Surcharge et Redéfinition.

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.

La surcharge.

La surcharge est la possibilité d'utiliser dans une même classe, le même nom pour déclarer des services différents mais qui ont la même sémantique. Nous avons déjà vu le mécanisme de surcharge lorsque plusieurs constructeurs sont définis pour une classe. Un exemple de surcharge est le suivant:

```
public interface Point2D {
        ......
        public void homothetie(double rapport); // service surcharge
        public void homothetie(Point2D p, double rapport);//service surcharge
        .....
}
```

Dans ce cas, le service homothétie est surchargé puisqu'il est déclaré deux fois dans l'interface Point2D avec des paramètres différents. L'utilisation de la surcharge se fait de la même manière avec des classes.

```
Public class Point2DCartesien implements Point2D {
        ........
        public void homothétie(double rapport) {
         homothetie(new PointCartesion(0,0), rapport) ; // appel à l'autre fonction homothétie de la
        }                                                    // classe Point2DCartesien
        public void homothétie(Point2D p, double rapport){
        ..............
        }
}
```

Dans ce cas, la surcharge permet de simplifier l'appel à la fonction homothétie lorsqu'elle doit être faite avec comme centre l'origine du repère.

L'intérêt de la surcharge est de diminuer le nombre d'identificateur utilisé pour évoquer les services. Mais il faut utiliser la surcharge pour des services qui ont la même sémantique (c'est à dire qui font la même chose). Par exemple, si on a compris la signification de la fonction println elle peut être

utilisée pour afficher n'importe quel type.

La surcharge est autorisée si les services surchargés différent par le nombre de paramètres ou par le type des paramètres à une position donnée. La type de retour du service surchargé n'est pris en compte, de même le fait que le service soit un service de classe ou un service d'instance n'est pas distinctif de la surcharge.

Par exemple,

```
public class ArithmLib{
        public static double min(int i, double d){.....};
        public static double min(double d, int i){.....};
}
```

La surcharge de la fonction min est autorisée car le type des paramètres en position 1 et 2 diffèrent.

En C la surcharge n'est pas possible alors qu'en Java elle est autorisée. Pour évoquer un service en Java, il y a deux manières, soit on utilise le nom de la classe pour sélectionner un service de classe, soit on utilise le type déclaré d'un objet pour sélectionner soit un service d'instance, soit un service de classe.

Type déclaré et type réel.

Le type déclaré d'un objet est le type de la variable qui le référence, ce type peut varier en fonction de la déclaration. Le type réel d'un objet est le type de la classe qui l'a instancié, ce type ne peut jamais changé. Par exemple,

```
FormeFacto f = new Cercle();
Forme g = f;
Cercle c = (Cercle) g;
```

On suppose que Cercle est un sous type de FormeFacto qui lui même est un sous type de Forme.

L'objet crée par new Cercle() est un objet de type réel Cercle(). La variable f est de type déclaré FormeFacto, mais le type réel de l'objet référencé par f est Cercle. La variable g référence le même objet que la variable f, le type déclaré de g est Forme et le type réel est toujours Cercle. Enfin le type déclaré et le type réel de c concorde.

Algorithme de résolution de la surcharge.

L’algorithme de surcharge peut produire trois résultats différents :

- Erreur de compilation si aucune fonction ne correspond à l’appel
- Erreur de compilation si plusieurs fonctions sont à la même distance minimale de l’appel
- Un résultat correct si une seule fonction est à une distance minimale de l’appel.

Soit l'appel o.f(p1,....pn) ou T.f(p1,....pn), on doit maintenant sélectionner la signature du service f qui correspond au mieux à cet appel. On notera T1 **le type déclaré** de p1,.... TN le type déclaré de pn.

1. L'étape initiale consiste à collecter l'ensemble des signatures qui correspondent à un service de nom f. Pour cela, il faut déterminer le type déclaré de O, ce type sera appelé T. Dans le cas, de la sélection d'un service de classe, le type de la classe correspondra au type déclaré T. Une fois le type déclaré, on collecte l'ensemble des signatures des services de nom f en partant du type T et en remontant en considérant tous les services de nom f dans tous les sur types de T et tous les sur....sur type de T jusqu'en qu'il ne soit plus possible de remonter au sur-type. On obtient à cette étape un ensemble de signatures de services qui ont le nom f.
2. La deuxième étape est une étape de filtrage qui consiste à supprimer l'ensemble des services sélectionnés qui n'ont pas la bonne arité (c'est à dire le bon nombre de paramètres) ainsi que les services de signature f qui ne sont pas compatibles avec l'appel. Pour chacune des signatures des fonctions f restantes on procède de la façon suivante, soit une signature f(S1,...Sn). On doit savoir si elle est compatible avec l'appel f(p1,....pn). Pour un Si à une place i, on regarde si il est compatible avec Ti le type déclaré du paramètre pi de l'appel à la place i. **ON CONSIDERE UNIQUEMENT LE TYPE DECLARE Ti du paramètre pi. JAMAIS LE TYPE REEL du paramètre pi, ne sera pris en compte**. Un paramètre Ti est compatible avec un paramètre Si si :
   - si Ti et Si sont deux types primitifs, alors on peut convertir un Ti en un Si, soit par promotion soit par cœrcition
   - si Ti et Si sont deux types non primitifs, alors il faut que Ti soit égal à Si ou bien que Ti soit un sous type de Si, il faut que Ti hétire de Si.
   - Sinon les deux types sont incompatibles.

Une fois cette étape réalisée, il ne reste plus qu'un ensemble de signatures de fonctions qui sont toutes compatibles avec l'appel. **A cette étape, si aucune signature de fonction n'est compatible avec l'appel, il y a une erreur de compilation puisque aucune ne correspond à l'appel.**

3. Après l'étape 2, il faut maintenant choisir la signature de fonction qui est la plus proche de l'appel si bien sûr plusieurs sont possibles. Pour cela, pour chacune signature un évalue une fonction de coût de conversion. Cette fonction de conversion se fait en évaluant pour chaque paramètre le coût de conversion et la fonction associée à la signature est la somme des coûts de conversion. Sans entrer dans les détails, une cœrcition à un coût supérieur à une promotion, si le type de la signature et le type de l'appel sont les mêmes le coût de conversion est nul, en cas de compatibilité par héritage, le coût dépend de la profondeur de l'arbre d'héritage. Après cette étape, chacune des signatures sélectionnées est compatible avec l'appel et un coût de conversion lui est associé.
4. Si il existe une seule signature de coût de conversion minimale, c'est à dire si il existe une unique signature de fonction qui est la plus près de l'appel, **alors le résultat de l'algorithme de sélection de la surcharge produit une unique signature f(S1,...Sn).**

   Si par contre il existe plusieurs signatures qui sont à une même distance minimale de l'appel, la résolution de la surcharge est considérée comme ambiguë est une erreur de compilation est produite.

Exemple des différents résultats sélectionnés par l'algorithme de résolution de la surcharge.

public class A{}….. public class B extends A{……}

public class Exemple {

public double f(int n, double d){……}

public double f(double d, int t) {…….}

public double f(A a){……}

public double f(B b){…….}

}

Différents exemple d'appels et la signature sélectionnée

Exemple.f(3) ;// il n'existe aucune signature d'arité 1 avec un type primitif compatible avec int.

// un type primitif comme le type associé à 3 ne peut être compatible avec un type

// d'instance……..**ERREUR DE COMPILATION, appel incompatible**

int i = 3 ; int j = 2 ; Exemple.f(i,j) ;

Sur cet appel deux signatures pour la fonction f sont compatibles, il s'agit des signatures double f(int n, double d) , double f(int n, double d). Ces deux fonctions ont le même coût de conversion par rapport à l'appel f(int, int). Leur coût est égal à une promotion de int à double. Dans ce cas, le résultat produit par l'algorithme de sélection est **ERREUR DE COMPILATION résolution ambiguë**, car plusieurs signatures de fonction sont à une distance minimale de l'appel.

Int i, double d ; Exemple.f(i,d) ; // La signature sélectionnée est **double f(int n, double d).**

A b = new B() ; Exemple.f(b);

Le type réel de b est B mais son type déclaré est A, comme la signature sélectionnée ne dépend pas du type réel des paramètre, la signature sélectionnée par l'algorithme est bien **double f(A a).**

La redéfinition.

La redéfinition est la possibilité d'utiliser exactement la même signature pour définir un service dans un type et dans un sous type. Le type de retour du service doit être le même, mais la visibilité peut changer.

Un exemple, qui mêle surcharge et redéfinition.

public interface Forme {…….}

public class Carre implements Forme {……..}

public class Cercle implements Forme{…….}

public class Dessin

{

public void dessiner(Forme f) {……} // surcharge de dessiner dans la classe Dessin

public void dessiner(Carre c) {…….} // surcharge de dessiner dans la classe Dessin

}

public class DessinColorie extends Dessin

{

Public void dessiner(Forme f) {……..} // redéfinition du service dessiner de la classe Dessin

// dans la classe DessinColorie

}

Exécution de service.

Algorithme de sélection de services, permet d'exécuter un service dynamiquement. Soit l'appel

o.f(p1,....pn),

on parle de liaison dynamique ou de polymorphisme simple. Le mot polymorphisme simple, signifie que le type réel de l'**objet o** participe à la sélection du service f , et que le type réel des paramètres n'intervient pas, d'où le qualificatif simple.

Algorithme de sélection des services :

Soit le code

Tdeclare **o** = new Treel() ; // Type Déclare de l'objet **o** Tdeclare ;

// Type Réel de l'objet **o** Treel ;

o.f (p1,…pn) ;

L'algorithme de sélection des services doit trouver le service à exécuter en fonction de l'appel. Cet algorithme procède en deux étapes :

- **Etape 1 :** ***Pendant la compilation***, l'algorithme de résolution de la surcharge est appelé. Cet algorithme ne considère que le type Tdeclare de l'objet **o est fourni une signature unique d'un service f.**
- **Etape 2 :** ***Pendant l'exécution,*** à partir du type Treel de **l'objet o,** on cherche le service f qui a **exactement la même signature** que celle sélectionnée à l'étape 1. On exécute le premier service rencontré, si il n'existe pas de tel service dans la classe Treel, on passe dans la classe sur type de Treel, si il en existe pas…. On remonte encore… dans la sur sur classe. Il existe obligatoirement un service qui correspond à cette signature car sinon, il y aurait eu une erreur de compilation.

Comme nous allons le voir, dans les exemples suivants : L'algorithme de sélection des services permet **lorsqu'il y a redéfinition** du service **de préserver le comportement** d'un objet.

Principe de préservation du comportement d'un objet. Pour un objet **o** donné, l'appel o.f (p1,… pn) doit produire le même comportement (appelé le même code) indépendamment du type déclaré de o.

Exemple de sélection de services.

On reprend les définitions des types Forme, Carre, Cercles….. comme vu au début de ce chapitre. Nous allons voir différents appels et les différentes étapes :

DessinColorier dc = new DessinColorie() ;
Forme f = new Carre() ;
dc.dessiner(f) ;

Type Réel de **dc** = DessinColorie ; Type Déclaré de **dc** = DessinColorie;

Type reel de **f** = Carre ; Type Déclaré de **f** = Forme ;

Etape 1 : 2 signatures de fonctions sont candidates à partir du Type Déclaré de dc qui est DessinColorie et de son sur-types Dessin. Il s'agit de **public void dessiner(Carre c)** et **public void dessiner(Forme c)**. En fonction du type déclaré de f qui est forme, la plus près en terme de cout de conversion est : **public void dessiner(Forme c). On voit que le type réel de f qui est Carre n'intervient.**

Etape 2 : En partant du type réel de dc qui est **DessinColorie,** on recherche le service dont la signature est public void dessiner(Forme c) dans la classe DessinColorie. Il existe c'est celui là qui est exécuté. **public void dessiner(Forme c) DEFINIE DANS LA CLASSE DESSINCOLORIE.**

```
Dessin dc = new DessinColorie() ;
Forme f = new Carre() ;
dc.dessiner(f) ;
```

Type Réel de **dc** = DessinColorie ; Type Déclaré de **dc** = Dessin;
Type reel de **f** = Carre ; Type Déclaré de **f** = Forme ;

Etape 1 : Exactement la même signature que celle de l'appel précédent car elles sont toutes les deux déclarés dans Dessin.

Etape 2 : Comme le type réel de dc ne change pas et que la signature sélectionnée à l'étape 1 est la même on obtient le même résultat.
On voit que sur cette exemple, le service **dessiner(Forme)** est redéfinie par la classe DessinColorie, le comportement est bien préservé que l'on considère dc comme un **Dessin** ou comme un **DessinColorie**.

Maintenant nous allons rajouter une nouvelle classe dans la hiérarchie des Formes et des dessins

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

public class DessinColorieBis extends DessinColorie
{
        public void dessiner(Cercle c) {……..}
}
```

Comment peut-on qualifier le service dessiner(Cercle) de la classe DessinColorieBis. Il ne s'agit pas d'une redéfinition car il n'existe pas au dessus de la hiérarchie de classe, un service qui a cette signature. Par contre, nous savons qu'avec la propriété de l'héritage de type, il existe de manière implicite au niveau de la classe DessinColorieBis, les services **dessiner(Carre c)** et **dessiner(Forme c).** Donc le service **void dessiner(Cercle c)** est une surcharge au niveau de la classe DessinColorieBis, des services **dessiner(Carre c)** et **dessiner(Forme c).**

```
DessinColorierBis dc = new DessinColorieBis() ;
Cercle c = new Cercle() ;
dc.dessiner(c) ;
```

Type Réel de **dc** = DessinColorieBis; Type Déclaré de **dc** = DessinColorieBis;
Type reel de **c** = Cercle ; Type Déclaré de **c**=Cercle;

Etape 1 : 3 signatures de fonctions sont candidates à partir du Type Déclaré de dc qui est DessinColorieBis et de ses sur-types DessinColorie et Dessin. Il s'agit de **void dessiner(Carre c)** et **void dessiner(Forme c) et dessiner(Cercle c)**. La signature **dessiner(Carre c)** n'est pas compatible car les Carre ne sont pas des Cercles. Il reste deux signatures, la signature de coût minimale est **dessiner(Cercle c), car elle correspond au type déclaré de l'appel.**

Etape 2 : En partant du type réel de dc qui est **DessinColorieBis,** on recherche le service dont la signature est public void dessiner(Cercle c) dans la classe DessinColorieBis. Il existe c'est celui là qui est exécuté. **public void dessine(Cercle c) DEFINIE DANS LA CLASSE DESSINCOLORIEBIS.**

```
Dessin    dc = new DessinColorieBis() ;
Cercle c   = new Cercle() ;
dc.dessiner(c) ;
```

Type Réel de **dc** = DessinColorieBis; Type Déclaré de **dc** = Dessin;
Type reel de **f** = Cercle; Type Déclaré de **f** =Cercle;

Etape 1 : 2 signatures de fonctions sont candidates à partir du Type Déclaré de dc qui est Dessin. Il s'agit de **void dessiner(Carre c)** et **void dessiner(Forme c)**. La signature **dessiner(Carre c)** n'est pas compatible car les Carre ne sont pas des Cercles. Il reste une seule signatures, la signature **dessiner(Forme) est compatible avec l'appel car tous les cercles sont des Forme.**

Etape 2 : En partant du type réel de dc qui est **DessinColorieBis,** on recherche le service dont la signature est public void dessiner(Forme c) dans la classe DessinColorieBis, il n'en existe pas donc on passe au type du dessus qui est DessinColorie. Il existe une fonction qui correspont à la signature selectionnée à l'étape 2 c'est celui là qui est exécuté. **public void dessine(Forme f) DEFINIE DANS LA CLASSE DESSINCOLORIE.**

Sur cet exemple, on voit que le résultat de l'appel dépend du type déclaré de dc. Cela n'est pas en contradiction avec le principe de conservation du comportement car le service
dessiner(Cercle)
n'est pas une redéfinition. Il aurait fallu le déclarer dans la classe Dessin.

Un autre exemple rapide.

class X{} ; class Y extends X {…} ; class Z extends Y{….}

```
public class A {
        public void f(Y y) {System.out.println(" A::Y");}
}
public class B extends A {
        public void f(Y y) {System.out.println(" B::Y");}
}
public class C extends B {
        public void f(X x) {System.out.println(" C::X");}
```

```
    public void f(Z z) {System.out.println(" C::Z");}
}
```

Soit le code suivant:

C c = new Z() ; Z z = new Z();

B b = c; Y y = z;

A a = c ; X x = z;

On va regarder les différents appels :

1. **c.f(z)** :
   - Etape 1 : f(Y), f(X),f(Z). Celle de coût minimum par rapport à l'appel est f(Z)
   - Etape 2 : C ::Z
2. **c.f(y)**
   - Etape 1 : f(Y), f(X),f(Z). Celle de coût minimum par rapport à l'appel est f(Y)
   - Etape 2 : B::Y
3. **c.f(x)**
   - Etape 1 : f(Y), f(X),f(Z). Celle de coût minimum par rapport à l'appel est f(x)
   - Etape 2 : C::X
4. **b .f(z)**
   - Etape 1 : f(Y). Appel compatible
   - Etape 2 : B::Y
5. **b .f(y)**
   - Etape 1 : f(Y). Appel compatible
   - Etape 2 : B::Y
6. **b .f(x)**
   - Etape 1 : f(Y). Appel incompatible, tous les x ne sont pas des y**.**
7. **a .f(z)**
   - Etape 1 : f(Y). Appel compatible
   - Etape 2 : B::Y
8. **a .f(y)**
   - Etape 1 : f(Y). Appel compatible
   - Etape 2 : B::Y
9. **a .f(x)**
   - Etape 1 : f(Y). Appel incompatible, tous les x ne sont pas des y**.**