COURS PROGRAMMATION 2

(LICENCE INFORMATIQUE Université de Bordeaux)

Table des matières

POO généralités.....	2
Structure d'une classe.....	2
Constructeur:	2
L'objet:	3
Création et définition d'une classe.....	4
Spécification et implémentation.....	5
Objets, classes, interfaces.....	6
L'héritage de type.....	7
Les propriétés de l'héritage de type.....	8
Relation entre classes et interface.....	9
Interface et factorisation de code.....	11
Héritage de Type et décomposition ensembliste.....	13
Héritage de Code et classe abstraite.....	16
Héritage de code.....	16
Propriété de l'héritage de Code.....	17
Le mot clef protected.....	19
Classe Abstraite.....	20
Surcharge et Redéfinition.....	23
La surcharge.....	23
Type déclaré et type réel.....	24
Algorithme de résolution de la surcharge.....	25
Polymorphisme, la classe Object, les package et la visibilité en Java.....	25
Polymorphisme.....	25
Le DownCast	29
La Classe Object.....	29
Factorisation de Code.....	31
La notion de package.....	33
La visibilité en Java.....	34
Exception en Java.....	34

POO généralités.

Structure d'une classe.

Lorsqu'une déclaration est précédée du mot clef **static**, cette information concerne la classe, tous les services définis dans la classe auront accès à cette information. On parle alors de services ou de variables de classes (par opposition aux services ou variables d'instances).

Static : Contexte de classe

Rien : Contexte d'objet ou d'instance.

```
Public class Rationnel {  
    private int numerateur; // variable d'instance spécifique à un objet particulier  
    private int denominateur; // variable d'instance spécifique à un objet particulier  
    public void reduction () { // service d'instance spécifique à un objet particulier  
        .....  
    }  
}
```

Constructeur:

Syntaxiquement le constructeur est un service qui porte le même nom que la classe et qui ne possède pas de type de retour. Le rôle du constructeur est d'initialiser les valeurs des variables d'instances d'un objet. On dit aussi que le constructeur donne l'état initial d'un objet.

Remarque: Pour construire un objet (instancier), on peut avoir besoin d'information en provenance de l'extérieur. Pour cela, il est nécessaire de pouvoir disposer de plusieurs constructeurs différents, on parle alors de surcharge des constructeurs.

Exemple:

```
Public class Rationnel {  
    private int numerateur;  
    private int denominateur;  
    public Rationnel() // constructeur sans paramètre.  
    {  
        numerateur = denominateur = 1; // on crée l'élément neutre.  
    }  
}
```

```

public Rationnel(int num, int deno) // un autre constructeur qui surcharge le précédent.
{
    numérateur = nu;
    dénominateur = deno;
}

```

Une classe définit un nouveau type qui peut être instancier (créer des objets à partir d'une classe). Pour créer un objet, il faut utiliser le mot clef **new**. Par exemple,

```

Rationnel r = new Rationnel();
Rationnel r1 = new Rationnel(2,3);

```

La création d'un objet se fait en deux temps:

1. Réservation de la place mémoire nécessaire à cet objet (en fonction du type de l'objet).
2. Appel à un constructeur en fonction des paramètres (résolution de la surcharge). Le constructeur est appelé pour initialiser l'objet (lui donner son état initial).

Sur cet exemple, le constructeur sans paramètre est appelé, la variable r référence alors un objet rationnel qui a pour valeur l'élément neutre. La variable r1 référence maintenant un objet rationnel dont la valeur initiale est 2/3.

L'objet:

Un objet est instance d'une unique classe, c'est à dire qu'il est crée par une classe. Comme une classe définit un type, **un objet au moment de sa création aura comme type réel** le type de la classe qui l'a instancié. Par exemple, lorsqu'on écrit new Rationnel(), on vient de créer un nouvel objet instance de la classe Rationnel, **le type réel de cet objet est définitivement Rationnel**.

Un objet a un **comportement**, c'est à dire l'ensemble des services d'instances qui ont été définis par la classe qui a créé le nouvel objet. Pour évoquer le comportement associé au **unService** d'un objet **O** on utilise la syntaxe:

O.unService(p1,...pn);

On dit que l'objet O est le **sélecteur ou aussi le récepteur du service**. On demande à l'objet O de **sélectionner et d'exécuter** le service associée au mot clef **unService** en fonction des paramtres p1,..pN. On peut remarquer que comme l'objet participe activement à la sélection du service, on peut avoir des services de même nom mais dans des classes différentes. Comme le nombre et le type des paramètres interviennent aussi dans la sélection du service, on peut avoir plusieurs fois le même nom de service dans une même classe mais avec des paramètres de type et/ou de nombre différents. On parlera alors de **surcharge**.

Lorsqu'on évoque un service d'instance sur un objet O. Le service s'exécute dans l'environnement de l'objet. Cet objet est présent dans le code du service, il est appelé **this** (ce qui correspond au **self** vu en C). Par contre, pour les services de classe, l'objet this n'existe pas, puisqu'il ne fait pas partie de la selection.

Un objet a aussi un état, il y a deux types d'état:

1. L'état logique. L'état logique d'un objet dépend de son état initial (celui donné par le constructeur au moment de l'instanciation de l'objet) et de la succession des services qu'il a

rendu. Les services appliqués sur un objet font donc évoluer son état.

2. L'état physique. C'est la valeur des variables d'instances d'un objet.

Par exemple, si on considère la création d'un objet Rationnel par $r = \text{new Rationnel}(4,8)$. L'état physique est représenté par le couple (4,8), alors que l'état logique consiste à dire que r est dans l'état $\frac{1}{2}$. Si on fait une réduction de r par le service $r.\text{reduire}()$. L'état physique change, car maintenant il est représenté par le couple (1,2) mais son état logique n'a pas changé.

Création et définition d'une classe.

Une classe permet de définir un nouveau type, d'instancier des objets, d'appeler les services sur cet objet. Elle se fait en plusieurs étapes de réflexion.

1. **Définition de l'interface fonctionnelle publique.** Il s'agit de trouver l'ensemble des services que pourra rendre la classe. Cette interface fonctionnelle est constituée des services publiques de classe et des services publiques d'instances. Il n'est pas toujours facile de distinguer entre un service public de classe ou un service public d'instance. Prenons par exemple, la division de deux rationnels.
 - Service de classe: $\text{Rationnel res} = \text{Rationnel.division}(p,q)$; On doit interpréter cet appel, comme demander à la classe Rationnel de rendre le service division en considérant deux rationnels qui sont p et q . Le service de classe division rend alors un nouvel objet de type Rationnel qui est le résultat de la division.
 - Service d'instance: $\text{Rationnel res} = p.\text{division}(q)$; Dans ce cas, on demande à l'objet p de rendre le service associé à la division en prenant comme paramètre l'objet q , l'objet p a en charge de créer un nouvel objet, de faire le calcul et de rendre le résultat.

Sur cet exemple, il n'y a aucune raison de demander à un objet de rendre ce service, il est donc préférable d'en faire un service de classe. Une autre différence essentielle, entre service de classe et service d'instance est la suivante:

- Les services de classes sont décidés à la compilation.
- Les services d'instances sont vérifiés à la compilation et décidés à l'exécution.

1. **Implémentation de l'interface fonctionnelle publique.** Une fois l'interface fonctionnelle publique de la classe définie, il est nécessaire de l'implémenter. Il faut se rappeler que l'implémentation est un choix temporaire qui peut évoluer au fil du temps ou même changer radicalement. Ce n'est pas le cas, de l'interface fonctionnelle publique qui elle doit respecter la compatibilité ascendante tout au long de son évolution. Les choix d'implémentation portent alors sur :
 - Le choix des variables d'instances ou de classes qui vont permettre de définir l'implémentation.
 - Le choix des services privés (classes/services) qui vont permettre d'assurer une bonne modularité du code (éviter la duplication, permettre la factorisation, accroître la lisibilité).
1. **Ecriture du code des constructeurs:** Une fois, les variables d'instances choisies, il faut maintenant choisir les constructeurs, c'est à dire comment définir l'état d'un objet au moment de sa création et surtout quelles sont les informations nécessaires pour pouvoir créer l'état.

Public Rationnel() // création de l'élément neutre.

```
Public Rationnel(int n) // création d'un rationnel numérateur = n, dénominateur =1
```

```
Public Rationnel(int n, int d) // création d'un rationnel numérateur = n, dénominateur =d
```

Spécification et implémentation

Dans tous les cas de figure, l'implémentation d'une classe doit être conforme à la spécification. Par exemple, pour les rationnels est ce que l'on utilise la définition suivante. Un rationnel est un couple d'entiers relatifs, tel que le dénominateur ne soit jamais nul. Dans ce cas, le code du constructeur est alors le suivant :

```
Public Rationnel(int n, int q) {  
    this.numerateur = n; // on utilise this  
    this. denominateur = q;  
}
```

Par contre si les spécifications pour un rationnel sont les suivantes: Un rationnel est un couple d'entiers relatifs, tel que le dénominateur ne soit jamais nul et tel que le numérateur et le dénominateur sont premiers entre eux. Dans ce cas, le code du constructeur est alors le suivant :

```
public Rationnel(int n, int q) {  
    this.numerateur = n; // on utilise this  
    this. denominateur = q;  
    this.reduire().  
}  
private void reduire(){  
    int n = //on calcule le pgcd de numérateur et de dénominateur.  
    this.numerateur /= n;  
    this.denominateur /=n;  
}
```

En effet, pour être fidèle aux spécifications, il est nécessaire de réduire, le rationnel au moment de sa création. Mais les spécifications influent sur l'interface fonctionnelle publique d'une classe. Si on reprend l'exemple de Rationnel, dans le second cas, le service d'instance réduire est défini pour éviter la duplication de code, puisque toutes les opérations qui feront évoluer un rationnel devront appeler réduire. Dans le cas 2, ce service ne peut être que privé puisque par définition un rationnel est réduit et il serait stupide de mettre ce service dans l'interface fonctionnelle publique.

Par contre, dans le premier cas, où il n'est pas obligatoire qu'un rationnel soit réduit, il est nécessaire que ce service fasse partie de l'interface fonctionnelle publique.

Discussion sur le positionnement d'un service.

Comme nous l'avons vu précédemment le service réduire a besoin d'un service qui est le calcul du PGCD de deux nombres entiers relatifs. Plusieurs solutions sont envisageables pour positionner et donner une visibilité au service PGCD.

1. La première solution consiste à écrire directement le code du calcul du PGCD dans le code du service réduire. Cette solution est mauvaise car elle engendre de la duplication de code et nuit à la lisibilité du code.
2. La deuxième solution consiste à en faire un service privé d'instance à l'intérieur de la classe Rationnel.

```
private int pgcd();
```

Dans ce cas, on augmente la lisibilité du code, mais il peut tout de même exister de la duplication de code au sein de l'application, car si on a besoin du calcul du PGCD à l'extérieur de la classe Rationnel, il faudra écrire le code.

3. La troisième solution consiste à en faire un service public d'instance à l'intérieur de la classe Rationnel.

```
public int pgcd();
```

On diminue maintenant la duplication de code, mais on augmente le couplage, puisque pour calculer le pgcd des entiers p,q, nous sommes obligés de créer un nouveau rationnel:

```
Rationnel tmp = new Rationnel(p,q);  
int pgcd = tmp.PGCD();
```

4. La quatrième solution est d'en faire un service public de classe de la classe Rationnel
static public int pgcd(Rationnel r);
static public int pgcd(int a, int b);

Le premier prototype n'est pas plus intéressant que la solution 3, car il est toujours nécessaire de créer un objet de type Rationnel, pour obtenir le calcul du PGCD. Par contre, le deuxième prototype qui lui considère un couple d'entier est plus efficace car il ne nécessite pas la création d'un nouveau rationnel, pour calculer le PGCD de deux entiers. Par contre, il est difficile de comprendre pourquoi le calcul du pgcd serait un service public de la classe Rationnel, il n'y a pas de réelles justifications, simplement le fait que le service réduire à besoin du calcul du PGCD, autant le prototype est le bon autant le positionnement de ce service dans la classe Rationnel est problématique.

1. La dernière solution consiste à déplacer le service de calcul du PGCD dans une classe bibliothèque.

```
Public class ArithmetiqueLib {  
    static public int PGCD(int a, int b).....  
    .....  
}
```

Cette dernière solution est la bonne, car le service est correctement positionné et il ne nécessite aucun couplage pour son utilisation.

Objets, classes, interfaces.

Une **interface** définit un type en déclarant seulement l'interface fonctionnelle publiques des services d'instance. Elle ne donne pas de code au service qu'elle déclare. Le seul code qui peut-être présent dans une interface est la déclaration de constante en utilisant les mots final static. Par exemple,

```
public interface Point2D  
{  
    public static final PI = 3.14159;  
    public void homothetie(double rapport);  
    public void rotation(double angle);  
    public void translation(Point2D debut, Point2D fin);  
    .....  
}
```

L'interface peut-être considérée comme l'implémentation d'un type abstrait de donnée. Elle ne fait que décrire un ensemble de services indépendamment de toute implémentation.

Pour définir une interface, il faut se poser la question de quelles sont les entités présentes dans le domaine du problème. Donner un nom à une interface ou à un type et la première étape qui permet d'obtenir une modularité correcte. Une fois que l'entité peut-être nommée, il est nécessaire de trouver l'ensemble des services que peut rendre ce type. A ce stade aucune considération d'implémentation ne doit être prise en compte, mais il faut que l'ensemble des services publics, soit complet et aussi minimal. Par exemple, si on considère l'interface Point2D, on ne peut pas supposer que l'implémentation de cette interface se fera en utilisant les coordonnées cartésiennes ou les coordonnées polaires. Par contre, cette interface n'est pas complète car on ne peut obtenir les coordonnées d'un point. Il faut donc la compléter en rajoutant les services publics suivants:

```
public double getX();
public double getY();
public double getRho();
public double getTheta();
```

L'héritage de type.

Une interface définit un type, l'héritage de type est une relation entre un type et un ensemble de sous type. La relation d'héritage de type se traduit comme une relation « est une sorte de » ou bien encore comme une relation « est un ». Elle établit une hiérarchie entre les types, si un mammifère est une sorte d'animal alors le type mammifère est un sous type du type Animal.

Par exemple,

```
public interface Vehicule
{
    public Point2D deplace(Point2D depart);
}
```

L'interface Vehicule déclare un nouveau type « Véhicule » ainsi que le service deplace. On peut dire qu'un véhicule a la possibilité de se déplacer en allant d'un point à un autre.

Maintenant si je considère l'interface suivante:

```
public interface VehiculeTerrestre
{
    public void roule();
    public void freine();
    public void demarre();
}
```

Cette déclaration crée un nouveau type qui est le type VéhiculeTerrestre avec comme services: roule, freine, demarre. On peut se poser la question de savoir si il n'existe pas une relation entre le type Vehicule et le type VehiculeTerrestre. Par exemple, est ce qu'un véhiculeTerrestre n'est pas une sorte de véhicule? Est ce qu'un véhiculeTerrestre n'est pas une spécialisation d'un véhicule.

Dans ce cas, on peut considérer que le Véhicule Terrestre n'est qu'un véhicule particulier, qui se déplace sur la terre ferme et dont le déplacement peut se faire en utilisant les services demarre, freine et roule. Si on considère cette possibilité, comme un véhiculeTerrestre est un véhicule alors

il doit fournir les mêmes services qu'un véhicule, c'est à dire qu'il doit se déplacer.

En java, l'héritage de type entre interface se fait en utilisant le mot clef extends. Une interface peut-être n sous type de plusieurs autres types (interfaces). Exemple de déclaration,

```
public interface VehiculeTerrestre extends Véhicule
{.....}
```

Les propriétés de l'héritage de type.

Si une type A hérite d'un type B, alors on a les deux propriétés suivantes:

1. Principe de substitution. On peut adresser les objets du sous type en utilisant le sur-type. Par exemple,

```
public interface A extends B
```

Alors on peut écrire A a = ... et B b = a; dans ce cas on adresse un objet de type A en utilisant une référence de type B. **On dira alors que le type A est compatible avec le type B.**

2. Interface fonctionnelle publique de A contient tous les services de l'interface fonctionnelle de B . IFP(B) est incluse dans IFP(A). Tout ce que peut faire un objet de type B, un objet de type peut le faire et il peut aussi en faire plus ou le faire différemment. En reprenant l'exemple précédent, un véhicule peut se **déplacer donc un véhicule terrestre peut aussi se déplacer même si le service deplace n'est pas explicitement déclaré dans l'interface véhicule.**

```
VehiculeTerrestre vt = new .....(....); // instantiation d'une classe.
```

```
vt.demarrer(); // service décrit dans véhicule terrestre.
```

```
Vehicule v = vt; // premiere propriété (substitution);
```

```
v.deplace(); // service déclaré dans véhicule;
```

```
vt.deplace(); // utilisation de la seconde propriété de l'héritage.
```

Sur cet exemple, v et vt sont des variables qui référence le même objet physique, comme un véhicule terrestre est un véhicule, il est normal qu'il puisse faire la même chose qu'un Véhicule. **IL N'Y A PAS D'HERITAGE EN RESTRICTION DES SERVICES.**

Héritage Multiple.

Une interface peut-être un sous-type de plusieurs interfaces. On parle alors d'héritage multiple de type. La syntaxe est alors la suivante:

```
public interface X extends Y,Z.
```

Dans ce cas, le type X est un sous type de Y et aussi un sous type de Z. On peut donc considérer à un moment un même objet physique comme étant un objet de type Y et à un autre moment considéré le même objet physique comme étant de type Z.

```
X x = new.... (); // instantiation d'une classe
```

```
Y y = x; // on considère x comme étant un Y
```

```
Z z = x; // on considère le même objet mais comme un Z.
```

Un autre exemple:

```
public interface VehiculeNautique extends Vehicule {
    public void navigue();
}

public interface Voiture extends VehiculeTerrestre {
    public void changerRoue()
}

public interface VoitureAmphibie extends Voiture, VehiculeNautique {
}
```

Dans ce cas, la voitureamphibie cumule le comportement de VehiculeNautique, de Voiture, de vehiculeTerrestre et aussi celui de Vehicule. On peut donc, la faire rouler, freine, demarrer, naviguer et changer sa Roue. On peut aussi la convertir en tout ces types pour la manipuler.

La relation d'héritage est une relation transitive, si A herite de B et si B herite de C alors A hérite de C. C'est une relation anti-symetrique si A hérite de B alors B ne peut pas hériter de A. La relation d'héritage est donc une relation d'ordre partiel.

Relation entre classes et interface.

Rappel: une classe définit un type, on parlera alors de type concret car la classe permet de créer des objets, ce n'est pas le cas d'une interface qui elle ne fait que définir une interface fonctionnelle publique.

Une classe peut être considérée comme l'implémentation d'un type abstrait, c'est à dire qu'elle a en charge de fournir le code nécessaire à l'implémentation des services présentées dans l'interface fonctionnelle publique. Que cette interface soit celle de la classe elle-même ou celle d'une interface.

En Java, une classe peut implémenter, concrétiser, réaliser une interface. Par exemple, on peut définir l'interface du type abstrait Point2D.

```
Public interface Point2D {
    public void homothétie(double rapport);
    public void translation(Point2D p, Point2D q);
    public void rotation(double angle);
    public double getX();
    public double getY();
    public double getRho();
    public double getTheta();
}
```

Le type Point2D est un type abstrait car il est indépendant de tout choix d'implémentation. En se répétant, il ne fait que présenter un ensemble de service que peut rendre ce type.

Maintenant, il faut pouvoir concrétiser le type Point2D en faisant un choix d'implémentation pour réaliser ce type. Par exemple, on peut considérer les coordonnées cartésiennes pour représenter un point en dimension 2. La relation de réalisation entre une interface et la classe qui la réalise se fait en utilisant le mot clef **implements**. **La relation de réalisation entre une classe et les interfaces qu'elle concrétise est une relation type/sous type.**

```
public class Point2DCartesien implements Point2D
{
    private double abs;
    private double ord;

    public void translation(Point2D p, Point2D q)
    {
        this.abs += q.getX() - p.getX();
        this.ord += q.getY() - p.getY();
    }
    // il faut ajouter le code de tous les autres services présents dans l'interface fonctionnelle.
}
```

Pour qu'une classe puisse être instanciée en utilisant new, il faut que l'ensemble des services de l'interface fonctionnelle publique soient TOUS IMPLEMENTES.

Les variables d'instances représentent la structure des objets instances de la classe. Par exemple, tous les objets dont le type réel est Point2DCartesien, auront tous les champs abs et ord. Ces champs appartiennent uniquement à l'objet, il faut donc utiliser l'objet pour pouvoir y accéder.

Une autre réalisation de l'interface Point2D aurait pu être faite par la classe Point2DPolaire qui serait déclarée de la manière suivante:

```
public class Point2DPolaire implements Point2D
{
    private double theta;
    private double rho;
    .....
}
```

Ainsi, le type abstrait Point2D a deux réalisations différentes qui sont donnés par les types concrets Point2DPolaire et Point2DCartesien.

En java, il n'est pas obligatoire qu'une classe réalise une interface, en effet la déclaration d'une classe en Java, suffit pout déclarer à la fois un type abstrait et un type concret. Dans ce cas, le type abstrait et le type concret sont confondu. Pour obtenir un objet, il faut utiliser une classe.

En résumé,

Une interface représente un type abstrait de donnée,

Une classe est la concrétisation d'un type abstrait de donnée on parle alors de type concret.

Un objet est instance d'une unique classe, le type réel d'un objet est le type de la classe qui a instanciée l'objet.

Tous les objets qui ont le même type réel ont la même structure et le même comportement (ensemble des services de l'interface fonctionnelle publique), c'est à dire la même implémentation des services.

Une classe est donc une fabrique d'objet qui définit la structure et le comportement de ces instances.

Sans faire la distinction entre type réel et type concret, un type est un ensemble d'objets qui ont une structure commune et une interface fonctionnelle commune.

Une interface peut-être un sous type d'une ou plusieurs interfaces.

Une classe peut-être la réalisation d'une ou plusieurs interfaces, et elle ne peut hériter du code que d'une seule classe. La relation de réalisation implique la relation d'héritage de type.

En java, il y a de l'héritage multiple de type et de l'héritage simple de code.

Interface et factorisation de code.

Une interface permet de définir un type et son comportement sans supposer une quelconque implémentation, il est donc possible qu'une même interface est plusieurs réalisations. C'est par exemple, le cas de Point2D qui est une interface et qui a par exemple deux réalisations, Point2DCartesien et Point2DPolaire.

Soit par exemple, la classe Client

```
public class Client {
    static public void traiter(Point2D [] t)
    {
        for(int i = 0; i < t.length; i++)
            t[i].homothetie(r);
    }
}
```

Cette classe ne travaille que sur le type abstrait Point2D, elle ne connaît pas les concrétisations de ce type. Elle peut donc travailler avec un ensemble de Point2D sans connaître l'implémentation. Il est donc possible d'écrire le code suivant.

```
Public class App1
{
    static public void main(String [] arg)
```

```

{
    Point2D [] tab = new Point2D[10];
        // on crée simplement un tableau de 10 références à des Point2D, mais il y a aucune
        // instanciation de Point2D (qui est non instanciable)

    for(int i = 0; i < 5; i++)
        tab[i] = new Point2DPolaire(i,j);
        // cette ecriture est possible car Point2DPolaire est un sous      type de Point2D,
        // c'est la première propriété de l'héritage de type (principe de substitution). On a donc dans tab
        // des objets dont le type déclaré est Point2D et le type réel est Point2DPolaire.

    for(int i = 5; i < 10; i++)
        tab[i] = new Point2DCartesien(i,j);
        // cette ecriture est possible car Poin2DCartesien est un sous type de Point2D.
        Client.traiter(tab); // maintenant on peut traiter un tableau qui contient à la fois des
        Point2DCartesien et des Point2DPolaire.
}

```

L'utilisation de l'héritage de type permet donc d'étendre le champs d'une application sans avoir besoin de modifier le client. Sur cette exemple, si une troisième concrétisation de Point2D était fournie, on aurait pas besoin de reprendre le Code de client et il pourrait donc fournir des services pour ce nouveau type.

La factorisation de code qui est fournit par l'utilisation de l'héritage de type, concerne les clients de la hiéarchie de type. Nous verrons plus tard que l'utilisation de l'héritage de code permet de fournir une factorisation du code à l'intérieur de la hiérarchie de classe.

Un autre exemple est donnée par le schéma suivant.

Nous pouvons considérer que le jeu ne connait que les personnages et ne dépend pas des personnages concrets qui sont Pacman et SuperMario.

Le code suivant :

```

class Jeu
{
    public void anime(Personnage p)
    {
        p.deplace();
        p.anime();
    }
}

```

Pour que ce mécanisme fonctionne, il faut que l'interface personnage contiennent le comportement commun à Pacman et à SuperMario. En utilisant la propriété 2 de l'héritage de type, nous avons IFP(Personnage) qui est incluse dans l'interserction de IFP(Pacman) et de IFP(SuperMario).

Mais cela n'est pas suffisant, il faut que les Pacman et les SuperMario conservent leur comportement quand ils sont vus comme des personnages. Par exemple, si on demande à un personnage de se dessiner, il faut que ce personnage se dessine comme un Pacman, si il a été créé par la classe pacman, ou comme un superMario si il a été créé par la classe SuperMario. Il faut donc que la propriété suivante soit vrai.

Un objet doit conserver le comportement qui lui a été donné au moment de son instanciation, indépendamment du type qui le référence. Un pacman doit se comporter comme un pacman tout au long de sa vie, même si on ne le perçoit comme un personnage.

Héritage de Type et décomposition ensembliste.

Nous avons vu précédemment comment un type peut être associé à un ensemble d'objets ayant un comportement commun, si on passe au niveau des classes on doit aussi prendre en compte la structure des objets. Comme la relation d'héritage de type, se traduit par « est un » ou « est une sorte de ».

La relation type sous-type peut se traduire sous la forme d'inclusion d'ensemble.

Si A est un sous type de B, alors l'ensemble des objets A est inclus dans l'ensemble des objets B. On dira que A est un sous ensemble de B.

Si A est un sous type de B et de C (héritage multiple), alors A est un sous ensemble de B et A est aussi un sous ensemble de C, donc A est un sous ensemble de B inter C. L'héritage Multiple est une cumulation des comportements des sur types et une intersection d'ensemble.

Prenons par exemple, les spécifications suivantes:

Les oiseaux mangent et se reproduisent. Les mammifères sont des animaux qui allaitent et qui se reproduisent en accouchant. Les oiseaux sont des animaux qui volent et qui se reproduisent en pondant des oeufs. Parmi, les animaux les herbivores mangent de l'herbe, les carnivores mangent de la viande, les omnivores ne sont ni des herbivores ni des carnivores mais ils mangent parfois comme un herbivore et parfois comme un carnivore. Les poules sont des oiseaux herbivores, les vaches sont des mammifères herbivores, les chauves souris sont des mammifères omnivores qui volent, le tigre est un mammifère carnivore, le corbeau est un oiseau qui mange de la viande et de l'herbe.

On ne s'intéresse pour l'instant qu'à la relation type sous-type. C'est à dire exclusivement à l'héritage de type et non à l'héritage de code.

On suppose qu'il n'existe pas d'animaux qui sont à la fois des mammifères et des oiseaux. L'intersection entre les oiseaux et les mammifères est égale à l'ensemble vide. Donc, en aucun cas la chauve souris qui est un mammifère ne peut-être un sous-type de oiseaux même si la chauve souris vole.

Il en est de même pour les herbivores et les carnivores, même si les omnivores mangent comme les herbivores et les carnivores, cette particularité concerne le code et non les types. L'intersection entre les carnivores et les omnivores est elle aussi égale à l'ensemble vide.

A partir des spécifications précédentes, on obtient donc le diagramme ensembliste suivant:

Ce diagramme ensembliste se traduit donc sous la forme suivante en terme de hierarchie de type.

Maintenant nous devons nous intéresser à définir les interfaces et les classes de notre application.

Concernant le type `Animal`, à ce stade nous n'avons aucune information sur comment les animaux mangent et se reproduisent. Nous ne connaissons donc que l'interface fonctionnelle de `Animal`.

```
Public interface Animal {  
    public void manger();  
    public void reproduire();  
}
```

Nous savons que les Mammifères sont des Animaux qui ont une particularité supplémentaire celle d'allaiter. Il s'agit donc d'une extension du comportement de `Animal`. Nous savons également comment il se reproduit, c'est à dire que nous pourrions donner du code aux services `allaite` et `reproduire`. Par contre, nous ne pouvons pas donner du code pour le service `manger`. Donc, si nous faisons de `Mammifere` une classe, elle ne pourrait être instanciable car elle ne fournit pas une implémentation pour l'ensemble des services décrits par l'interface fonctionnelle publique. Nous allons donc en faire une interface.

```
Public interface Mammifere extends Animal  
{  
    public void manger();  
    public void reproduire(); //code possible  
    public void allaite(); // extension du comportement de Animal.  
}
```

Nous avons exactement les mêmes caractéristique pour le type `Oiseau`, avec une extension du comportement avec le service `vole`, et la possibilité de définir du code pour `reproduire`, mais nous n'avons toujours pas la possibilité de donner du code au service `manger`. Nous avons donc une nouvelle interface qui est la suivante.

```
Public interface Oiseau extends Animal  
{  
    public void manger();  
    public void reproduire(); // code possible  
    public void vole(); // extension du comportement de Animal.  
}
```

Pour les types `Herbivore`, `Carnivore` `Omnivore`, il n'y a pas d'extension du comportement. On peut donner du code pour le service `manger`, mais comme `Herbivore`, `Carnivore` et `Omnivore` sont des sous-types de `Animal`, ils doivent aussi posséder du code pour le service `reproduire`. Mais ceci, n'est pas possible donc pour les mêmes raisons que précédemment, à notre niveau de connaissance, ces trois types ne peuvent être représentés par des classes instanciables, ils seront donc associés à des interfaces. Nous ne montrerons que la déclaration de l'interface `Herbivore`, les autres seront identiques.

Public interface Herbivore extends Animal

```
{
    public void mange(); // code possible
    public void reproduire();
}
```

Maintenant, il nous reste à voir toutes les autres types, qui sont Cochon, Vache, Tigre, Poule, Chauve souris et Corbeau. Pour tous ces types, on peut donner du code pour tout leur service public. Mais nous allons voir que nous allons créer de la duplication de code. En effet, si l'on considère par exemple que tous les herbivores mangent de la même manière, le code pour faire manger la poule et la vache se trouveront déclarés dans ces deux classes. Si l'on considère que tous les mammifères allaitent de la même manière, il y aura de la duplication de code pour le service allaiter de tous les mammifères. Par contre, comme pour tous ces types on peut donner une implémentation de tous les services, on peut maintenant les concrétiser en faisant une classe. Par exemple,

public class Cochon implements Omivore, Mammifères // héritage multiple de type

```
{
    public void allaiter () { // code dupliquer pour tous les mammifères
        System.out.println(« Allaiter »);
    }
    public void reproduire() { // code dupliquer pour tous les mammifères
        System.out.println(« Reproduire »);
    }
    public void manger() { // code dupliquer pour tous les Omnivores
        System.out.println(« manger comme un herbivore »); // code de Herbivore
        System.out.println(« manger comme un carnivore »); // code de carnivore.
    }
}
```

Un autre exemple avec la chauve souris qui vole comme un oiseau mais qui n'est pas un oiseau.

public class ChauveSouris implements Omivore, Mammifères // héritage multiple de type mais aucun
// rapport avec le type oiseau

```
{
    public void vole () { // code dupliquer si la chauve souris vole comme les oiseaux
        System.out.println(« vole »);
    }
    public void allaiter () { // code dupliquer pour tous les mammifères
        System.out.println(« Allaiter »);
    }
    public void reproduire() { // code dupliquer pour tous les mammifères
        System.out.println(« Reproduire »);
    }
}
```

```

    }
    public void mange(){ // code dupliquer pour tous les Omnivores
        System.out.println(« mange comme un herbivore »); // code de Herbivore
        System.out.println(« mange comme un carnivore »); // code de carnivore.
    }
}

```

Nous verrons par la suite comment factoriser le code pour éviter la duplication au sein d'une application. Par cela, il faudra utiliser les notions de classe abstraite et d'héritage de code.

Héritage de Code et classe abstraite.

Nous avons vu dans le cours précédent la notion de classe et la notion d'interface ainsi que la relation d'héritage de type.

Relation entre une interface et des interfaces: Une interface peut-être une extension d'une ou plusieurs autres interfaces. Dans ce cas, une interface peut-être un sous-type de plusieurs autres types. Dans ce cas, il y a un héritage de type entre le sous-type et le sur-type.

Relation entre une classe et des interfaces: La relation qui existe entre une classe et un ensemble d'interfaces est la relation de concrétisation. Une classe peut réaliser plusieurs interfaces en fournissant l'implémentation des différents services publics déclarés par les interfaces. Dans ce cas, il y a encore un héritage de type entre le type déclaré par la classes et ses sur types qui sont ceux déclarés par les interfaces qu'elle concrétise.

Nous allons maintenant aborder une relation supplémentaire qui est:

Relation entre une classe et une autre classe: La relation qui existe entre une classe est une relation d'héritage simple. Elle se déclare en utilisant le mot clef extends.

Class A extends B {.....}

Cette relation implique :

3. **Héritage de type entre le type A et le sur-type B.** Cette propriété implique la validité de l'héritage de type. En aucun cas, on ne peut utiliser l'héritage entre classe pour des raisons de facilité d'implémentation. Il faut que dans le domaine du problème, les objets instances de A valident la relation sémantique « Les objets instance de A sont des sortes de B »
4. **Héritage de code, c'est à dire la possibilité pour la classe A de pouvoir utiliser tout au partie du code de la classe B pour implémenter ses propres services.**

Héritage de code.

Considérons par exemple, le code suivant inspiré des TD:

```

public class FormeFacto {
    private Point2D centre;

    public FormeFacto(Point2D p)

```

```

    {
        centre = p.creerCopie();
    }

    public void translate(Point2D p, Point2D q)
    {
        centre.translate(p,q);
    }

    public Point2D donneCentre()
    {
        return centre.creeCopie();
    }
}

```

Pour l'instant la classe `FormeFacto` est une classe instanciable car elle donne du code à la totalité des services qui sont présents dans son interface fonctionnelle publique.

Maintenant, nous allons nous intéresser à la classe `Cercle`. La première question à se poser est la suivante: Est ce qu'un cercle est une sorte de `FormeFacto` particulière qui possède un centre ? Si la réponse à cette question est oui. Alors on peut considérer le type `Cercle` comme un sous-type de `FormeFacto`, ce qui valide l'héritage de type et qui permet donc d'utiliser l'héritage entre classe.

```

Public class Cercle extends FormeFacto
{
    private double rayon;

    public Cercle(Point2D p, double rayon)
    {
        ..... // le code à ajouter.
    }
}

```

Comme un `Cercle` est une sorte de `FormeFacto`, il doit posséder tous les attributs d'une `FormeFacto` c'est à dire un centre. Comme un `Cercle` est un sous-type de `FormeFacto`, il doit au moins posséder les services publics de `FormeFacto`. La première propriété concerne l'héritage de type alors que la deuxième propriété concerne l'héritage de code. Nous allons maintenant aborder les propriétés de l'héritage de code.

Propriété de l'héritage de Code.

3. Les propriétés de l'héritage de type (principe de substitution et inclusion de IFP) sont impliquées par l'héritage de code.
4. A l'intérieur de la structure d'un sous-type, il y a la structure du sur-type. Cette structure n'est pas toujours accessible, mais elle est toujours présente. Pour construire un sous type il est donc nécessaire de construire un sur-type, lorsque le sur-type déclare du code. C'est à dire que le sur-type est une classe et non une interface. Cela veut dire que le constructeur du sous-type appelle le constructeur du sur-type. Par défaut, le constructeur du sur type est celui sans paramètres si ce dernier est accessible. Par exemple, si on définit de la manière suivante le constructeur de `Cercle`.

```

public Cercle(Point2D p, double rayon)
{
// appel implicite au constructeur FormeFacto()
this.rayon = rayon;
}

```

Ce code va générer une erreur car il essaye de construire la partie FormeFacto en utilisant le constructeur sans paramètres hors ce dernier n'est pas accessible. En effet, le fait d'avoir définie le constructeur FormeFacto(Point2D p) masque le constructeur sans paramètre qui est défini par défaut par Java. Pour pouvoir appeler le constructeur du sur-type depuis le constructeur du sous-type, il faut utiliser la syntaxe super(.....paramètre). Il faut que cette appel soit la première ligne du constructeur du sur-type. Comme il n'y a que de l'héritage simple de code en Java, il ne peut y avoir d'ambiguïté avec le mot clef super. Le constructeur de Cercle est alors le suivant:

```

public Cercle(Point2D p, double rayon)
{
super(p) ; // appel explicite au constructeur FormeFacto(Point2D)
this.rayon = rayon;
}

```

On peut même rajouter un autre constructeur en utilisant l'appel à un autre constructeur grâce au mot clef this.

```

public Cercle()
{
this(new Point2DCartesien(0,0), 1); // appel a Cercle(Point2D,double)
}

```

Donc maintenant lorsqu'on appelle new Cercle(p,3) on déclenchera un appel à FormeFacto(p).

2. La troisième propriété concerne le partage de code entre le sur-type et le sous type. Le code du sur-type peut-être totalement ou partiellement utilisé par le sous type. Il faut bien que ce code soit accessible (déclaré protected), nous reviendrons par la suite sur la déclaration permettant l'accessibilité. Nous avons vu précédemment que le sous-type contient la structure du sur-type, donc le code qui est défini sur le sur-type peut être utiliser avec le sur-type.

- Utilisation d'un service déclaré dans le sur-type. Si on reprend la déclaration de Cercle.

```

Public class Cercle extends FormeFacto
{
private double rayon;

public Cercle(Point2D p, double rayon)
{
..... // le code à ajouter.
}
}

```

Le service translation n'est pas défini, mais la classe cercle est instanciable donc le code pour faire fonctionner le service translation existe et il doit être accessible lorsqu'on le selectionne en utilisant un cercle. Prenons le code suivant:

```

Cercle c = new Cercle(p,2);
FormeFacto f = c; // principe de substitution
// c et f identifie exactement le même objet.
f.translate(.....); // instruction correcte car f est une classe instanciable et donc

```

```
// tous les services publics sont implémentés.
```

```
c.translate(.....); // doit lui aussi fonctionner car c a été instancié.
```

```
// le code qui sera utilisé sera celui de FormeFacto.
```

Dans ce cas, l'héritage de code permet d'utiliser l'intégralité d'un service déclaré dans surtype sans avoir besoin de le déclarer ou de le définir dans le sous-type.

le

- Utilisation et appel d'un service du sur-type à l'intérieur du code du sur-type.

Rajoutons par exemple le service toString() à la classe FormeFacto

```
public class FormeFacto {
    private Point2D centre;
    public FormeFacto(Point2D p)
    {.....}
    public void translate(Point2D p, Point2D q)
    {.....}
    public Point2D donneCentre()
    {.....}
    public String toString() {
    return « Je suis une formeFacto » + centre.toString();
    }
}
```

On veut maintenant définir un service toString() au niveau de la classe Cercle, mais on veut que les informations de FormeFacto apparaissent. Il faut donc à l'intérieur du service toString() de Cercle appelée le service toString() de FormeFacto(). Pour appeler le service de nom XX du sur type il faut utiliser la syntaxe suivante :

```
super.XX(...)
```

Nous pouvons maintenant écrire le code du service toString() de la classe Cercle.

```
Public class Cercle extends FormeFacto
```

```
{
    private double rayon;

    public Cercle(Point2D p, double rayon)
    {..... }
    public String toString()
    {
        String tmp = super.toString(); // appel a toString() de FormeFacto

        return « Je suis un cercle de rayon » + rayon + tmp;
    }
}
```

Le mot clef protected.

Nous avons vu dans les premiers cours que pour contrôler l'accès à l'information (service ou variables). On pouvait utiliser les mots clefs private et public. Les informations private ne sont accessibles qu'à l'intérieur de la classe qui les définit, les informations public sont accessible par tout le monde. La relation d'héritage de code crée un lien particulier entre une sous-classe et sur classe. Il peut-être nécessaire d'accéder depuis la sous classe à une partie de l'implémentation de la surclasse. Sans pour autant que l'information soit visible pour tout le monde. En effet comme une partie du code peut-être partagée par la sous classe, il est parfois nécessaire qu'elle puisse accéder à la structure du sur-type qui la constitue en partie. Pour permettre l'accès à l'information de la sur-classe depuis la sous-classe, il faut utiliser le mot clef protected lorsqu'on déclare l'information.

Par exemple, si la classe Cercle devait accéder à la variable centre de la classe FormeFacto pour la modifier. La variable centre est bien à l'intérieur de la structure d'un objet centre mais elle n'est pas accessible car son accès est privée. Par contre, elle est accessible en lecture en utilisant le service donneCentre(), mais on peut pas la modifier car c'est seulement une copie qui est retournée.

Comme il est bien sûr hors de question de la rendre publique, on doit la déclarer protected. Et elle devient alors publique pour les sous-classes et private pour toutes les autres classes. La déclaration devient alors.

```
public class FormeFacto {
    protected Point2D centre;
    .....
}
```

Classe Abstraite.

Nous avons vu qu'une classe est non instanciable si elle ne fournit pas l'intégralité du code pour implémenter l'ensemble de ces services publics. Qu'une interface n'est pas instanciable car elle n'implémente aucun service public. Une classe peut tout de même présenter une implémentation de certains services publics qui peut être factorisée pour l'ensemble de ses sous-classes, mais elle n'est pas capable de fournir l'intégralité de l'implémentation des services. Dans l'exemple sur les animaux, nous avons vu que nous pouvions donner par exemple du code pour le service mange des herbivores, mais nous ne pouvons pas en donner pour le service reproduire, donc cette restriction nous a obligé à transformer Herbivore en interface, et à engendrer de la duplication de code pour le service mange pour tous les animaux concrets qui sont herbivores (vache, poule,...)

Il existe une solution intermédiaire entre la classe instanciable qui donne l'intégralité du code et l'interface qui ne donne aucun cas. Cette solution consiste à définir une classe abstraite. Une classe abstraite est considérée comme une classe du point de vue de l'héritage de code. Cela veut dire qu'une classe ne peut hériter d'une code que d'une classe qu'elle soit abstraite ou non.

Une classe abstraite est non instanciable directement. C'est à dire que l'on ne peut écrire new..... Mais par contre ces sous-types pourront créer une instance de cette classe. Une classe abstraite fournit l'implémentation de certains services de l'interface fonctionnelle publique mais pas de tous. Les services implémentés par une classe abstraite servent pour la factorisation de code. Syntaxiquement, une classe abstraite XX avec un service abstrait serviceAbstrait se déclare de la manière suivante:

```
public abstract class XX
{
    protected int tmp1;

    protected XX(int v) // le constructeur est protected car seule les sous classes de XX peuvent
    {
        // l'instancier.
        tmp1 = v;
    }

    public int inc() // un service factorisé
    {
```

```

        return ++tmp1;
    }

    abstract public void serviceAbstrait(int x); // déclaration du service abstrait.
}

```

Sur cet exemple, on ne peut pas écrire directement `new XX(2)` même si le constructeur avait été déclaré public. La classe n'est pas instanciable directement.

Si on veut utiliser cette classe abstraite on peut définir maintenant une de ces sous classes.

```

Public class YY extends XX
{
    public YY(int v)
    {
        super(v); // on instancie implicitement un XX en créant un YY
                // on appelle le constructeur
    }

    public void serviceAbstrait(int x) // implementation du service abstrait de XX
    {
        tmp1 += 2; // on peut accéder à la variable tmp1 qui est protected
    }
}

```

La classe YY donne une implémentation aux services de son interface fonctionnelle publique. Cette interface fonctionnelle comprend le service `inc` défini par XX et le service `serviceAbstrait` qui est déclaré dans XX et qui est défini dans YY. La classe YY est donc instanciable. On peut écrire le code suivant :

```

YY y = new YY(2); // on crée tout de même un XX;
XX x = y; // principe de substitution

x.inc();
x.serviceAbstrait(); on utilise celui défini par YY.

```

Un autre exemple, peut-être de modifier la classe `FormeFacto` en lui rajoutant le service `surface`. Ce service est nécessairement abstrait car nous ne connaissons par réellement le type de `Forme` que nous allons manipuler. Mais par contre, tous les utilisateurs de `FormeFacto` peuvent appeler le service `surface`. Le code devient alors :

```

abstract public class FormeFacto { // FormeFacto est maintenant abstraite donc non instanciable.
    protected Point2D centre;

    protected FormeFacto(Point2D p)
    {
        centre = p.creerCopie();
    }

    public void translate(Point2D p, Point2D q) //code factorisé
    {
        centre.translate(p,q);
    }
}

```

```

    }

    public Point2D donneCentre() // code factorisé
    {
        return centre.creeCopie();
    }
    abstract double surface(); // le service surface est abstrait.
}

```

La classe Cercle maintenant pour continuer à être instanciable doit-implémenter ce service.

```

Public class Cercle extends FormeFacto
{
    private double rayon;

    public Cercle(Point2D p, double rayon {..... }
    public String toString(){
        String tmp = super.toString(); // appel à toString() de FormeFacto
        return « Je suis un cercle de rayon » + rayon + tmp;
    }
    public double surface() { // implementation du service abstrait.
        return Math.PI * rayon*rayon;
    }
}

```

Si nous revenons à l'exemple des Animaux vus précédemment, nous avons constaté qu'il y avait de la duplication de code au niveau des classes concrètes Vache, Poule,..... Par exemple, les services allaite et reproduire sont les mêmes pour tous les mammifères, les services vole et reproduire sont les mêmes pour tout les oiseaux, le service mange et le même pour tous les herbivores....etc. En utilisant les classes abstraites nous pouvons réduire la duplication de code soit pour tous les mammifères et pour tous les oiseaux, soit pour tous les herbivores, les canivores et les omnivores. Mais on ne pourra pas supprimer toute la duplication de code car il n'y a que de l'héritage simple de code en Java. Il faudra utilisée la délégation.

Nous allons simplement illustré, l'utilisation des classes abstraites et de l'héritage pour factorisez le code des mammifères et nous allons utiliser la délégation pour factoriser le code des herbivores.

```

Interface Mamifere extends Animal
{.....}

```

```

Abstract public class MammifèresImpl implements Mamifere // class non instanciable
{
    protected MammifereImpl(...) // le constructeur de mammifere
    {.....}
    public void allaite() {System.out.println(« J'allaite mes petits »);} // code facto
    public void reproduire() {System.out.println(« J'accouche»);} // code facto
    abstract public void mange(); // service dont on ne connait pas le code
}

```

```

Interface Herbivore extends Animal
{.....}

```

```

class HerbivoreImpl { // on ne peut la considérer comme un sous-type de Herbivore car cette classe
    // ne pourrait être instanciable, or elle doit l'être.
    Public void mange(){System.out.println(« Je mange de l'herbe »);}
}

```

```

public class Vache extends MamifereImpl implements Herbivore
{
    protected HerbivoreImpl delegHerbivore;
    public Vache() {
        super(.....); // appel au constructeur de MamifereImpl
        delegHerbivore = new HerbivoreImpl();
    }

    public mange(){
        delegHerbivore.mange();
    }
}

```

La classe Vache est bien une classe instanciable car elle donne du code à l'ensemble des services définis par son interface fonctionnelle publique. Le code de reproduire et de allaiter vient de la propriété de l'héritage de code de la classe MamifereImpl. Le code de manger déclaré abstrait dans MamifereImpl est maintenant écrit. Il se fait par délégation en utilisant le code de HerbivoreImpl.

Surcharge et Redéfinition.

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 PointCartesien(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 surchargées diffère par le nombre de paramètres ou par le type des paramètres à une position donnée. La type de retour du service surchargée 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.

Soit l'appel $o.f(p_1, \dots, p_n)$ ou $T.f(p_1, \dots, p_n)$, on doit maintenant sélectionner la signature du service f qui correspond au mieux à cet appel. On notera T_1 le **type déclaré** de $p_1, \dots, T_N$ le type déclaré de p_n .

5. 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 surtypes 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 .
6. 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 $f(S_1, \dots, S_n)$. On doit savoir si elle est compatible avec l'appel $f(p_1, \dots, p_n)$. Pour un S_i à une place i , on regarde si il est compatible avec T_i le type déclaré du paramètres p_i de l'appel, **ON CONSIDERE UNIQUEMENT LE TYPE DECLARE T_i du paramètre p_i . JAMAIS LE TYPE REEL du paramètre p_i , ne sera pris en compte.** Un paramètre T_i est compatible avec un paramètre S_i si :
 1. si T_i et S_i sont deux types primitifs, alors on peut convertir un T_i en un S_i , soit par promotion soit par coercion
 2. si T_i et S_i sont deux types non primitifs, alors il faut que T_i soit égal à S_i ou bien que T_i soit un sous type de S_i , il faut que T_i hérite de S_i .
 3. Sinon les deux types sont incompatibles.
7. Après l'étape 2, il ne reste plus qu'un ensemble de
8. La compatibilité se fait paramètre à paramètre. On regarde si le type déclaré d'un paramètre de l'appel est compatible avec le type du paramètre de la signature.
9. La troisième étape consiste à supprimer les signatures des services f qui ne sont pas compatibles

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

Polymorphisme.

Le polymorphisme est le résultant de :

10. **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.
11. **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 redéfini 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 instantiation 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 faite 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 typ e 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 il 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()** permet 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 définigt 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 reprend redefini 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. Un package est un ensemble de classes 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,

5. Les informations publiques: ceux sont des classes ou des interfaces qui sont visibles de l'extérieur du package.
6. Les informations locales aux paquets: 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 pouvoir 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 NOMPCKAGE seront accessible depuis ce fichier, les entités locales à NOMPCKAGE 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 privilégié 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:

	Même classe	Même package	Sous classe
<code>public</code>	oui	oui	oui
<code>private</code>	oui	non	non
<code>protected</code>	oui	oui	oui
vide	oui	oui	non

Exception en Java.

Le principe général des exceptions

Le mécanisme d'exception est un mécanisme de déroutement du programme, c'est à dire qu'elle brise la séquentialité du programme (comme le fait `return`, `break`, `continue`). Le mécanisme qui brise la séquentialité est la **levée d'exception**. Lorsqu'une exception est levée, le programme est dérouté vers les **premiers gestionnaires d'exceptions**.

Les gestionnaires d'exception sont associés à **des blocs de code qui ont été sensibilisés** à certains types d'exception.

L'exception se propage depuis l'endroit où elle est levée, elle remonte d'appellant en appellant en s'arrêtant à chaque gestionnaire d'exceptions qu'elle rencontre dans la remontée. Chaque gestionnaire d'exception est passée en revue, en commençant par le premier qui est déclaré. Si celui-ci est de type compatible avec le type de l'exception, alors le code associé à la récupération de l'exception est exécuté et le programme reprend normalement son exécution après le dernier gestionnaire d'exception. Si aucun gestionnaire d'exception ne convient, le même processus est réitéré en remontant, jusqu'à qu'un gestionnaire d'exception la capture ou que la fonction main est atteinte.

La syntaxe des exceptions.

La gestion des exceptions en Java se fait par l'utilisation de trois mots clefs **try, catch, throw**. Le mot clef **try** sert à définir un bloc dans lequel les exceptions sont susceptibles d'être capturées. On parle aussi de sensibilisation d'un bloc à un ensemble d'exceptions. La capture des exceptions est confiée à des gestionnaires d'exception. Après la définition de ce bloc se trouve la liste des gestionnaires d'exception. Un gestionnaire d'exception est défini en utilisant le mot réservé **catch**. La liste des gestionnaires d'exception ressemble à un ensemble de fonctions surchargées. Une exception est levée en utilisant le mot clef **throw**. L'identification de l'exception, se fait en utilisant le type d'un objet.

La hiérarchie des exceptions:

En java, toutes les exceptions hérite de Throwable. Il y a deux types d'exceptions :

- celles qui héritent soit de Error soit de RuntimeException, ces exceptions sont des exceptions d'implémentation et elles n'ont pas besoin d'être déclarée dans la signature de la fonction.
- Les autres qui sont alors considérées comme des exceptions d'utilisation et qui doivent être impérativement déclarées dans la signature de la fonction si elles ne sont pas capturées par la fonction.

Les services de la classe Throwable sont :

Throwable **fillInStackTrace()** Fills in the execution stack trace.

Throwable **getCause()** Returns the cause of this throwable or null if the cause is nonexistent or unknown.

String **getLocalizedMessage()** Creates a localized description of this throwable.

String **getMessage()** Returns the detail message string of this throwable.

StackTraceElement[] **getStackTrace()** Provides programmatic access to the stack trace information printed by **printStackTrace()**.

Throwable **initCause(Throwable cause)** Initializes the cause of this throwable to the specified value.

void **printStackTrace()** Prints this throwable and its backtrace to the standard error stream.

void **printStackTrace(PrintStream s)** Prints this throwable and its backtrace to the specified print stream.

void **printStackTrace(PrintWriter s)** Prints this throwable and its backtrace to the specified print writer.

void **setStackTrace(StackTraceElement[] stackTrace)** Sets the stack trace elements that will

be returned by
by `printStackTrace()`

`getStackTrace()` and printed

and related methods.

`String toString()` Returns a short description of this throwable.

Exemple d'exception.

Ces méthodes permettent de capturer le contexte d'une exception. Mais on peut aussi créer une nouvelle catégorie d'exception en créant sa propre classe et en la rattachant à la hiérarchie des exceptions de Java.

```
class MonException extends Exception{
    private String mesDonnées;
    public MonException(String donnees){ MesDonnées = new String(donnees);}
    public String getInfo(){ return mesDonnées;}
}
class MonAutreException extends MonException{.....}
class MonError extends Error{.....}
```

Sur cet exemple, on vient de définir trois nouveaux types d'exception qui permettront de définir les types de saut de notre programme.

```
public class Ex1 {
    static public void f(int i) throws MonException, MonAutreException {
        try {
            if(i==0) throw new MonError(.....); // Lancement d'une exception
            if(i % 2 == 0) throw new MonAutreException(.....);
            else throw new MonException(.....);
            System.out.println(« La FIN du Bloc »);
        }
        catch(MonError e) {System.out.println(« Exception Traitée »);}
        System.out.println(« La fonction est finie »);
    }
}
```

Sur cette exemple, la ligne

public void f(int i) throws MonException, MonAutreException

complète la signature d'une fonction en indiquant quelles sont les exceptions que le service f peut lever. En java toutes les exceptions d'utilisation qui peuvent être levée d'une fonction doivent être déclarées dans la signature de cette fonction. Les exceptions ne sont pas pris en compte dans la distinction de la surcharge. Par contre, les exceptions d'implémentation n'ont pas être obligatoirement déclarées dans la signature de la fonction.

Le bloc **try** sensibilise le bloc d'instruction de la fonction à une seule nature d'exception qui est

représentée par le type **MonError**. La classe **catch** représente le gestionnaire d'exception qui est traitera les exceptions de type **MonError**. Si une exception de type compatible avec le type **MonError** est levée dans ce bloc, alors le code associé à ce **catch** sera exécuté et le programme continuera après la dernière classe **catch** du code de ce service.

Suivant la valeur de **i** le code sera :

- **Si i est égal à zéro**, l'exception est levée, le programme sort du bloc **try** et commence à parcourir les clauses **catch** les unes après les autres en commençant par la première. Ici, il n'y a qu'une clause **catch**. Le type réel de l'exception est **MonError** ce qui correspond au type déclaré de la clause **catch**. Le code de la clause **catch** est exécutée, le message « **Exception Traitée** » est affiché, puis le programme reprend séquentiellement et le message « **La fonction est finie** » est ensuite affiché.
- **Si i est différent de zéro**, une exception de type **MonException** ou de type **MonAutreException** sera levée dans le bloc d'instruction. Puis la clause **catch** sera consultée, comme les types des exceptions ne sont pas compatibles avec **MonError** cette clause **catch** ne sera pas exécutée. Le programme continuera en se dérivant sur l'appelant et le message « **La fonction est finie** » ne sera pas affiché.

Exception et signature de service.

Si maintenant on regarde un client possible de **Ex1**, par exemple

```
public class Client {  
    public g(int i) throws MonException  
    { Ex1.f(i); }  
}
```

Dans ce cas, la fonction **g** appelle la fonction **f** qui a été définie précédemment. La fonction **f** lève deux natures d'exceptions, comme la fonction **g** ne capture aucune exception, elle doit les déclarer dans sa signature.

En Java, les exceptions qui peuvent être levées par une fonction doivent appartenir à la signature de la fonction. Mais, il n'est pas obligatoire de les identifier précisément, il suffit que toutes les exceptions levées par la fonction soient compatibles avec les exceptions déclarées dans la signature. La déclaration des exceptions dans la signature de la fonction permet au compilateur Java, de vérifier que toute les fonctions déclarent bien les exceptions qu'elles lèvent. Il suffit de regarder le code de la fonction, de collecter les exceptions qui peuvent être levées par le code (ELLES sont nécessairement dans la signature), d'enlever pour chaque bloc **try celles qui sont capturées, puis de vérifier que les exceptions restantes sont bien déclarées dans la signature de la fonction.**

RAPPEL, seules les exceptions d'utilisation sont vérifiées par le compilateur.

Sur l'exemple précédent, la fonction **g** lève deux types différents d'exception qui sont **MonAutreException** et **MonException**, comme le type **MonAutreException** est compatible avec le type **MonException**. Il est suffisant de déclarer seulement :

```
public g(int i) throws MonException
```

Propagation des Exceptions.

```
public class Client1
{
    public void h(int i) throws MonException
    {
        Tmp e = new Tmp();
        try {
            double j = 1/i;
            System.out.println(« La valeur de i est non nulle »);
            e.fun(i);
            System.out.println(« Ligne jamais écrite »);
        }
        catch(MonAutreException e){ System.out.println(« J'ai traité mon Exception »);}
        catch (Error e) {System.out.println(« J'ai divisé par zero »)}
        System.out.println(« Je continue »);
    }
}

public class Tmp
{
    public void fun(int i) throws MonException, MonAutreException
    {
        Ex.f(i);
    }
}
```

Sur cet exemple, le service f de la classe Client1 a sensibilisé une partie du code à des exceptions. Elle capture les sous type de Error et elle capture les exceptions de type MonAutreException.

Lorsque la valeur de i est nulle, Java lance une Exception qui est division par zéro, cette exception est un sous-type de Error. Le programme sera donc dérouté vers les clauses catch, la première clause catch traitée correspond au type MonAutreException. Le type Error n'est pas compatible avec le type MonAutreException, cette clause catch ne convient donc pas. On passe alors à la suivante. La seconde clause est bien compatible avec l'exception.