON	OUI	NON
private	OUI	NON	NON	NON