SOMMAIRE

INTRODUCTION

Cette deuxième partie traite des concepts de Programmation Orientée Objet en langage JAVA. Ce support de formation est constitué d'une liste d'exercices permettant de construire pas à pas une classe d'objet : la classe. Bien que les corrigés types de chaque étape soient présentés dans des répertoires séparés, il s'agit en fait du même cas qui va évoluer, étape par étape, pour aboutir à un fonctionnement correct de la classe dans l'univers JAVA.

Chaque exercice est structuré de la façon suivante :

- Description des objectifs visés.
- Explications des techniques à utiliser (Ce qu'il faut savoir).
- Enoncé du problème à résoudre (Travail à réaliser).
- Renvois bibliographiques éventuels dans les ouvrages traitant de ces techniques (Lectures).

Tous ces exercices sont corrigés et commentés dans le document intitulé :

- Proposition de corrigé JAVA_BPC.
  Apprentissage d'un langage de programmation Orientée Objet
  Le langage C++ (objet).

- Dans ce support, les programmes produits fonctionneront en mode texte dans une fenêtre MS-DOS. La création d'*applets* graphiques sera l'objet du troisième support de la collection.

Pour pouvoir utiliser ce support, les bases de la programmation en langage JAVA doivent être acquises.

Pour les personnes qui ne connaissent pas JAVA, il est possible d'acquérir ces bases à l'aide du document intitulé :

- Support de Formation JAVA_ASF
  Apprentissage d'un langage de programmation Orientée Objet
  JAVA (bases).

Les ouvrages auxquels il sera fait référence dans le présent support sont les suivants :

- **JAVA (Le Macmillan)**
  Alexander Newman
  Editions Simon & Schuster Macmillan

- **Formation à JAVA**
  Stephen R. Davis
  Editions Microsoft Press

- **JAVA (Megapoche)**
  Patrick Longuet
  Editions Sybex

T.P. N°1 - RÉFÉRENTIEL OU CLASSE D'OBJET ENCAPSULATION

1.1 OBJECTIFS

- Notion de classe d'objet
- Définition d'une nouvelle classe d'objet en JAVA.
- Encapsulation des propriétés et des méthodes de cet objet.
- Instanciation de l'objet.

1.2 CE QU'IL FAUT SAVOIR

1.2.1 Notion de classe

Créer un nouveau type de données, c'est modéliser de la manière la plus juste un objet, à partir des possibilités offertes par un langage de programmation.

Il faudra donc énumérer toutes les propriétés de cet objet et toutes les fonctions qui vont permettre de définir son comportement. Ces dernières peuvent être classées de la façon suivante :

- **Les fonctions d'entrée/sortie**. Comme les données de base du langage JAVA, ces fonctions devront permettre de lire et d'écrire les nouvelles données sur les périphériques (clavier, écran, fichier, etc.).
- **Les opérateurs de calcul**. S'il est possible de calculer sur ces données, il faudra créer les fonctions de calcul.
- **Les opérateurs relationnels**. Il faut au moins pouvoir tester si deux données sont égales. S'il existe une relation d'ordre pour le nouveau type de donnée, il faut pouvoir aussi tester si une donnée est *inférieure* à une autre.
- **Les fonctions de conversion**. Si des conversions vers les autres types de données sont nécessaires, il faudra implémenter également les fonctions correspondantes.
- **Intégrité de l'objet**. La manière dont est modélisé le nouveau type de données n'est probablement pas suffisant pour représenter l'objet de façon exacte. Par exemple, si l'on représente une fraction par un couple de deux entiers, il faut également vérifier que le dénominateur d'une fraction n'est pas nul et que la représentation d'une fraction est unique (simplification).

La déclaration d'un nouveau type de données et les fonctions qui permettent de gérer les objets associés constituent le ***référentiel*** de ce type de données. En Programmation Orientée Objet, on parle de ***classe*** de l'objet. Les propriétés de l'objet seront implémentées sous la former de ***données membres*** de la classe. Les différentes fonctions ou méthodes seront implémentées sous la forme de ***fonctions membres*** de la classe. De façon courante, dans le *patois* de la programmation orientée objet, ***données membres*** et ***fonctions membres*** de la classe sont considérées respectivement comme synonymes de ***propriétés*** et ***méthodes*** de l'objet.

1.2.2 Encapsulation

L'encapsulation est un concept important de la Programmation Orientée Objet.

L'encapsulation permet de rassembler les ***propriétés*** composant un objet et les ***méthodes*** pour les manipuler dans une seule entité appelée ***classe*** de l'objet.

Une classe, en JAVA se déclare par le mot clé **`class`** suivi d'un identificateur de classe choisi par le programmeur de la façon suivante :

```
class NomDeLaClasse
{
   // Déclaration des propriétés et
   // des méthodes de l'objet
}
```

Les identifiants de classe obéissent aux même règles que les identifiants de variables et de fonctions (voir page 6 du précédent support). Par convention, ils commencent par une Majuscule (JAVA considère les majuscules comme des caractères différents des minuscules).

Les propriétés et les méthodes de l'objet seront déclarées et implémentées dans le bloc **`{ }`** contrôlé par le mot clé **`class`**. Cela constitue l'implémentation du concept d'encapsulation en JAVA.

1.2.3 Déclaration des propriétés d'un objet

Les propriétés d'un objet sont déclarées, comme des variables, à l'intérieur du bloc **`{ }`** contrôlé par le mot clé **`class`**.

```
class NomDeLaClasse
{
   TypeDeLaProprieté  m_nomDeLaPropriete;


   // Déclaration des méthodes de l'objet
}
```

- Les propriétés peuvent être déclarées à tout moment à l'intérieur du corps de la classe.

- Chaque déclaration de propriété est construite sur le modèle suivant :
  `TypeDeLaProprieté  m_nomDeLaPropriete;`

- Une propriété peut être initialisée lors de sa déclaration :
  `TypeDeLaProprieté  m_nomDeLaPropriete = valeurInitiale;`

- Les identifiants de propriété obéissent aux même règles que les identifiants de variables et de fonctions (voir page 6 du précédent support). Par convention, ils commencent par une minuscule (JAVA considère les majuscules comme des caractères différents des minuscules)et sont préfixés **`m_`**. Comme toutes les instructions du langage JAVA, chaque déclaration de propriété DOIT absolument être terminée par un point-virgule **`;`** Le point-virgule ne constitue pas un séparateur, mais plutôt un *terminateur* d'instructions.

1.2.4 Implémentation des méthodes

Les méthodes d'une classe sont implémentées , à l'intérieur du bloc **`{ }`** contrôlé par le mot clé **`class`**.

Quand on considère une méthode par rapport à l'objet à laquelle elle est appliquée, il faut voir l'objet comme étant sollicité de l'extérieur par un ***message***. Ce ***message*** peut comporter des paramètres. L'objet doit alors réagir à ce ***message*** en exécutant cette fonction ou méthode.

```
class NomDeLaClasse
{
   // Déclaration des propriétés de l'objet


   TypeResultat nomDeMethode(Type1 par1, Type2 par2)
   {
      // Instructions de la méthode
      return expression;
   }
}
```

- Les identifiants de méthodes obéissent aux même règles que les identifiants de variables et de fonctions (voir page 6 du précédent support). Par convention, ils commencent par une minuscule (JAVA considère les majuscules comme des caractères différents des minuscules).

- Comme toutes les instructions du langage JAVA, les instructions définissant le fonctionnement de la méthode DOIVENT absolument être terminées par un point-virgule **`;`** Le point-virgule ne constitue pas un séparateur, mais plutôt un *terminateur* d'instructions.

- Si aucun paramètre n'est désigné explicitement entre les parenthèses, le compilateur considère la méthode comme étant sans paramètre. Dans ce cas, les parenthèses son néanmoins obligatoires.

- Sauf si le résultat de la méthode est de type **`void`**, l'une des instructions doit être **`return`** suivi d'une expression dont le résultat est de même type. Ce résultat constitue le retour de la méthode et peut être exploité par le programme envoyant le message.

> *Les fonctions développées, dans le cadre des T.P. du support précédent, peuvent être considérées comme des méthodes de la classe application. L'une de ces méthodes est la fonction* ***`main`****. Cette fonction constituant le point d'entrée de l'application peut être considérée comme la méthode exécutée en réponse d'un message envoyé par le système (la machine virtuelle JAVA) qui signifierait "démarre".*

1.2.5 Instanciation

Pour qu'un objet ait une existence, il faut qu'il soit ***instancié***. Une même classe peut être instanciée plusieurs fois, chaque instance ayant ses propriétés à une valeur particulière. Cette instanciation peut être effectuée dans l'une quelconque des méthodes de n'importe quelle classe, y compris dans les méthodes de la classe elle-même.

En JAVA, il n'existe qu'une seule manière de créer une ***instance*** d'une classe. Cette création d'instance se fait en deux temps :

- Déclaration d'une variable du type de la classe de l'objet,
- Instanciation de cette variable par l'instruction **new**.

Par exemple, l'instanciation de la classe **String** dans la fonction **main** d'une classe application se passerait de la façon suivante :

```
public static void main(String[] args)
{
   String s;
   s = new String("AFPA");
}
```

La déclaration d'une variable **s** de classe **Salarie** n'est pas suffisante. En effet, **s** ne *contient* pas une donnée de type **Salarie**. **s** est une variable qui contient une référence sur un objet. Par défaut, la valeur de cette référence est **null**, mot clé JAVA signifiant que la variable n'est pas référencée. La référence d'un objet doit être affectés à cette variable. Cette référence est calculée par l'instruction **new** au moment de l'instanciation.

Ces deux étapes peuvent être réduites à une seule instruction :

```
String s = new String("AFPA");
```

1.2.6 Accès aux propriétés et aux méthodes

Bien qu'un accès direct aux propriétés d'un objet ne corresponde pas exactement au concept d'encapsulation de la programmation orientée objet, il est possible de le faire en JAVA. C'est le fonctionnement effectué par défaut. On verra, au chapitre suivant, comment protéger les données membres en en interdisant l'accès.

L'accès aux propriétés et aux méthodes d'une classe se fait par l'opérateur **.**.

Opérateur .

```
Fraction fr1 = new Fraction();
fr1.m_nNumerateur = 3;
fr1.m_nDenominateur = 4;
fr1.afficher();
```

1.2.7 Conventions d'écriture

Quand on débute la Programmation Orientée Objet, l'un des aspects le plus rebutant est l'impression d'éparpillement du code des applications. En respectant quelques conventions dans l'écriture du code et en organisant celui-ci de façon rationnelle, il est possible de remédier facilement à cet inconvénient.

Ces conventions ne sont pas normalisées et peuvent différer en fonctions des ouvrages consultés.

Les classes

Tous les identificateurs de classe commencent par une Majuscule. Par exemple : **Fraction**.

Les propriétésConvention d'écriture:propriétés

Tous les identificateurs de propriétés sont préfixés **`m_`**. Par exemple : **`m_nDenominateur`**, **`m_nNumerateur`**.

Les noms de fichiersConvention d'écriture:

On codera chaque classe d'objet dans des fichiers différents dont l'extension sera **`.java`**. Le nom de fichier sera identique au nom de la classe. Exemple pour la classe **`Fraction`** : **`Fraction.java`**.

1.3 TRAVAIL À RÉALISER

- Créer la classe **`Salarie`**. Cette classe aura 5 propriétés :
  - matricule **`m_nMatricule int`**
  - catégorie **`m_nCategorie int`**
  - service **`m_nService int`**
  - nom **`m_strNom String`**
  - salaire **`m_dSalaire double`**

- Créer une méthode **`calculSalaire`** pour afficher la mention "Le salaire de " suivie du nom du salarié, suivi de " est de ", suivi de la valeur du salaire.

- Implémenter une classe application, avec une méthode **`main`** dans laquelle, la classe **`Salarie`** sera instanciée, pour en tester les composantes

1.4 POUR EN SAVOIR PLUS

Pour d'autres informations sur l'encapsulation en JAVA, lire :

- **JAVA (Le Macmillan)**
  Ch. 8 : Les classes

T.P. N° 2 - ENCAPSULATION PROTECTION ET ACCÈS AUX DONNÉES MEMBRES

2.1 OBJECTIFS

- Protection des propriétés (données membres).
- Fonctions ***Get*** et ***Set***, d'accès aux propriétés.

2.2 CE QU'IL FAUT SAVOIR

2.2.1 Protection des propriétés

En Programmation Orientée Objet, on évite d'accéder directement aux propriétés par l'opérateur **.** . En effet, cette possibilité ne correspond pas au concept d'encapsulation. Certains langages l'interdisent carrément. En JAVA, c'est le programmeur qui choisit si une donnée membre ou une fonction membre est accessible directement ou pas.

Par défaut, en JAVA, toutes les propriétés et méthodes sont accessibles directement. Il faut donc préciser explicitement les conditions d'accès pour chaque propriété et chaque méthode. Pour cela, il existe trois mots-clés :

- **`public`** - Après ce mot clé, toutes les données ou fonctions membres sont accessibles.
- **`private`** - Après ce mot clé, toutes les données ou fonctions membres sont verrouillées et ne seront pas accessibles dans les classes dérivées.
- **`protected`** - Après ce mot clé, toutes les données ou fonctions membres sont verrouillées mais sont néanmoins accessibles dans les classes dérivées.

> *La distinction entre **`private`** et **`protected`** n'est visible que dans le cas de la déclaration de nouvelles classes par héritage. Ce concept sera abordé ultérieurement dans ce cours.*

Afin d'implémenter correctement le concept d'encapsulation, il convient de verrouiller l'accès aux propriétés et de les déclarer **`private`**, tout en maintenant l'accès aux méthodes en les déclarant **`public`**.

Exemple :

```
class Salarie
{
   private int m_nMatricule;
   private int m_nCategorie;
   private int m_nService;
   private String  m_strNom;
   private double m_dSalaire;

   public String calculSalaire()
   {
      return "Le salaire de " + m_strNom +
             " est de " + m_dSalaire + ".";
   }
```

```
}
```

2.2.2 Fonctions d'accès aux propriétés

Si les propriétés sont verrouillées, on ne peut plus y avoir accès à l'aide de l'opérateur **.** . Il faut donc créer des méthodes dédiées à l'accès aux propriétés pour chacune d'elles. Ces méthodes doivent permettre un accès dans les deux sens :

- pour connaître la valeur de la propriété. Ces méthodes sont appelées méthodes ***Get***. Elles peuvent être considérées comme des messages, que l'on enverrait à un objet, et qui signifierait "*Quelle est la valeur de ta propriété untel ?*". La réponse de l'objet, dont la valeur retournée par la méthode ***Get*** doit être cette valeur.

  Par exemple, pour la propriété **m_nMatricule**, déclarée **int**, la fonction ***Get*** sera déclarée de la façon suivante :

```
public int matricule()
{
   return m_nMatricule;
}
```

  Cette fonction pourra être utilisée dans la fonction **main**, par exemple :

```
Salarie sal = new Salarie();
int nMatr = sal.m_nMatricule;
int nMatr = sal.matricule();
```

- pour modifier la valeur d'une propriété. Ces méthodes sont appelées méthodes ***Set***. Elles peuvent être considérées comme des messages, que l'on enverrait à un objet, et qui signifierait "*Maintenant la valeur de ta propriété est de tant*". Cette méthode ne retourne aucune réponse. Par contre, un paramètre de même nature que la propriété doit lui être indiquée.

  Par exemple, pour la propriété **m_nMatricule**, déclarée **int**, la fonction ***Set*** sera déclarée de la façon suivante :

```
public void matricule(int nMatr)
{
   m_nMatricule = nMatr;
}
```

  Cette fonction pourra être utilisée dans la fonction **main**, par exemple :

```
Salarie sal = new Salarie();
sal.m_nMatricule = 5;
sal.matricule(5);
```

  L'intérêt de passer par des fonctions ***Set*** est de pouvoir y localiser des contrôles de validité des paramètres passés pour assurer la cohérence de l'objet, en y déclenchant des exceptions par exemple. La sécurisation des classes sera abordée ultérieurement dans ce cours.

2.2.3 Conventions d'écriture

En JAVA, par convention, les méthodes d'accès ***Get*** et ***Set*** porte le même nom que la propriété. Par exemple, **matricule** pour la propriété **m_nMatricule**. Ce qui oblige à écrire deux méthodes qui ont le même nom. Ceci est possible, en JAVA, à condition que la signature de ces méthodes soit différente. Ce qui est le cas puisque

la fonction ***Get*** n'a pas de paramètre et renvoie un résultat, alors que la fonction ***Set*** a un paramètre et est déclarée **`void`**.

2.3 TRAVAIL À RÉALISER

A partir du travail réalisé au T.P. N° 1, modifier la classe **`Salarie`** pour :

- protéger les propriétés et en interdire l'accès de l'extérieur de l'objet (l'accès aux fonctions membres doit toujours être possible),
- créer les méthodes d'accès aux propriétés.
- Modifier la fonction **`main`** pour tester ces fonctions.

2.4 POUR EN SAVOIR PLUS

Pour d'autres informations sur l'encapsulation et l'accès aux propriétés en JAVA, lire :

JAVA (Le Macmillan)
Ch. 8 : Les classes

T.P. N° 3 - CONSTRUCTION ET DESTRUCTION

3.1 OBJECTIFS

- Constructeurs et destructeur des objets
- Propriétés et méthodes de classe

3.2 CE QU'IL FAUT SAVOIR

3.2.1 Constructeurs

Quand une instance d'une classe d'objet est créée au moment de l'instanciation d'une variable avec **new**, une fonction particulière est exécutée. Cette fonction s'appelle le ***constructeur***. Elle permet, entre autre, d'initialiser chaque instance pour que ses propriétés aient un contenu cohérent.

Jusqu'à présent, chaque propriété était initialisée par un appel de la fonction ***Set*** correspondante. Supposons qu'un contrôle de cohérence soit effectué systématiquement à chaque appel d'une fonction ***Set***. Tant que les propriétés ne sont pas toutes initialisées, il est possible qu'une exception soit générée (lors de l'initialisation de la première propriété par exemple) alors que l'initialisation de toutes les propriétés aurait abouti à objet cohérent.

Les constructeurs permettent de résoudre ce type de problème. Il suffit de déclarer un ***constructeur*** qui initialise toutes les propriétés avant d'effectuer le contrôle de cohérence.

Un constructeur est déclaré comme les autres fonctions membres à deux différences près :

- Le nom de l'identificateur du constructeur est le même nom que l'identificateur de la classe.
- Un constructeur ne renvoie pas de résultat.

Exemple :

```
class Salarie
{
   private int m_nMatricule;
   private int m_nCategorie;
   private int m_nService;
   private String  m_strNom;
   private double m_dSalaire;

   public Salarie(int nMat, int nCatg, int nServ,
                  String strNom, double dSal)
   {
      m_nMatricule = nMat;
      m_nCategorie = nCatg;
      m_nService = nServ;
      m_strNom = strNom;
      m_dSalaire = dSal;
   }
```

```
};
```

3.2.2 Surcharge des constructeurs

Il peut y avoir plusieurs constructeurs pour une même classe, chacun d'eux correspondant à une initialisation particulière. Tous les constructeurs ont le même nom mais se distinguent par le nombre et le type des paramètres passés (cette propriété s'appelle ***surcharge*** en JAVA). Quand on crée une nouvelle classe, il est indispensable de prévoir tous les constructeurs nécessaires. Deux sont particulier :

Constructeur d'initialisation

Ce constructeur permet de procéder à une instanciation en initialisant toutes les propriétés, la valeur de celles-ci étant passée dans les paramètres.

Pour la classe **`Salarie`**, ce constructeur est déclaré comme ceci :

```
class Salarie
{
   private int m_nMatricule;
   private int m_nCategorie;
   private int m_nService;
   private String  m_strNom;
   private double m_dSalaire;

   public Salarie(int nMat, int nCatg, int nServ,
                  String strNom, double dSal)
   {
      m_nMatricule = nMat;
      m_nCategorie = nCatg;
      m_nService = nServ;
      m_strNom = strNom;
      m_dSalaire = dSal;
   }
};
```

Cela va permettre d'instancier la classe **`Salarie`** de la façon suivante :

```
public static void main (String[] args)
{
   Salarie sal = new Salarie(2, 2, 10,
                             "Michel ARDANT", 17500.00);
   // ...
}
```

Constructeur par défaut

Un constructeur par défaut existe déjà pour chaque classe si aucun autre constructeur n'est déclaré. A partir du moment où le constructeur d'initialisation de la classe **`Salarie`** existe, il devient impossible de déclarer une instance comme on l'a fait dans le T.P. précédent :

```
public static void main (String[] args)
{
   Salarie sal = new Salarie();
   // ...
}
```

Pour qu'une telle déclaration, sans paramètre d'initialisation, soit encore possible, il faut créer un ***constructeur par défaut***. En fait ce n'est réellement indispensable que si une instanciation de l'objet, avec des valeurs par défaut pour ses propriétés, a un sens.

Pour la classe **Salarie**, on peut s'interroger sur le sens des valeurs par défaut des propriétés. A titre d'exemple, ce constructeur serait déclaré comme ceci :

```
public Salarie()
{
   m_nMatricule = 1;
   m_nCategorie = 3;
   m_nService = 10;
   m_strNom = "Toto";
   m_dSalaire = 6500.00;
}
```

3.2.3 Propriétés de classe

Jusqu'à présent, les propriétés déclarées étaient des ***propriétés d'instance***. C'est à dire que les propriétés de chaque objet, instancié à partir de la même classe, peuvent avoir des valeurs différentes.

Supposons donc maintenant, que dans la classe **Salarie**, il soit nécessaire de disposer d'un compteur d'instance, dont la valeur serait le nombre d'instances en cours à un instant donné.

En JAVA, il est possible de créer des ***propriétés de classe***. Leur valeur est la même pour toutes les instances d'une même classe. Pour déclarer une telle propriété, on utilise le mot-clé **static**. Par exemple, dans la classe **Salarie**, le compteur d'instance pourrait être déclaré :

```
class Salarie
{
   private static int m_nCount = 0;
};
```

La propriété de classe **m_nCount**, initialisée à 0 lors de sa déclaration, sera incrémentée de 1 dans tous les constructeurs développés pour la classe **Salarie**. Sa valeur sera le nombre d'instances valides à un instant donné.

3.2.4 Méthodes de classe

Comme pour les autres propriétés déclarées **private**, il est nécessaire de créer les méthodes d'accès associées. Pour ce compteur, seule la méthode ***Get*** est nécessaire. Cependant, comme les propriétés de classe sont partagées par toutes les instances de la classe, le fait d'envoyer un message ***Get*** pour obtenir leur valeur n'a pas de sens. En reformulant, on pourrait dire que le message ***Get*** à envoyer pour obtenir le nombre d'instances de classe **Salarie** ne doit pas être envoyée à une instance donnée de cette classe, mais plutôt à la classe elle même. De telles méthodes sont appelées ***méthodes de classe***.

Une méthode de classe est déclarée par le mot-clé **static**. Pour la méthode ***Get*** d'accès au compteur d'instance on déclarerait :

```
public static int count()
{
    return m_nCount;
}
```

L'appel à une méthode **static** est sensiblement différent aux appels standard. En effet, ce n'est pas à une instance particulière que le message correspondant doit être envoyé, mais à la classe.

Dans la fonction **main**, par exemple, si l'on veut affecter à une variable le nombre d'instances de la classe Salarie, cela s'écrirait :

```
public static void main (String[] args)
{
    Salarie sal = new Salarie(2, 2, 10,
                              "Michel ARDANT", 17500.00);
    // ...
    int nInst = sal.count();
    int nInst = Salarie.count();
}
```

3.2.5 Destructeur

En JAVA la destruction d'un objet n'est pas synchrone. Cela signifie que l'on ne peut pas déclencher explicitement la destruction d'un objet. Les instances sont automatiquement détruites lorsqu'elles ne sont plus référencées. Le programme qui se charge de cette tâche s'appelle le ***Garbage Collector*** ou, en français, le ***ramasse poubelles***.

Or si l'on veut que le compteur d'instances soit à jour, il est nécessaire de connaître le moment ou les instances sont détruites pour décrémenter la variable **m_nCount**.

C'est pour cela que le ***ramasse poubelle*** en voie un message à chaque instance avant qu'elle soit détruite, et ceci quelle que soit la classe. Il suffit d'écrire la méthode de réponse à ce message, c'est la méthode **finalize**. Cette méthode s'appelle ***destructeur***. C'est , bien sûr, une méthode d'instance de la classe que l'on est en train de développer. Dans la classe **Salarie** par exemple, elle se déclare OBLIGATOIREMENT de la façon suivante :

```
protected void finalize()
{
    m_nCount--;
}
```

3.2.6 Constructeur et retour de méthode

Quand une fonction renvoie comme résultat une instance, en fait, la valeur de retour ne peut pas être l'instance elle-même, mais seulement une référence sur celle-ci. Ce qui risque de générer des effets de bord gênants. C'est pourquoi, il faut s'assurer de renvoyer une référence sur une nouvelle instance correspondant au résultat. Ce problème est difficile à mettre en évidence pour une classe comme la classe **Salarie**. Mais imaginons une classe **Fraction** pour laquelle il faut développer un opérateur de calcul pour effectuer la somme de deux fractions. Dans l'exemple ci-

dessous, **f1** et **f2** sont instanciées par l'instruction **new**. Le message **somme** est envoyé à **f1** avec **f2** comme paramètre correspondant à l'opérande d'addition.

```
public static void main (String[] args)
{
    Fraction f1 = new Fraction(1, 3);
    Fraction f2 = new Fraction(2, 5);
    // ...
    Fraction fRes = f1.somme(f2);
}
```

Mais qu'en est-il de **fRes**. Pour que **fRes** soit instanciée, cela ne peut se passer que dans la méthode **somme**. C'est pourquoi, dans le code de la méthode **somme**, il est indispensable ne procéder à une instanciation avec **new** en utilisant le constructeur d'initialisation :

```
public class Fraction
{
    private long m_nNum;
    private long m_nDen;

    public Fraction(int nNum, int nDen)
    {
        m_nNum = nNum;
        m_nDen = nDen;
    }

    public Fraction somme(Fraction fr)
    {
        return new Fraction(m_nNum*fr.m_nDen+m_nDen*fr.m_nNum,
                            m_nDen*fr.m_nDen);
    }
```

3.3 TRAVAIL À RÉALISER

- A partir du travail réalisé au T.P. N° 2, implémenter les constructeurs et le destructeur de la classe **Salarie**.
- Afin de mettre en évidence les rôles respectifs des constructeurs et du destructeur, implémenter ceux-ci pour qu'ils affichent un message à chaque fois qu'ils sont exécutés.
- Implémenter un compteur d'instances pour la classe **Salarie**.
- Modifier le jeu d'essai pour tester ces fonctions.

T.P. N°4 - L'HÉRITAGE

4.1 OBJECTIFS

- Généralités sur l'héritage
- Dérivation de la classe racine **`Object`**

4.2 CE QU'IL FAUT SAVOIR

4.2.1 L'héritage

Le concept d'héritage est l'un des trois principaux fondements de la Programmation Orientée Objet, le premier étant l'encapsulation vu précédemment dans les T.P. 1 à 3 et le dernier étant le polymorphisme qui sera abordé plus loin dans ce document.

L'héritage consiste en la création d'une nouvelle classe dite ***classe dérivée*** à partir d'une classe existante dite ***classe de base*** ou ***classe parente***.

L'héritage permet trois choses essentielles :

- récupérer le comportement standard d'une classe d'objet (classe parente) à partir de propriétés et des méthodes définies dans celles-ci,
- ajouter des fonctionnalités supplémentaires en créant de nouvelles propriétés et méthodes dans la classe dérivée,
- modifier le comportement standard d'une classe d'objet (classe parente) en surchargeant certaines méthodes de la classe parente dans la classe dérivée.

Ce concept permet un gain de temps important pour les tâches de codage :

- Le programmeur dispose d'une bibliothèque d'objets standards qu'il peut modifier à sa guise pour les besoins de son application. Par exemple, pour développer une ***applet*** JAVA, l'API JAVA fournit une multitude de classes de base (Applet, Fenêtre, objets d'interface, etc.).
- Pour la maintenance des applications, il est possible de corriger le comportement anormal d'un objet en créant une classe dérivée à partir de la classe de cet objet et en surchargeant les méthodes boguées. Il n'est même pas nécessaire d'en avoir les sources.

Exemple :

```
class ClasseA
{
    // Propriétés de la classe A
    public int m_DataA;
    // Méthodes de la classe A
    public int fonctionA1()
    {
        // Code de la méthode fonctionA1
    }
    public int fonctionA2()
    {
        // Code de la méthode fonction A2
```

```
    }
}

class ClasseB extends ClasseA
{
    // Propriétés de la classe B
    public int m_DataB;
    // Méthodes de la classe B
    public int fonctionA2()
    {
        // Code de la méthode fonctionA2
    }
    public int fonctionB1()
    {
        // Code de la méthode fonctionB2
    }
}
```

Dans cet exemple :

- **ClasseA** est la ***classe parente***.
- **ClasseB** est une ***classe dérivée*** de la classe **ClasseA**.
- **m_DataA** est une ***propriété*** de la classe **ClasseA**. Par héritage, **m_DataA** est aussi une ***propriété*** de la classe **ClasseB**.
- **m_DataB** est une ***propriété*** de la classe **ClasseB** (mais pas de **ClasseA**).
- **fonctionA1** est une ***méthode*** de la classe **ClasseA**. Par héritage, c'est aussi une ***méthode*** de la classe **ClasseB**.
- **fonctionB1** est une ***méthode*** de la classe **ClasseB** (mais pas de **ClasseA**).
- **fonctionA2** est une ***méthode*** des classes **ClasseA** et **ClasseB**. Mais le fait de la déclarer de nouveau comme méthode de la classe **ClasseB** en fait une fonction différente. Lors de l'appel de la fonction **fonctionA2**, ce sont les fonctions respectives de **ClasseA** et **ClasseB** qui seront utilisée pour les instances respectives de ces classes.

4.2.2 Protection des propriétés et des méthodes

En plus des mots-clés **public** et **private** décrits dans les chapitres précédents, il est possible d'utiliser un niveau de protection intermédiaire de propriétés et des méthodes par le mot-clé **protected**. Les exemples ci-dessous permettent d'illustrer ces différents niveaux de protection.

Exemple pour la déclaration des classes **ClasseA** et **ClasseB** :

```
class ClasseA
{
    // Propriétés de la classe A
    private int m_DataA1;
    protected int m_DataA2;
    public int m_DataA3;

    // Méthodes de la classe A
    public int fonctionA()
    {
        m_DataA1 = 5;       // 1
        m_DataA2 = 6;
        m_DataA3 = 7;
        m_DataB = 8;     // 2
```

```
    }
}

class ClasseB extends ClasseA
{
    // Propriétés de la classe B
       public int m_DataB;
    // Méthodes de la classe B
    public int fonctionB()
    {
       m_DataA1 = 15;      // 3
       m_DataA2 = 16;
       m_DataA3 = 17;
       m_DataB = 18;
    }
};
```

Exemple pour l'utilisation d'instances des classes **ClasseA** et **ClasseB** :

```
public static void main (String[] args)
{
    ClasseA ca = new ClasseA();
    ClasseB cb = new ClasseB();

    ca.m_DataA1 = 5;// 4
    ca.m_DataA2 = 6;// 5
    ca.m_DataA3 = 7;// 6

    cb.m_DataA1 = 5;// 7
    cb.m_DataA2 = 6;// 8
    cb.m_DataA3 = 7;// 9
    cb.m_DataB = 8;
}
```

Dans les exemples ci-contre :

- **m_DataA1**, **m_DataA2** et **m_DataA3** sont des propriétés de la classe **ClasseA** respectivement **private**, **protected** et **public**.
- **fonctionA** est une méthode de **CClasseA** et **fonctionB** une méthode de **ClasseB**.

1 **m_DataA1**, **m_DataA2** et **m_DataA3** sont des propriétés de **ClasseA**. Elles sont donc directement accessibles dans toutes les méthodes de **ClasseA**, donc dans **fonctionA**.

2 **m_DataB** est propriété de **ClasseB** (classe dérivée de **ClasseA**). Une propriété d'une classe dérivée est inaccessible dans la classe de base.

3 **m_DataA1** est déclarée **private** dans la classe **ClasseA**. Elle fait bien partie des composantes de la classe **ClasseB** mais est inaccessible dans toutes les méthodes de cette classe.

4 **m_DataA1** est déclarée **private** dans la classe **ClasseA**. Elle est inaccessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseA**. Il n'y a pas ici de distinction entre **private** et **protected**.

5 **m_DataA2** est déclarée **protected** dans la classe **ClasseA**. Elle est inaccessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseA**. Il n'y a pas ici de distinction entre **private** et **protected**.

6 **m_DataA3** est déclarée **public** dans la classe **ClasseA**. Elle est donc directement accessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseA**.

7 **m_DataA1** est déclarée **private** dans la classe **ClasseA**. Elle est inaccessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseB**.

8 **m_DataA2** est déclarée **protected** dans la classe **ClasseA**. Elle est inaccessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseB**.

9 **m_DataA3** est déclarée **public** dans la classe **ClasseA**. Elle est donc directement accessible dans toutes les fonctions qui utilisent des instances de la classe **ClasseB**.

4.2.3 Compatibilité et affectation

Lors des affectations (opérateurs =), il est possible de mélanger des instances de classes différentes. Cependant, il y a quelques règles à observer :

```
ClasseA caObj;
ClasseB cbObj;
```

- **ClasseA** est une classe de base.
- **ClasseB** est une classe dérivée de **ClasseA**.
- **caObj** est une instance de la classe **ClasseA**.
- **cbObj** est une instance de la classe **ClasseB**.

- L'instruction **caObj = cbObj;** est légale. Elle affecte à **caObj** l'objet référencé par **cbObj**. Ceci est possible car une instance de **ClasseB** est aussi une instance de **ClasseA**. Il n'y pas de conversion ici au sens propre. L'objet référencé par **caObj** reste une instance de **ClasseB**. Une variable d'une ***classe de base*** est compatible avec les variables de toutes ses ***classes dérivées***.

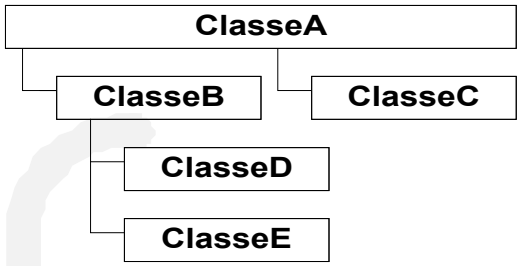
- L'instruction **cbObj = caObj;** génère une erreur de compilation. Cependant, si **caObj** contient une référence sur une instance de **ClasseB**, une telle affactation peut être forcée par un ***casting*** (changement de type) : **cbObj = (ClassB)pcaObj;**

4.2.4 Mode de représentation

Le concept d'héritage peut être utilisé pratiquement à l'infini. On peut créer des classes dérivées à partir de n'importe quelle autre classe, y compris celles qui sont déjà des classes dérivées.

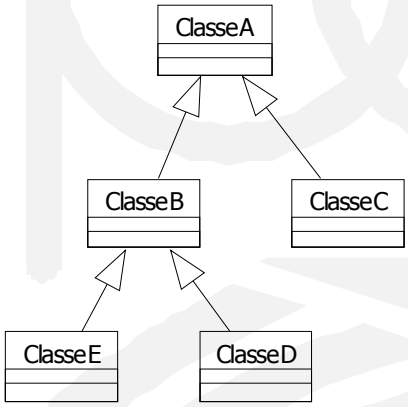
Supposons que **ClasseB** et **ClasseC** soient des classes dérivées de **ClasseA** et que **ClasseD** et **ClasseE** soient des classes dérivées de **ClasseB**. Les instances de

la classe **`ClasseE`** auront des données et des fonctions membres communes avec les instances de la classe **`ClasseB`**, voire de la classe **`ClasseA`**. Si les dérivations sont effectuées sur plusieurs niveaux, une représentation graphique de l'organisation de ces classes devient indispensable. Voici la représentation graphique de l'organisation de ces classes :



Le diagramme ci-dessus constitue la représentation graphique de la ***hiérarchie de classes*** construite à partir de **`ClasseA`**.

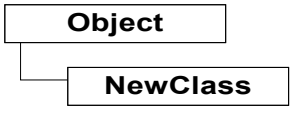
Dans le cadre de la conception orientée objet, la méthode UML (United Modeling Language) propose une autre représentation graphique d'une telle hiérarchie :



4.2.5 Insertion d'une classe dans une hiérarchie

Bien que l'on ait opéré comme cela depuis le début de ce cours, la création d'une nouvelle classe indépendante et isolée n'a pratiquement aucun intérêt en Programmation Orientée Objet. Afin de rendre homogène le comportement des instances d'une nouvelle classe il est important de l'insérer dans une bibliothèque de classe existante. Cette bibliothèque est fournie sous la forme d'une ***hiérarchie de classes*** construite à partir d'une ***classe racine***.

En JAVA, il est impossible de créer une classe isolée. En effet, lorsqu'on crée une nouvelle classe sans mentionner de classe de base, c'est la classe **`Object`** , la classe racine de toutes les classes JAVA, qui est utilisée.



`NewClass` est une nouvelle classe insérée dans la hiérarchie de classes construite à partir de **`Object`**.

JAVA dispose d'une hiérarchie de classes normalisées (Java API Hierarchy), proposant une multitude de classes prêtes à l'emploi, ou à dériver.

4.2.6 Insertion d'une nouvelle classe à partir de Object

La dérivation d'une classe par rapport à une classe prédéfinie doit obéir à un certain nombre de règles définies dans la documentation de la ***classe de base*** à dériver. L'une de ces contraintes est la surcharge OBLIGATOIRE de certaines méthodes de la classe de base avec la même signature. Pour dériver la classe **`Object`**, cas le plus courant, cela se résume à ne réécrire que quelques méthodes :

Méthode finalize

Il s'agit du destructeur dont on déjà décrit le rôle dans le chapitre précédent.

Méthode toString

Cette méthode doit créer une chaîne de caractères (instance de la classe **`String`**) qui représente les propriétés des instances de la classe. Par exemple, pour la classe **`Fraction`**, la chaîne de caractères représentant une fraction pourrait être **`"nnnn/dddd"`** où **`nnnn`** et **`dddd`** correspondraient respectivement aux chiffres composant le numérateur et le dénominateur de la fraction.

```
class Fraction
{
   private long m_nNum;
   private long m_nDen;

   public String toString()
   {
      return m_nNum + "/" + m_nDen;
   }
}
```

La méthode **`toString`** est appelée lorsqu'une conversion implicite d'un objet en chaîne de caractères est nécessaire comme c'est le cas pour la fonction **`println`**, par exemple :

```
Fraction fr = new Fraction(1,2);
System.out.println("fr = " + fr);
```

Méthode equals

Cette méthode doit répondre VRAI si deux instances sont rigoureusement égales. Deux conditions sont à vérifier :

- Il faut que les deux objets soient de la même classe. Le paramètre de la méthode **`equals`** étant de type **`Object`**, notre instance peut donc être comparée à une instance d'une classe quelconque dérivée de **`Object`**. L'instruction **`instanceof`** permet de connaître à quelle classe appartient une instance donnée.

- Il faut que leurs propriétés soient identiques (mais pas forcément égales). Par exemple, pour la classe **`Fraction`**, on peut dire que deux fractions sont égales si le produit des *extrêmes* est égal au produit des mo*y*ens. Ce qui peut s'écrire :

```
class Fraction
```

```
{
   private long m_nNum;
   private long m_nDen;

   public boolean equals(Object o)
   {
      return o instanceof Fraction &&
         m_nNum * ((Fraction)o).m_nDen ==
            m_nDen * ((Fraction)o).m_nNum;
   }
}
```

4.2.7 Constructeur et héritage

Chaque constructeur d'une classe dérivée doit obligatoirement appeler le constructeur équivalent de la classe de base. Si **BaseClass** est une classe de base appartenant à la hiérarchie de classe construite à partir de **Object**, si **NewClass** est une classe dérivée de **BaseClass**, cela se fait de la façon suivante en utilisant le mot-clé **super** :

Constructeur par défaut

```
public NewClass()
{
   super()
   // Initialisation des nouvelles données membres
}
```

Constructeur d'initialisation

```
public NewClass (Valeurs initiales des
                    propriétés de NewClass)
{
   super(Valeur initiales des propriétés
          de la classe BaseClass);
   // Initialisation des nouvelles données membres
}
```

4.2.8 Appel aux méthodes de la classe de base

Lors de la surcharge d'une méthode de la classe de base dans une classe dérivée, il peut être utile de reprendre les fonctionnalités standard pour y ajouter les nouvelles fonctionnalités. Pour ne pas avoir à réécrire le code de la classe de base (dont on ne dispose pas forcément), il est plus simple de faire appel à la méthode de la classe de base. Pour cela on utilise le mot-clé **super** avec une syntaxe différente à celle utilisée pour le constructeur :

```
public String methode1(String s)
{
   // Calcul du résultat standard par appel
   // à la méthode de la classe de base
   String strRes = super.methode1(s);
   // Calcul du nouveau résultat en modifiant
   // le résultat standard
   strRes = strRes + "," + m_nData2;
   return strRes;
}
```

4.3 TRAVAIL À RÉALISER

Pour ce travail, à réaliser à partir de ce qui a été produit au T.P. N° 3, on procédera en deux temps :

- Ajouter à la classe **`Salarie`** les méthodes **`equals`** et **`toString`**. La règle d'égalité pour la classe **`Salarie`** peut s'énoncer de la façon suivante : deux salariés sont égaux s'ils ont le même numéro de matricule et le même nom. **`toString`** doit renvoyer toutes les propriétés séparées par des virgules.

- En passant par TOUTES les étapes décrites précédemment, créer une classe **`Commercial`** en dérivant la classe **`Salarie`**. Cette classe aura 2 propriétés supplémentaires pour calculer la commission :
  - chiffre d'affaire **`m_dChiffreAffaire double`**
  - commission en % **`m_pcCommission int`**

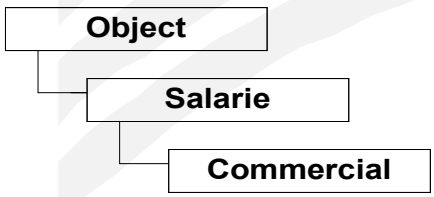
    - Créer les deux constructeurs standards de la classe **`Commercial`**. Ne pas oublier d'appeler les constructeurs équivalents de la classe de base.

    - Créer les méthodes d'accès aux propriétés supplémentaires.

    - Surcharger la méthode **`calculSalaire`** pour calculer le salaire réel (fixe + commission).

    - Surcharger les autres méthodes de la classe de base pour lesquelles on jugera nécessaire de faire ainsi.

    - Créer une fonction **`main`** permettant de tester les classes **`Salarie`** et **`Commercial`**.



T.P. N°5 - COLLECTIONS

5.1 OBJECTIFS

- Utilisation des classes collections

5.2 CE QU'IL FAUT SAVOIR

5.2.1 Les classes Collections

La plupart des applications modernes (Word, Excel, etc.) gèrent les données sur le mode ***Open/Save*** qui peut se résumer par les opérations suivantes :

- Lecture (***Open***) du document à traiter en chargeant à partir d'un fichier la totalité des données le composant en mémoire.
- Traitement du document et ajoutant, modifiant ou supprimant des données.
- Enregistrement (***Save***) du document en écrivant la totalité des données dans le fichier.

On oppose le mode ***Open/Save*** au mode ***Open/Close*** :

- Ouverture (***Open***) d'un fichier sans charger les données.
- Traitement des données une à une dans le cycle transactionnel suivant :
  - lecture d'une donnée à partir du fichier,
  - modification de la donnée,
  - écriture de la donnée modifiée dans le fichier.
- Fermeture (***Close***) du fichier.

Le mode ***Open/Save*** est très adapté au traitement des données de la bureautique orientée "Document".

Le mode ***Open/Close*** est plutôt adapté au traitement des bases de données.

Le mode ***Open/Save*** impose de pouvoir accéder à chacune des données élémentaires en mémoire. Il est donc indispensable de conserver une référence sur chaque élément composant un document.

La Programmation Orientée Objet propose des classes d'objets particulières appelées ***Collections*** dont le rôle est de contenir d'autres objets. Il existe plusieurs types de ***collections*** :

- Les tableaux,
- Les listes chaînées,
- Les dictionnaires,
- Les arbres.

En JAVA, seuls les trois premiers types de ***collections*** sont implémentés :

Les tableaux

Les tableaux contiennent des éléments, chacun d'eux étant repéré par son indice. En JAVA, il existe une manière très simple de créer des tableaux sans faire référence aux classes ***collections*** :

```
int[] aTab = new int[20];
for (int i = 0; i < 20; i++)
{
   aTab[0] = i + 1900;
}

String[] aStr = new String[5];
aStr[0] = "André";
aStr[1] = "Mohamed";
aStr[2] = "Marc";
aStr[3] = "Francis";
aStr[4] = "Paul";

Salarie[] aSal = new Salarie[5];
aSal[0] = new Salarie(16, 1, 10, "CAUJOL", 10900.00);
aSal[1] = new Salarie(5, 1, 10, "DUMOULIN", 15600.00);
aSal[2] = new Salarie(29, 3, 20, "AMBERT", 5800.00);
aSal[3] = new Salarie(20, 2, 20, "CITEAUX", 8000.00);
aSal[4] = new Salarie(34, 2, 30, "CHARTIER", 7800.00);
```

Le problème de ce type de tableaux réside en deux points :

- Tous les éléments du tableau doivent être de même type.
- Le nombre d'éléments que peut contenir un tableau est limité au moment de la déclaration. Cela sous-entend que le programmeur connaît, au moment de l'écriture du programme, le nombre maximum d'éléments que doit contenir le tableau.

Les ***collections*** permettent de contenir un nombre quelconque d'éléments, leur taille augmentant dynamiquement au fur et à mesure des besoins.

Les éléments d'une ***collection*** peuvent être de classe (de taille) différentes. Les instances rangées dans les collections doivent être de classe dérivée de **`Object`**. Cela exclu les types standards tels que **`int`**, **`long`**, **`double`**, etc.. Cette contrainte peut être levée en utilisant les classes numériques dérivées de la classe **`Number`**.

En JAVA, les tableaux sont implémentés par la classe **`Vector`**.

```
Vector tab = new Vector();
```

Les listes chaînées

Les listes chaînées contiennent des éléments dont il n'est pas nécessaire de connaître en permanence les références. Les listes maintiennent une référence sur le premier et le dernier élément de la liste. On peut accéder à n'importe quel élément de la liste en la parcourant dans un sens ou dans l'autre.

En JAVA, les listes chaînées sont également implémentées par la classe **`Vector`**.

```
Vector list = new Vector();
```

D'autres collections comme les ***piles*** et les ***files d'attente*** sont des cas particuliers de ***listes chaînées***. Elles sont implémentées en JAVA par la classe **`Stack`**.

Les dictionnaires

Les dictionnaires contiennent des éléments, chacun d'eux étant repéré par une clef. En JAVA, les dictionnaires sont implémentés par la classe **`Hashtable`**[1].

```
Hashtable dict = new Hashtable();
```

5.2.2 Insertion d'un objet dans une collection

L'insertion d'une instance dans une ***collection*** suppose toujours que celle-ci a été créée par instruction **`new`**. La manière de le faire diffère d'un type de collection à l'autre :

Les tableaux

On considère la classe **`Vector`**, comme si elle contenait des objets indicés à partir de 0. Trois méthodes permettent d'insérer un nouvel élément dans un tableau :

- **`addElement`** ajoute un élément à la fin du tableau. La taille du tableau augmente automatiquement.
- **`setElementAt`** remplace un élément à un endroit précis du tableau. Si on ne veut pas perdre la référence de l'élément qui se trouve déjà dans le tableau à cet endroit, il faut l'obtenir avec la méthode **elementAt** .
- **`insertElementAt`** ajoute un élément à un endroit précis du tableau. La taille du tableau est automatiquement incrémentée. Cette méthode provoque un décalage de tous les indices supérieurs.

Les Listes

On considère la classe **`Vector`**, en tant que liste chaînée. Les méthodes décrites ci-dessus sont utilisées, mais dans un contexte différent :

- **`insertElementAt`** ajoute un élément au début de la liste, à condition que l'indice passé en paramètre soit égal à 0.
- **`addElement`** ajoute un élément à la fin de la liste.

Les dictionnaires)

Une seule méthode permet d'insérer un nouvel élément dans un dictionnaire :

- **`put`** remplace un élément pour une clef précise du dictionnaire. Si on ne veut pas perdre la référence de l'élément qui se trouve déjà dans le dictionnaire, il faut l'obtenir avec la méthode **`get`** .

5.2.3 Repérage d'un objet dans une collection

Les tableaux

On considère la classe **`Vector`**, comme si elle contenait des objets indicés à partir de 0. Pour cela on utilise :

- **`elementAt`** permet de retrouver la référence d'un élément à un indice donné.

[1] La classe **`Dictionary`** existe mais n'est pas directement instanciable. Le programmeur doit dériver cette classe pour créer sa propre classe dictionnaire en fonction du type de clé utilisé pour repérer les objets. La classe **`Hashtable`** est une classe dérivée de **`Dictionary`**, prête à l'emploi, mais dont la clé est de type **`Object`**, c'est à dire à un niveau de généricité très élevé.

Les Listes

On considère la classe **`Vector`**, en tant que liste chaînée. On repère un élément dans une liste pas sa position. La position d'un élément ne peut être connue que lors du parcours de la liste (voir itération d'une collection). Outre la méthode décrite précédemment, deux autres méthodes permettent de repérer un élément dans une liste :

- **`firstElement`** permet de retrouver la référence d'un élément à un indice donné permet de retrouver la référence du premier élément de la liste.
- **`lastElement`** permet de retrouver la référence du dernier élément de la liste.

Les dictionnaires

On repère un élément dans un dictionnaire par sa clef :

- **`get`** permet de retrouver la référence d'un élément pour une clef donnée.

5.2.4 Itération d'une collection

Pour effectuer un traitement particulier pour chaque élément d'une collection, il est indispensable de faire une ***itération*** de la collection, c'est-à-dire de parcourir toute la collection élément par élément. En JAVA, la manière de procéder est la même, quel que soit le type de collection.

```
for (Enumeration e = coll.elements() ; e.hasMoreElements() ;)
{
   Object o = e.nextElement());
   // Traitement particulier pour l'instance o
   // extraite de la collection
}
```

`coll` est une collection quelconque (instance de **`Vector`** ou de **`Hashtable`**, par exemple).

`e` est une requête, effectué sur la collection par la méthode **`elements`**. **`elements`**. est une méthode qui existe dans toutes les classes collections.

Un index est initialisé par **`elements`** et automatiquement incrémentée par la méthode d'itération **`nextElement`** **`nextElement`** renvoie en résultat, élément par élément, la référence des objets contenus dans la collection. La boucle d'itération fonctionne tant qu'il y a des éléments dans la requête **`e`**, information renvoyé par la méthode **`hasMoreElement`**.

5.2.5 Suppression d'un objet dans une Collection

Supprimer un objet d'une ***collection*** ne signifie pas qu'il soit détruit. Cela signifie uniquement que l'instance en question ne fait plus partie de la ***collection***. C'est le ramasse-poubelle de JAVA qui se charge de la destruction effective de l'objet quand celui-ci n'est plus référencé.

Les listes et les tableaux)

Trois méthodes de la classe **`Vector`** permettent de supprimer des éléments :

- **`removeElementAt`** permet de supprimer un élément d'une liste ou d'un tableau à une position donnée. Cela suppose l'on connaît l'indice de l'élément à supprimer.

- **`removeElement`** permet de supprimer un objet d'une liste ou d'un tableau dont on connaît la référence.
- **`removeAllElements`** permet de supprimer tous les éléments d'une liste ou d'un tableau.

Les dictionnaires

Deux méthodes de la classe **`Hashtable`** permettent de supprimer des éléments :

- **`remove`** permet de supprimer un élément d'un dictionnaire pour une clé donnée.
- **`clear`** permet de supprimer tous les éléments d'un dictionnaire.

5.3 TRAVAIL À RÉALISER

A partir de la classe **`Salarie`** implémentée dans les T.P. précédents, dans la fonction **`main`**,

- Créer une instance d'un dictionnaire dans lequel on va ranger des instances de la classe **`Salarie`** repérées par leur numéro de matricule.
- Créer au moins cinq instances de la classe **`Salarie`** et les insérer dans le dictionnaire.
- Faire l'itération du dictionnaire pour afficher son contenu.

T.P. N° 6 - POLYMORPHISME

6.1 OBJECTIFS

- Polymorphisme
- Fonctions virtuelles
- Classes génériques

6.2 CE QU'IL FAUT SAVOIR

6.2.1 Polymorphisme

Considérons l'exemple suivant.

```
Salarie sal = new Commercial(7, 2, 10, "PEDROL Jean",
                             8000.00,45000.00, 12);
String str = sal.calculSalaire();
```

La variable **sal** est déclarée de type **Salarie**, mais elle est instanciée à partir de la classe **Commercial**. Une telle affectation est correcte et ne génère pas d'erreur de compilation. La question que l'on doit se poser maintenant est la suivante : lors de l'envoi du message **calculSalaire** à **sal**, quelle est la méthode qui va être exécutée ? La méthode **calculSalaire** de la classe **Salarie** qui calcul le salaire uniquement à partir du salaire de base, ou la méthode **calculSalaire** de la classe **Commercial** qui prend en compte les propriétés commission et chiffre d'affaire de la classe **Commercial** ? Un simple test va permettre de mettre en évidence que c'est bien la méthode **calculSalaire** de la classe **Commercial** qui est exécutée. JAVA *sait* et *se souvient* comment les objets ont été instanciés pour pouvoir appeler la bonne méthode.

En Programmation Orientée Objet, c'est ce que l'on appelle le concept de ***Polymorphisme***. Pour toutes les instances de salarié, quelles soient instanciées à partir de la classe de base **Salarie** ou d'une classe dérivée comme **Commercial**, il faut pouvoir calculer le salaire. C'est le ***comportement polymorphique***. Par contre le calcul ne se fait pas de façon identique pour les salariés normaux et les commerciaux.

6.2.2 Méthodes virtuelles

L'ambiguïté est levée du fait que la résolution du lien (entre le programme appelant et la méthode) ne se fait pas au moment de la compilation, mais pendant l'exécution en fonction de la classe qui a été utilisée pour l'instanciation. On parle de ***méthode virtuelle***. **calculSalaire** est une méthode virtuelle qui défini un comportement polymorphique sur la hiérarchie de classe construite sur **Salarie**.

Pour associer un comportement polymorphique à une méthode, il suffit de la surcharger la méthode la classe de base, avec exactement la même signature, c'est-à-dire le même nom, le même type de résultat, le même nombre de paramètres, de mêmes types, dans le même ordre.

La classe racine **Object** est elle-même un polymorphisme. Les méthodes **finalize**, **toString** et **equals**, surchargées dans les T.P. précédents pour les

classes **Salarie** et **Commercial**, sont des fonctions virtuelles qui définissent un comportement polymorphique pour toutes les classes dérivées de **Object**.

6.2.3 Classes génériques

Dans certains cas, lors de l'élaboration d'une hiérarchie de classe ayant un comportement polymorphique, il peut être intéressant de mettre en facteur le comportement commun à plusieurs classes, c'est-à-dire créer une ***classe générique***. Il est probable que le fait de créer une instance de cette classe ne corresponde pas à une réalité au niveau conceptuel. C'est le cas de la classe racine **Object** : Il est absurde de créer une instance de la classe **Object** qui, par ailleurs, définit le comportement commun à toutes les classes JAVA. C'est le cas également de la classe **Dictionary** qui définit le comportement commun à toutes les classes dictionnaire, mais qui ne peut être instanciée. Une telle classe est déclarée **abstract** :

```
abstract public class Dictionary
{
   // Définitions des membres de la classe
}
```

Essayer d'instancier une classe **abstract** avec l'instruction **new** va générer une erreur de compilation.

6.2.4 Méthodes abstraites

L'intérêt de ce type de classe est de pouvoir déclarer toutes les méthodes virtuelles qui vont définir le comportement polymorphique des classes dérivées, sans avoir à les développer. De telles fonctions sont également déclarées **abstract**.

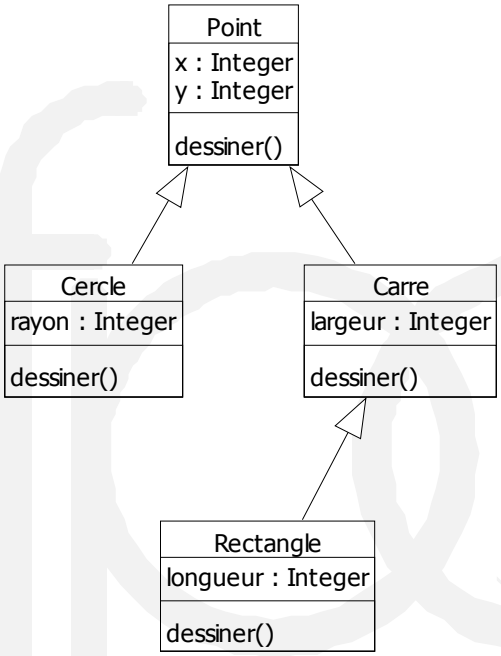
```
abstract public class Dictionary
{
   public abstract Object put(Object key, Object value);
   public abstract Object get(Object key);
   public abstract Enumeration elements();
   public abstract Object remove(Object key);

   // Définitions des autres membres de la classe
}
```

Le développement effectif appartient au programmeur qui va dériver cette classe de base et surcharger ces méthodes pour définir explicitement le comportement spécifique de la classe dérivée. C'est de cette façon qu'a procédé le programmeur de la classe **Hashtable**, dérivée de la classe **Dictionary**. La classe **Hashtable** peut alors être instanciée.

6.2.5 Un peu de conception

L'exemple le plus courant de polymorphisme qui est donné est la hiérarchie de classe basée sur **Point** permettant de définir une classe pour chaque type d'objets graphiques que l'on peut rencontrer dans un logiciel de dessin (Cercle, Carré, Rectangle, etc.). La classe de base **Point** correspond en fait à la position de l'objet sur le document. Le comportement polymorphique de cette hiérarchie repose sur la méthode virtuelle **Dessiner** pour que chaque type d'objet graphique sache se dessiner correctement.

Afin d'en faciliter la description, voici la représentation U.M.L. de ce polymorphisme. En U.M.L., l'héritage est symbolisé par une flèche :



Cette implémentation est possible dans tous les langages de Programmation Orientée Objet, et est correcte quant aux fonctionnalités :

- La classe **`Cercle`** possède bien un centre de coordonnées **`x`**, **`y`** ***par héritage*** de **`Point`** et un rayon **`rayon`**.
- La classe **`Carre`** possède bien la position de son coin supérieur gauche de coordonnées **`x`** et **`y`** ***par héritage*** de **`Point`** et la largeur de son côté **`largeur`**.
- La classe **`Rectangle`** possède bien la position de son coin supérieur gauche de coordonnées **`x`** et **`y`** par ***héritage*** de **`Point`**/**`Carre`**, une **`largeur`** ***héritée*** de **`Carre`** et une **`longueur`**.

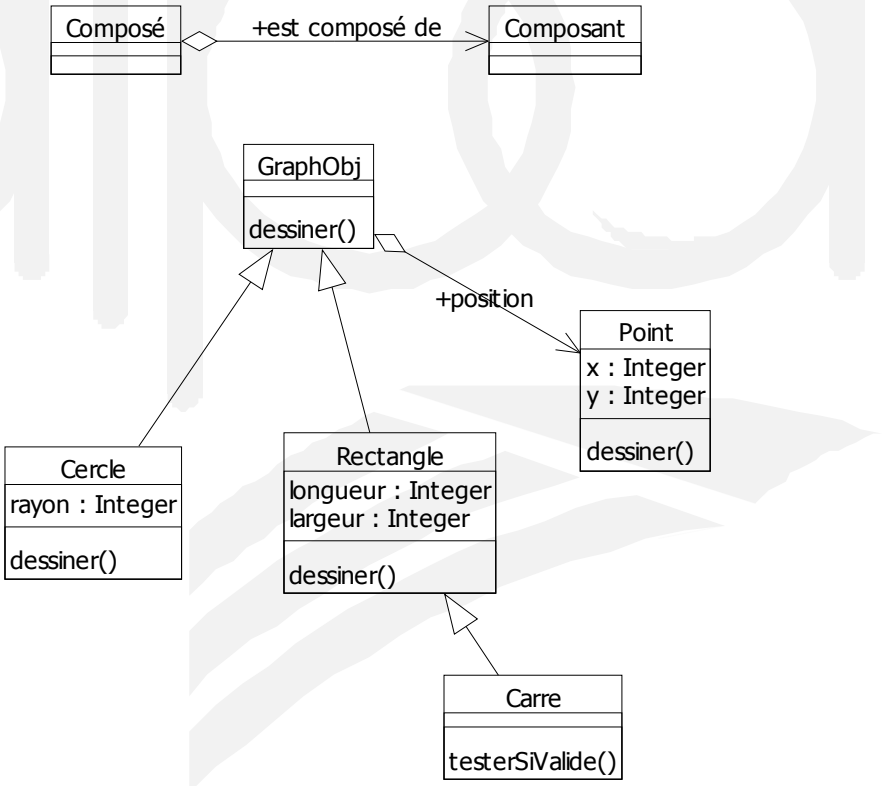
Cependant, au niveau conceptuel, cette implémentation est absurde. En effet, on ne peut pas considérer un cercle ou un carré comme un point particulier, ni un rectangle comme un carré particulier (ce serait plutôt le contraire). En utilisant l'héritage de façon absurde on aboutit rapidement à utiliser des techniques d'héritage multiple (dérivation d'une classe à partir de plusieurs classes de base) que les puristes de la P.O.O. considèrent un peu comme le vilain **GoTo** dans la programmation structurée. Il est vrai que l'utilisation de l'héritage multiple est souvent révélatrice d'un défaut d'analyse dans la conception de la hiérarchie de classes. De toute façon, l'héritage multiple est impossible en JAVA.

On oppose la technique d'héri***t***age à la technique d'***agrégation***. Cette dernière consiste à utiliser comme données membres d'une classe, des instances d'autres classes.

Pour le problème décrit ci-dessus, il aurait mieux valu construire un polymorphisme d'objets graphiques basé sur une classe générique **`GraphObj`**:

- Créer une classe générique **`GraphObj`** dont l'une des données membres est, par ***agrégation***, une instance de **`Point`**.
- Déclarer dans celle-ci une méthode abstraite (**`dessiner`**) permettant de dessiner les objets graphiques.
- Créer les classes **`Rectangle`** et **`Cercle`** par héritage de **`GraphObj`**.
- Créer la classe **`Carre`** par héritage de **`Rectangle`** et en ajoutant une clause de cohérence (méthode **`testerSiValide`**) pour tester si la largeur et la longueur sont égales.

On peut représenter en U.M.L. le polymorphisme **`GraphObj`**. En U.M.L., l'héritage est symbolisé par un petit losange :



Mais quand faut-il utiliser l'agrégation plutôt que l'héritage ? La tentation des débutants de la programmation orientée objet est d'utiliser l'héritage parce que *cela fait plus objet*. Et le choix entre les deux techniques n'est pas évident.

En fait, il existe un moyen mnémotechnique pour faire ce choix. Considérons deux classes d'objet.

- Si la relation entre ces deux classes peut être exprimée sous la forme ***"est une sorte de…"***, alors l'héritage s'impose. Le carré ***est une sorte de*** rectangle dont la largeur et la longueur sont égales.

- Si la relation entre ces deux classes peut être exprimée sous la forme ***"est composé de…"***, il faut utiliser l'agrégation. Les objets graphiques de notre application de dessin ***sont composés d'***un point correspondant à la position de cet objet.

6.3 TRAVAIL À RÉALISER

- Modifier la fonction **`main`** développée dans le T.P. précédent en insérant quelques instances de la classe **`Commercial`** dans le dictionnaire.
- Vérifier le fonctionnement du *Polymorphisme* lié à la fonction **`calculSalaire`**.

T.P. N° 7 - ROBUSTESSE ET SÉCURITÉS

7.1 OBJECTIFS

- Concept de robustesse
- Contrôle de cohérence
- Aide au deboguage
- Gestion des exceptions

7.2 CE QU'IL FAUT SAVOIR

7.2.1 La robustesse

La robustesse est le dernier concept de la Programmation Orientée Objet étudié ici. Il consiste en l'écriture de programmes exempte de bogues. Pour cela, il est indispensable d'intégrer la gestion des exceptions aux classes d'objets développées et éventuellement de créer de nouvelles classes d'exceptions pour gérer les cas critiques propres à la nouvelle classe.

7.2.2 Les exceptions

Les exceptions et l'utilisation de celles-ci, à travers les mots-clés **`try`**, **`catch`**, **`throw`** et **`throws`**, ont déjà été étudiées dans le chapitre 5 du support de cours **JAVA_ASF**. Il convient donc de se reporter à ce chapitre avant d'aborder la suite de ce cours.

7.2.3 Génération d'exception

La robustesse d'une classe va donc commencer par le recensement de toutes les méthodes de la classe susceptibles de générer des exceptions. Cela se fait en deux étapes :

- La première est similaire à la démarche utilisée dans le chapitre 5 du cours **JAVA_ASF**. Elle consiste à recenser tous les appels à des méthodes utilisées dans le développement des méthodes la nouvelle classe et, à partir de la documentation technique, identifier toutes les exceptions qui peuvent être générées. Dans ce cas, soit on transmet les exceptions telles quelles au programme appelant par l'instruction **`throws`**, soit on capture les exceptions par **`try-catch`** et l'on génère une autre exception par l'instruction **`throw`**.

- La deuxième consiste à associer à d'éventuels contrôles d'intégrité de l'objet, la génération d'exceptions par l'instruction **`throw`**.

Ces deux phases peuvent entraîner la création de nouvelles classes d'exception, permettant de mieux distinguer les différents cas particuliers qui sont susceptible de se produire.

En considérant la classe **`Salarie`**, par exemple, on peut constater qu'aucune méthode ne fait appel à des instructions susceptibles de générer des exceptions. Par contre, il est possible d'ajouter des contrôles d'intégrité sur certaines propriétés. Par exemple, le salaire (propriété **`m_nSalaire`**) peut être contrôlé pour que sa valeur soit toujours positive pour ne pas générer de calculs aberrants. Il faut donc examiner

toutes les méthodes de la classe **Salarie** qui peuvent modifier la propriété **m_nSalaire**. Cet examen révèle que seules, deux méthodes modifie la valeur de cette propriété, la méthode d'accès ***Set*** associée à cette propriété et le constructeur d'initialisation de la classe.

Cas du constructeur d'initialisation de la classe

Le constructeur d'initialisation de la classe **Salarie** reçoit 5 paramètres correspondant respectivement aux 5 propriétés de la classe. L'un d'eux, le salaire, doit être contrôlé pour vérifier que sa valeur soit positive. En cas d'erreur sur le paramètre, une exception est générée par l'instruction **throw**.

```
public Salarie(int nMat, int nCatg, int nServ,
                String strNom, double dSal)
    throws SalarieException
{
    m_nMatricule = nMat;
    m_nCategorie = nCatg;
    m_nService = nServ;
    m_strNom = strNom;
    m_dSalaire = dSal;

    if (dSal < 0)
    {
        SalarieException se = new SalarieException(this);
        throw se;
    }

        m_nCount++;
}
```

Plusieurs remarques :

- Les propriétés de la classe sont initialisées AVANT le test de validité du paramètre **dSal**. Cela permet de passer en paramètre du constructeur de l'exception le salarié en cours de création (**this**) et ainsi d'afficher éventuellement dans un message d'erreur les propriétés du salarié dont la construction a causé l'exception.

- Une nouvelle classe d'exception (**SalarieException**) doit être créée pour prendre en compte les spécificités des exceptions générées par la classe **Salarie**. L'une de ces spécificités est ici l'existence d'un constructeur, pour la classe **SalarieException,** qui accepte un **Salarie** en paramètre. La création de nouvelle classe d'exception sera étudiée dans le paragraphe suivant.

- L'instruction **throw** effectue un branchement inconditionnel dans le bloc **catch** correspondant au traitement relatif à la capture de l'exception générée (**se**). La construction de l'objet n'aboutira donc pas puisqu'elle sera incomplète. L'objet en cours de création ne sera donc jamais référencé dans une variable et sera détruit immédiatement par le ramasse-poubelle. De plus l'incrémentation du compteur d'instance ne sera pas effectuée.

- Le constructeur étant susceptible de générer une exception de classe **SalarieException**, il faut le signaler au programme appelant avec l'instruction **throws**.

Cas d'une méthode d'accès** Set **à une propriété

La méthode d'accès ***Set*** à la propriété **m_dSalaire** de la classe **Salarie** reçoit un paramètre correspondant à cette propriété. Ce paramètre doit être contrôlé pour vérifier que sa valeur soit positive. En cas d'erreur, une exception est générée par l'instruction **throw**.

```
public void salaire(double dSal)
    throws SalarieException
{
    if (dSal < 0)
    {
        SalarieException se = new SalarieException(this);
        throw se;
    }
    m_dSalaire = dSal;
}
```

Plusieurs remarques :

- La propriété **m_dSalaire** est modifiée APRES le test de validité du paramètre **dSal**. En cas d'erreur sur le paramètre, l'instruction **throw** effectue un branchement inconditionnel dans le bloc **catch** correspondant au traitement relatif à la capture de l'exception générée (**se**). La propriété ne sera donc pas modifiée.

- Une nouvelle classe d'exception (**SalarieException**) doit être créée pour prendre en compte les spécificités des exceptions générées par la classe **Salarie**. L'une de ces spécificités est ici l'existence d'un constructeur, pour la classe **SalarieException,** qui accepte un **Salarie** en paramètre. La création de nouvelle classe d'exception sera étudiée dans le paragraphe suivant.

- La méthode d'accès **salaire** étant susceptible de générer une exception de classe **SalarieException**, il faut le signaler au programme appelant avec l'instruction **throws**.

7.2.4 Création d'une nouvelle classe d'exception

La création d'une nouvelle classe d'exception s'impose quand on veut identifier ou distinguer des exceptions propres à une classe donnée. Cela se passe en trois phases :

- Dériver la classe **Exception** pour en créer une nouvelle par héritage. En fonction de la nature exacte de l'exception générée, il est possible de dérivée une autre classe dérivée elle-même d'**Exception**.

- Créer un constructeur pour la nouvelle classe, compatible avec les arguments que l'on souhaite afficher dans un message d'erreur. Ce constructeur doit lui-même appeler le constructeur de la classe de base. Si cette dernière est la classe **Exception**, il faut convertir le paramètre du constructeur de la nouvelle classe

en chaîne de caractères, car il n'existe, pour cette classe, qu'un seul constructeur dont le paramètre est de type **`String`**.

- Surcharger la méthode **`toString`**. Cette méthode est utilisée pour afficher un message d'erreur en passant l'instance de l'exception dans la fonction **`println`** lors de la capture de celle-ci dans le bloc **`catch`**.

On pourrait créer la classe d'exception **`SalarieException`** de la façon suivante :

```
class SalarieException extends Exception
{
    SalarieException(Salarie sal)
    {
        super(sal.toString());
    }

    public String toString()
    {
        return super.toString()+
                    "\nLe salaire ne peut etre que positif.";
    }
}
```

5.3 TRAVAIL À RÉALISER

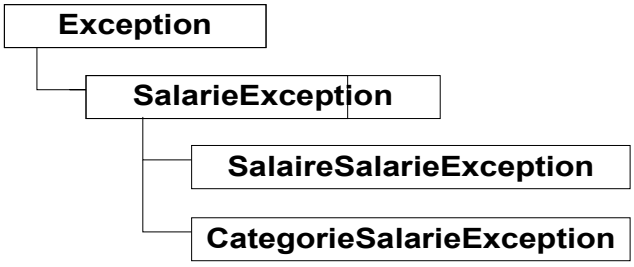
A partir du T.P. précédent implémenter le concept de robustesse en contrôlant l'intégrité de la classe **`Salarie`** en vérifiant :

- Que le salaire est toujours positif.
- Que la catégorie de salarié ne peut être que 1 (cadre), 2 (technicien) ou 3(employé).

Pour cela :

- Modifier les méthodes d'accès ***Set*** concernées pour générer les exceptions appropriées.
- Modifier le constructeur d'initialisation de la classe **`Salarie`** pour générer les exceptions appropriées.
- Créer les classes d'exceptions nécessaires pour permettre de distinguer les erreurs se produisant sur les différentes propriétés.

On peut suggérer la hiérarchie suivante :



CONCLUSION

Ici se termine la deuxième partie de ce cours. On peut en déduire une méthodologie de construction d'objets dont la démarche peut être résumée dans les étapes ci-dessous :

- Enumérer les données membres d'un objet.
- Protéger les données membres et créer les fonctions d'accès à ces propriétés.
- Créer, même s'ils ne sont pas indispensables, au moins les deux constructeurs (défaut, initialisation) et le destructeur.
- Enumérer les opérations à opérer sur l'objet (Entrées/sorties, calcul, comparaison, conversion, etc.).
- Enumérer les exceptions générées par les fonctions utilisées dans les méthodes le classe pour les traiter correctement.
- Identifier les contrôles nécessaires pour assurer l'intégrité des objets et créer les classes d'exceptions correspondantes.

INDEX ANALYTIQUE