T.P. N° 1

Salarie.java

```
1  public class Salarie
   {
2        int m_nMatricule;
         int m_nCategorie;
         int m_nService;
         String    m_strNom;
         double m_dSalaire;

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

Commentaires

1 En JAVA, il est impossible de déclarer une classe indépendamment d'une hiérarchie. Il est indispensable de dériver une classe existante. Si cette précision est omise (ce qui est le cas ici), la classe dérivée est la classe **Object**, qui constitue la racine de l'arbre de toutes les classes JAVA.

2 On commence par déclarer les propriétés de l'objet. Les identifiants de ces propriétés sont tous préfixés **m_**, ce qui permettra de mieux les distinguer des autre variables déclarées dans une fonction par exemple. Si aucune mention (**public**, **private** ou **protected**) n'est précisée, les propriétés sont considérée comme accessibles directement par les programmes. Ce fonctionnement n'est pas conforme au concept d'encapsulation. La protection des données membres sera traité dans le T.P. suivant.

3 Déclaration de la méthode **calculSalaire**. Cette méthode ne reçoit aucun paramètre, cependant, les parenthèses (vides) sont obligatoirement présentes. Cette méthode renvoie un résultat de type **String** (chaîne de caractères).

4 Le rupteur **return** permet à une méthode de renvoyer un résultat exploitable par le programme appelant. Ce résultat doit être de même type que le résultat de la fonction mentionnée dans sa déclaration, ici **String**. Les instructions qui suivent l'instructions **return** ne sont pas exécutées. Ici, il s'agit de la dernière instruction, ce qui masque le problème.

monApp.java

```
1  public class monApp
   {
2        public static void main (String[] args)
         {
3             Salarie sal = new Salarie();
```

```
4               sal.m_nMatricule = 1;
                sal.m_nCategorie = 2;
                sal.m_nService = 10;
                sal.m_strNom = "Michel ARDANT";
                sal.m_dSalaire = 17500.00;

5               System.out.println("Matricule = " + sal.m_nMatricule);
                System.out.println("Categorie = " + sal.m_nCategorie);
                System.out.println("Service = " + sal.m_nService);
                System.out.println("Nom = " + sal.m_strNom);
                System.out.println("Salaire = " + sal.m_dSalaire);

6               System.out.println(sal.calculSalaire());
        }
}
```

Commentaires

1 La classe application (ici **monApp**) se déclare comme toute les classes. Elle n'échappe pas aux règles standards qui régissent les classes en JAVA. Une particularité toutefois : elle doit obligatoirement comporter une méthode de nom **main** qui constitue le point d'entrée du programme.

2 La fonction **main** de la classe application peut être considérée comme la méthode de cette classe qui correspond à la réception d'un message envoyé par la machine virtuelle qui signifierait "démarre".

3 Instanciation de la classe **Salarie**. Elle se fait en deux temps. D'abord la déclaration d'une variable **sal** de type **Salarie**. Puis l'instanciation proprement dite par l'instruction **new** suivi du nom de la classe. Les parenthèse signifie qu'il s'agit en fait d'une fonction. Cette fonction est un ***constructeur*** de la classe. On étudiera cette notion dans le troisième T.P. de ce cours.

4 Initialisation des données membre. Utilisées comme ***LValue*** (à gauche du = d'affectation), cela signifie que les propriétés vont changer de valeur. Normalement, le concept d'encapsulation interdit cela. On ne doit accéder aux propriétés que par l'intermédiaire de méthodes particulières appelées méthodes ***Set***.

5 Affichage des données membres. Pour cela, il est nécessaire d'en obtenir la valeur. Cet accès direct ne correspond pas au concept d'encapsulation. On de doit accéder aux données membres par l'intermédiaire de méthodes particulières appelées méthodes ***Get***.

6 Utilisation de la méthode **calculSalaire**. L'utilisation d'une méthode peut être comparée à un envoi de message. Ici, l'application envoie le message **calculSalaire** à l'instance **sal** de la classe **Salarie**. Celle-ci répond en exécutant la fonction membre **calculSalaire()** et répond, par l'intermédiaire du résultat de la fonction retourné par **return**, par une chaîne de caractères contenant le nom et le salaire calculés à partir des données membres de l'instance **sal**.

*Ces deux fichiers se trouvent dans le répertoire **JOTP1**.*

T.P. N° 2

Salarie.java

```
1   public class Salarie
    {
2         private int m_nMatricule;

3         public int matricule()
          {
4               return m_nMatricule;
          }

5         public void matricule(int nMat)
          {
6               m_nMatricule = nMat;
          }


2         private int m_nCategorie;

3         public int categorie()
          {
4               return m_nCategorie;
          }

5         public void categorie(int nCatg)
          {
6               m_nCategorie = nCatg;
          }


2         private int m_nService;

3         public int service()
          {
4               return m_nService;
          }

5         public void service(int nServ)
          {
6               m_nService = nServ;
          }


2         private String  m_strNom;

3         public String nom()
          {
4               return m_strNom;
          }

5         public void nom(String strNom)
          {
6               m_strNom = strNom;
          }


2         private double m_dSalaire;

3         public double salaire()
          {
4               return m_dSalaire;
          }

5         public void salaire(double dSal)
          {
6               m_dSalaire = dSal;
          }
```

```
7
      public String calculSalaire()
      {
            return "Le salaire de " + m_strNom +
                        " est de " + m_dSalaire + ".";
      }
}
```

Commentaires

1 La classe **Salarie** est déclarée publique. Cela permet de l'instancier n'importe où dans un programme.

2 Les propriétés sont déclarées **private**. L'accès en est alors interdit en dehors de la classe.

3 Les méthodes ***Get*** sont déclarées **public**. Elles doivent être de même type que la propriété à laquelle elles donnent accès *en lecture*. Elles n'ont pas besoin de paramètres. Il faut cependant écrire les parenthèses car ce sont des fonctions.

4 Les méthodes ***Get*** doivent renvoyer un résultat de même type que la propriété à laquelle elles donnent accès.

5 Les méthodes ***Set*** sont déclarées **public**. Elles doivent être de type **void** car elle n'ont pas besoin de renvoyer un résultat. Le paramètre passé doit être de même type que la propriété à laquelle elles donnent accès *en écriture*.

6 Le traitement des méthodes ***Set*** correspond à l'affectation du paramètre passé à la donnée membre. Un contrôle sur la validité du paramètre peut y être ajouté en déclenchant une exception. La sécurisation des objets sera abordée ultérieurement.

7 A l'intérieur des méthodes d'une classe, on peut accéder directement aux propriétés de celle-ci.

monApp.java

```
public class monApp
{
      public static void main (String[] args)
      {
            Salarie sal = new Salarie();

1           sal.matricule(1);
            sal.categorie(2);
            sal.service(10);
            sal.nom("Michel ARDANT");
            sal.salaire(17500.00);

2           System.out.println("Matricule = " + sal.matricule());
            System.out.println("Categorie = " + sal.categorie());
            System.out.println("Service = " + sal.service());
            System.out.println("Nom = " + sal.nom());
            System.out.println("Salaire = " + sal.salaire());

            System.out.println(sal.calculSalaire());
```

```
        }
}
```

Commentaires

1 Les données membres de la classes **Salarie** ayant été déclarées **private**, il est impossible de les utiliser dans les méthodes d'une autre classe. Ce qui est le cas de la fonction **main** de la classe **monApp**. Dans le sens de l'affectation, on utilise les fonctions ***Set*** en leur passant en paramètre la valeur de la propriété à modifier.

2 De même, il est impossible d'accéder aux propriétés pour en obtenir leur valeur. On utilise pour cela les fonctions ***Get*** dont le résultat peut être affiché ou utilisé dans des expressions. Le fait que les fonctions ***Get*** et ***Set*** aient le même nom n'est pas gênant en JAVA car leurs signatures est différente. En effet, les fonctions ***Set*** sont déclarées **void** (donc sans résultat) et on un paramètre (la valeur de la propriété à modifier) alors que les fonctions ***Get*** n'ont pas de paramètre et renvoient un résultat (la valeur actuelle de la propriété).

*Ces deux fichiers se trouvent dans le répertoire **JOTP2**.*

T.P. N° 3

Salarie.java

```
public class Salarie
{
      private int m_nMatricule;

      public int matricule()
      {
            return m_nMatricule;
      }

      public void matricule(int nMat)
      {
            m_nMatricule = nMat;
      }


      private int m_nCategorie;

      public int categorie()
      {
            return m_nCategorie;
      }

      public void categorie(int nCatg)
      {
            m_nCategorie = nCatg;
      }


      private int m_nService;

      public int service()
      {
            return m_nService;
      }

      public void service(int nServ)
      {
            m_nService = nServ;
      }


      private String  m_strNom;

      public String nom()
      {
            return m_strNom;
      }

      public void nom(String strNom)
      {
            m_strNom = strNom;
      }



      private double m_dSalaire;

      public double salaire()
      {
            return m_dSalaire;
      }

      public void salaire(double dSal)
      {
            m_dSalaire = dSal;
      }




1     private static int m_nCount = 0;

2     public static int count()
      {
```

```
                return m_nCount;
        }


3       public Salarie()
        {
                System.out.println("Création d'un salarié par défaut.");
                m_nMatricule = 1;
                m_nCategorie = 1;
                m_nService = 10;
                m_strNom = "Toto";
                m_dSalaire = 6500.00;
4               m_nCount++;
        }

5       public Salarie(int nMat, int nCatg, int nServ,
                                String strNom, double dSal)
        {
                System.out.println("Création du salarié " + strNom);
                m_nMatricule = nMat;
                m_nCategorie = nCatg;
                m_nService = nServ;
                m_strNom = strNom;
                m_dSalaire = dSal;
4               m_nCount++;
        }

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


6       protected void finalize()
        {
                System.out.println("Destruction du salarié " + m_strNom);
7               m_nCount--;
        }
}
```

Commentaires

1 Pour implémenter un compteur d'instance, il faut déclarer une propriété (**m_nCount**) de classe pas le mot-clé **static**. Cette propriété est initialisée à 0. Cette propriété est partagée par toutes les instances de la classe **Salarie**.

2 La propriété **m_nCount**, correspondant au compteur d'instance, est déclarée **private**. Il faut donc créer les fonctions d'accès à cette propriété. Cette propriété étant initialisée 0 lors de sa déclaration et mise à jour par les constructeurs et le destructeur (voir points **4** et **7**), seule la fonction ***Get*** est nécessaire. Cette fonction est également déclarée **static**, ce qui en fait une méthode de classe. Le message devra donc être envoyé à la classe elle-même, pas à une instance particulière.

3 Constructeur par défaut. Pour la classe **Salarie**, un constructeur par défaut n'offre pas d'intérêt réel. Car quelles sont les valeurs par défaut à affecter aux propriétés ? Ce constructeur n'existe ici qu'à titre d'exemple.

4 Mise à jour du compteur d'instance. La propriété **m_nCount** est incrémentée de 1 dans les deux constructeurs.

5 Constructeur d'initialisation. Chaque propriété est initialisée à la valeur passée en paramètre du constructeur. Il y a donc autant de paramètres à passer dans le constructeur que de propriétés à initialiser.

6 Destructeur. Le ramasse poubelle envoie un message à chaque instance avant de les détruire. Ce qui permet de mettre à jour certaines données comme notre compteur d'instance. En réponse à ce message, la méthode **finalize** est exécutée. Reformulé autrement, la méthode **finalize** est appelée implicitement par le ramasse poubelle avant la destruction de l'instance. Cette méthode doit impérativement est déclarée avec cette signature. Une signature différente correspondrait à une autre méthode (en JAVA la surcharge est possible) qui n'aurait aucun rapport avec le message de destruction, et qui ne serait donc jamais exécutée (sauf par un appel explicite).

7 Mise à jour du compteur d'instance. A chaque appel du destructeur, la propriété **m_nCount** est décrémentée de 1.

monApp.java

```
public class monApp
{
     public static void main (String[] args)
     {
          Salarie sal1 = new Salarie();
1         System.out.println("Nombre d'instances de Salarie : " +
                                Salarie.count());

          Salarie sal2 = new Salarie(2, 2, 10, "Michel ARDANT",
                                     17500.00);

1         System.out.println("Nombre d'instances de Salarie : " +
                                Salarie.count());

          System.out.println("Matricule = " + sal1.matricule());
          System.out.println("Categorie = " + sal1.categorie());
          System.out.println("Service = " + sal1.service());
          System.out.println("Nom = " + sal1.nom());
          System.out.println("Salaire = " + sal1.salaire());

          System.out.println("Matricule = " + sal2.matricule());
          System.out.println("Categorie = " + sal2.categorie());
          System.out.println("Service = " + sal2.service());
          System.out.println("Nom = " + sal2.nom());
          System.out.println("Salaire = " + sal2.salaire());

          System.out.println(sal1.calculSalaire());
          System.out.println(sal2.calculSalaire());
2    }
}
```

Commentaires

1 Pour obtenir la valeur de la propriété **m_nCount** de la classe **Salarie**, on utilise la méthode d'accès **count**. Cette méthode est une méthode de classe (déclarée **static**). Il faut dont l'associer à l'identifieur de classe (**Salarie**) et non pas à un identifieur d'instance (**sal1** ou **sal2**).

2 Si on met un point d'arrêt ici et sur la première instruction de la méthode **finalize**, on constatera que le programme s'arrête d'abord à cet endroit. Ce qui

signifie que les deux instances créées existent toujours. C'est normal, car elles sont toujours référencées par les variables **sal1** et **sal2**

*Ces deux fichiers se trouvent dans le répertoire **JOTP3**.*

T.P. N° 4

Salarie.java

1

```
public class Salarie
{
      private int m_nMatricule;

      public int matricule()
      {
            return m_nMatricule;
      }

      public void matricule(int nMat)
      {
            m_nMatricule = nMat;
      }


      private int m_nCategorie;

      public int categorie()
      {
            return m_nCategorie;
      }

      public void categorie(int nCatg)
      {
            m_nCategorie = nCatg;
      }



      private int m_nService;

      public int service()
      {
            return m_nService;
      }

      public void service(int nServ)
      {
            m_nService = nServ;
      }



      private String  m_strNom;

      public String nom()
      {
            return m_strNom;
      }

      public void nom(String strNom)
      {
            m_strNom = strNom;
      }


      private double m_dSalaire;

      public double salaire()
      {
            return m_dSalaire;
      }

      public void salaire(double dSal)
      {
            m_dSalaire = dSal;
```

```
        }


        private static int m_nCount = 0;

        public static int count()
        {
             return m_nCount;
        }


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

             m_nCount++;
        }

        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;

             m_nCount++;
        }

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


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


3       public String toString()
        {
             return     m_nMatricule + "," +
                        m_nCategorie + "," +
                        m_nService + "," +
                        m_strNom + "," +
                        m_dSalaire;
        }


4       public boolean equals(Object p1)
        {
             return p1 instanceof Salarie &&
                      m_nMatricule == ((Salarie)p1).m_nMatricule &&
                      m_strNom.compareTo(((Salarie)p1).m_strNom) == 0;
        }
}
```

Commentaires

1 Il n'est pas nécessaire de préciser **extends Object**, car par défaut, JAVA considère que l'on dérive cette classe.

2 La méthode **finalize**, traitée dans le T.P. précédent peut être aussi considérée comme une méthode de la classe de base **Object** qu'il est ici nécessaire de surcharger pour décrémenter le compteur d'instance.

3 **toString** renvoie comme résultat une chaîne de caractères composée par les différentes propriétés converties individuellement en chaînes de caractères, concaténées et séparées par le caractère virgule.

4 L'instance courante peut être comparée à n'importe qu'elle instance **p1** d'une classe quelconque dérivée de **Object**. Pour qu'il y ait égalité, compte tenu des règles de gestion adoptées, trois conditions sont nécessaires :

- Il faut que l'instance courante et **p1** soient de même classe, d'où utilisation de l'expression **instanceof Salarie** dont l'évaluation est VRAIE si **p1** est bien une instance de la classe **Salarie**.
- Il faut que les propriétés **m_nMatricule** de l'instance courante et de **p1** soient égales. Ici, un *casting* est nécessaire pour pouvoir accéder à la propriété **m_nMatricule**.
- Il faut que les propriétés **m_strNom** de l'instance courante et de **p1** soient égales. Ici, un *casting* est nécessaire pour pouvoir accéder à la propriété **m_strNom**.

Commercial.java

1

```
public class Commercial extends Salarie
{
      private double m_dChiffreAffaire;

      public double chiffreAffaire()
      {
            return m_dChiffreAffaire;
      }

      public void chiffreAffaire(double dChiffre)
      {
            m_dChiffreAffaire = dChiffre;
      }


      private int m_pcCommission;

      public int commission()
      {
            return m_pcCommission;
      }

      public void commission(int pcCommission)
      {
            m_pcCommission = pcCommission;
      }


      public Commercial()
      {
```

2

```
            super();
            m_pcCommission = 0;
            m_dChiffreAffaire = 0.0;
      }
```

```
        public Commercial(int nMat, int nCatg, int nServ,
                          String strNom, double dSal,
                          double dChiffre, int nCommission)
        {
3           super(nMat, nCatg, nServ, strNom, dSal);
            m_pcCommission = nCommission;
            m_dChiffreAffaire = dChiffre;
        }

        // SURCHARGE DES METHODES DE LA CLASSE DE BASE
        public String calculSalaire()
        {
4           return "Le salaire de " + nom() + " est de " +
                (salaire() + m_dChiffreAffaire * m_pcCommission / 100.0)
                 + ".";
        }


        public String toString()
        {
5           return super.toString() + "," +
                    m_pcCommission + "," +
                    m_dChiffreAffaire;
        }
    }
```

Commentaires

1 La classe **Commercial** est construite en dérivant la classe **Salarie**.

2 Utilisation de la méthode **super** pour un appel au constructeur par défaut de la classe **Salarie**.

3 Utilisation de la méthode **super** pour un appel au constructeur d'initialisation de la classe **Salarie**. Les paramètres correspondant aux propriétés de la classe **Salarie** sont passé au constructeur en question.

4 Les propriétés **m_strNom** et **m_dSalaire** de la classe **Salarie** étant déclarées **private**, sont inaccessible dans les méthodes de la classe **Commercial**. On est obligé ici d'utiliser les méthodes d'accès, ce qui n'est pas vraiment gênant. Cela aurait pu être résolu en déclarant **protected** toutes les propriétés de la classe **Salarie**. Ce qui obligerait à modifier la classe **Salarie**.

5 Appel à la fonction **toString** de la classe **Salarie** pour ne pas avoir à réécrire un code identique à ce qui avait été fait dans la classe de base et ajout des propriétés propres à la classe **Commercial**.

Il n'est pas nécessaire de réécrire les autres méthodes héritées des classes **Object** et **Salarie**.

- Le fonctionnement de **finalize** programmé dans la classe **Salarie,** dont le rôle est de décrémenter le compteur d'instance, convient car il comptera toutes les instances de **Salarie**, y compris les instances de **Commercial**.

- La méthode **equals** de la classe **Salarie** ne fait intervenir que les propriétés **m_strNom** et **m_nMatricule** qui sont des propriétés héritées de la classe de base.

monApp.java

```
public class monApp
{
     public static void main (String[] args)
     {
          Salarie sal1 = new Salarie(2, 2, 10, "Michel ARDANT",
17500.00);
          System.out.println("Nombre d'instances de Salarie : " +
Salarie.count());

          Commercial sal2 = new Commercial(3, 1, 20, "Isidore BAUTRELET",
10000.00, 102000.00, 7);
          System.out.println("Nombre d'instances de Salarie : " +
Salarie.count());

1         System.out.println("sal1 = [" + sal1 + "]");
          System.out.println("sal2 = [" + sal2 + "]");
          System.out.println(sal1.calculSalaire());
          System.out.println(sal2.calculSalaire());

2         Salarie sal3 = sal2;
          System.out.println("sal3 = [" + sal3 + "]");

3         if (sal2.equals(sal3))
          {
               System.out.println("sal2 est identique à sal3");
          }
     }
}
```

Commentaires

1 Le fait d'avoir réécrit les méthodes **toString** pour les classes **Salarie** et **Commercial** permet de passer des instances de ces classes en paramètres de fonctions telles que **println** qui exigent de paramètre de type **String**. Une conversion implicite des objets en chaînes de caractères est automatiquement opérée.

2 Bien que **sal2** référencie un objet de classe **Commercial**, une telle affectation est correcte. Même si **sal3** est une variable de type Salarie, elle n'en référencie pas moins un objet de classe **Commercial**, classe utilisée lors de l'instanciation de **sal2**.

3 Même si **sal2** est de type **Commercial** et **sal3** de type **Salarie**, on peut comparer les objets qu'elles référencent. En l'occurrence, comme **sal2** et **sal3** référencent le même objet, de part l'affectation effectuée au point **2**, l'égalité sera toujours vraie.

*Ces trois fichiers se trouvent dans le répertoire **JOTP4**.*

T.P. N° 5

monApp.java

```
1   import java.util.*;

    public class monApp
    {
        public static void main (String[] args)
        {
2           Hashtable dict = new Hashtable();

            Salarie s = new Salarie(16, 1, 10, "CAUJOL", 10900.00);
3           putInDictionary(dict, s);

            s = new Salarie(5, 1, 10, "DUMOULIN", 15600.00);
3           putInDictionary(dict, s);

            s = new Salarie(29, 3, 20, "AMBERT", 5800.00);
3           putInDictionary(dict, s);

            s = new Salarie(20, 2, 20, "CITEAUX", 8000.00);
3           putInDictionary(dict, s);

            s = new Salarie(34, 2, 30, "CHARTIER", 7800.00);
3           putInDictionary(dict, s);


4           for (Enumeration e = dict.elements() ; e.hasMoreElements() ;)
            {

5               Salarie sal = (Salarie)e.nextElement();
                System.out.println(sal);
                System.out.println(sal.calculSalaire());
            }
        }


3       public static void putInDictionary(Hashtable dict, Salarie s)
        {
6           Integer key = new Integer(s.matricule());
7           dict.put(key, s);
        }
    }
```

Commentaires

1 La classe **Hashtable** utilisée comme dictionnaire est définie dans le package **java.util**. Il faut donc importer ce package par l'instruction **import**.

2 Instanciation d'un dictionnaire de classe **Hashtable**. Au début cette collection ne contient aucun élément.

3 Comme l'opération d'insertion d'un salarié dans le dictionnaire est répétitive, une fonction **putInDictionary** a été créée dans ce but. Il prend deux paramètres, le dictionnaire et le salarié à y insérer.

4 On utilise une boucle **for** pour faire l'itération du dictionnaire. Pour cela, on initialise une **Enumeration** par un appel à la méthode **elements** affectée au dictionnaire.

5 L'appel de la méthode **nextElement** affectée à l'**Enumeration** permet de récupérer, élément par élément, tous les objets contenus dans la collection. **nextElement** renvoie une référence de type **Object**. Le programmeur doit assumer le *casting* pour effectuer la conversion en **Salarie**. Ce qui peut se faire sans risque, puisque les objets qui ont été insérés dans la collection étaient tous des instances de la classe **Salarie**.

6 Comme les dictionnaires de classe **Hashtable** ne peuvent accepter que des clefs instanciées à partir de classes dérivées de **Object**, on utilise la classe **Integer** pour instancier des clefs numériques à partir du N° de matricule du salarié.

7 Le salarie **s**, passé en paramètre de **putInDictionary** est placé dans le dictionnaire.

*Ce fichier se trouve dans le répertoire **JOTP5**.*

T.P. N° 6

monApp.java

```
import java.util.*;

public class monApp
{
      public static void main (String[] args)
      {
            Hashtable dict = new Hashtable();

            Salarie s = new Salarie(16, 1, 10, "CAUJOL", 10900.00);
            putInDictionary(dict, s);
1           s = new Salarie(5, 1, 10, "DUMOULIN", 15600.00);
            putInDictionary(dict, s);
            s = new Salarie(29, 3, 20, "AMBERT", 5800.00);
            putInDictionary(dict, s);
2           s = new Commercial(17, 2, 20, "ABRU", 6000.00,120000.0,7);
            putInDictionary(dict, s);
            s = new Salarie(20, 2, 20, "CITEAUX", 8000.00);
            putInDictionary(dict, s);
            s = new Salarie(34, 2, 30, "CHARTIER", 7800.00);
            putInDictionary(dict, s);
2           s = new Commercial(51, 2, 20, "BRUTI", 7500.00,150000.0, 11);
            putInDictionary(dict, s);


            for (Enumeration e = dict.elements() ; e.hasMoreElements() ;)
            {
                  Salarie sal = (Salarie)e.nextElement();
3                 System.out.println(sal);
4                 System.out.println(sal.calculSalaire());
            }

      }

      public static void putInDictionary(Hashtable dict, Salarie s)
      {
            Integer key = new Integer(s.matricule());
            dict.put(key, s);
      }
}
```

Commentaires

1 On peut utiliser la même varaible **s** pour instancier tous les **Salarie** et le **Commercial** puisque les références sur tous ces objets seront conservés dans la collection.

2 On peut utiliser une variable de type **Salarie** pour instancier un **Commercial** car **Commercial** est une classe dérivée de **Salarie**.

3 Il est impossible de savoir, à priori, de quelle classe, **Salarie** ou **Commercial**, est l'instance **sal** issue du dictionnaire,d'autant que l'itération d'un dictionnaire ne ramenne pas forcément les éléments dans le même ordre qu'ils y ont été rangés. Le fait de passer **sal** dans **println** va automatiquement provoquer une conversion automatique en chaîne de caractères en appelant la méthode **toString** de l'objet. Comme cette méthode défini un comportement

polymorphique pour toutes les classes dérivées de **Object**, c'est la méthode **toString** de la bonne classe qui sera utilisée.

4 La méthode **calculSalaire** défini un comportement polymorphique pour toutes les classes dérivées de **Salarie**. Là également, c'est la méthode de la bonne classe qui sera utilisée.

Résultat affiché

```
20,2,20,CITEAUX,8000.0
Le salaire de CITEAUX est de 8000.0.
51,2,20,BRUTI,7500.0,11,150000.0
Le salaire de BRUTI est de 24000.0.
29,3,20,AMBERT,5800.0
Le salaire de AMBERT est de 5800.0.
17,2,20,ABRU,6000.0,7,120000.0
Le salaire de ABRU est de 14400.0.
5,1,10,DUMOULIN,15600.0
Le salaire de DUMOULIN est de 15600.0.
16,1,10,CAUJOL,10900.0
Le salaire de CAUJOL est de 10900.0.
34,2,30,CHARTIER,7800.0
Le salaire de CHARTIER est de 7800.0.
```

*Ces trois fichiers se trouvent dans le répertoire **JOTP6**.*

T.P. N° 7

Salarie.java

```
public class Salarie
{
      private int m_nMatricule;
      public int matricule()
      {
            return m_nMatricule;
      }
      public void matricule(int nMat)
      {
            m_nMatricule = nMat;
      }

      private int m_nCategorie;
      public int categorie()
      {
            return m_nCategorie;
      }
      public void categorie(int nCatg)
1           throws CategorieSalarieException
      {
            if (nCatg < 1 || nCatg > 3)
            {
2                 CategorieSalarieException cse =
                                    new CategorieSalarieException(this);
                  throw cse;
            }
3           m_nCategorie = nCatg;
      }

      private int m_nService;
      public int service()
      {
            return m_nService;
      }
      public void service(int nServ)
      {
            m_nService = nServ;
      }

      private String  m_strNom;
      public String nom()
      {
            return m_strNom;
      }
      public void nom(String strNom)
      {
            m_strNom = strNom;
      }

      private double m_dSalaire;
      public double salaire()
      {
            return m_dSalaire;
      }
      public void salaire(double dSal)
4           throws SalaireSalarieException
      {
            if (dSal < 0)
            {
5                 SalaireSalarieException sse =
                                    new SalaireSalarieException(this);
                  throw sse;
            }
6           m_dSalaire = dSal;
      }
```

```
        private static int m_nCount = 0;
        public static int count()
        {
             return m_nCount;
        }

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

             m_nCount++;
        }

        public Salarie(int nMat, int nCatg, int nServ,
                                              String strNom, double dSal)
7            throws CategorieSalarieException, SalaireSalarieException
        {
8            m_nMatricule = nMat;
             m_nCategorie = nCatg;
             m_nService = nServ;
             m_strNom = strNom;
             m_dSalaire = dSal;

2            if (nCatg < 1 || nCatg > 3)
             {
                  CategorieSalarieException cse =
                                        new CategorieSalarieException(this);
                  throw cse;
             }

5            if (dSal < 0)
             {
                  SalaireSalarieException sse =
                                        new SalaireSalarieException(this);
                  throw sse;
             }

9            m_nCount++;
        }

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


        protected void finalize()
        {
             m_nCount--;
        }

        public String toString()
        {
             return     m_nMatricule + "," +
                        m_nCategorie + "," +
                        m_nService + "," +
                        m_strNom + "," +
                        m_dSalaire;
        }

        public boolean equals(Object p1)
        {
             return p1 instanceof Salarie &&
                     m_nMatricule == ((Salarie)p1).m_nMatricule &&
                     m_strNom.compareTo(((Salarie)p1).m_strNom) == 0;
        }
   }


10 class SalarieException extends Exception
```

```
11  {
        SalarieException(Salarie sal)
        {
            super(sal.toString());
        }

    }

12  class CategorieSalarieException extends SalarieException
    {

13      CategorieSalarieException(Salarie sal)
        {
            super(sal);
        }


14      public String toString()
        {
            return super.toString()+
                "\nLa categorie ne peut etre que 1, 2 ou 3.";
        }
    }

15  class SalaireSalarieException extends SalarieException
    {
16      SalaireSalarieException(Salarie sal)
        {
            super(sal);
        }

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

Commentaires

1 La fonction *Set* de la propriété **m_nCategorie** va contrôler si l'argument passé est bien 1,2 ou 3. Une exception de classe **CategorieSalarieException** peut être générée. Le programme appelant doit donc en être informé par l'instruction **throws**.

2 Si l'argument correspondant à la catégorie n'est pas 1, 2 ou 3, une exception de classe **CategorieSalarieException** est instanciée. L'objet est passé en paramètre du constructeur de l'exception pour que les informations concernant le salarié figurent dans le message d'erreur affiché lors du traitement de l'exception. **throw** provoque un branchement inconditionnel dans le bloc **catch** du programme appelant ou cette classe d'exception est capturée et traitée. Le reste de la fonction n'est donc pas exécuté.

3 Si tout ce passe bien, la propriété **m_nCategorie** est modifiée.

4 La fonction *Set* de la propriété **m_dSalaire** va contrôler si l'argument passé est bien positif. Une exception de classe **SalaireSalarieException** peut être générée. Le programme appelant doit donc en être informé par l'instruction **throws**.

5 Si l'argument correspondant au salaire n'est pas positif, une exception de classe **SalaireSalarieException** est instanciée. L'objet est passé en paramètre

du constructeur de l'exception pour que les informations concernant le salarié figurent dans le message d'erreur affiché lors du traitement de l'exception. **throw** provoque un branchement inconditionnel dans le bloc **catch** du programme appelant ou cette classe d'exception est capturée et traitée. Le reste de la fonction n'est donc pas exécuté.

6 Si tout ce passe bien, la propriété **m_dSalaire** est modifiée.

7 Le constructeur d'initialisation est susceptible de générer des exceptions de classes **CategorieSalarieException** et **SalaireSalarieException**. Le programme appelant doit donc en être informé par l'instruction **throws**.

8 Les données membres de l'objet sont initialisées avec les paramètres passés au constructeur.

9 Si tout se passe bien, le compteur d'instances est incrémenté.

10 Une nouvelle classe d'exception **SalarieException** est déclarée en dérivant la classe **Salarie**. Cette classe est générique pour toutes les classes d'exception générées dans le fonctionnement de la classe **Salarie**.

11 Le constructeur de la classe **SalarieException** reçoit une instance de **Salarie** comme paramètre. Ce paramètre est transmis au constructeur de la classe de base. Le constructeur de la classe **Exception** ne peut recevoir qu'un paramètre de type **String**. On utilise la méthode **toString** de la classe **Salarie** pour la conversion. Cela permet d'intégrer dans le message d'erreur associé à l'exception, les propriétés du salarié en cause dont un paramètre est incorrect.

12 Une nouvelle classe d'exception **CategorieSalarieException** est déclarée en dérivant la classe **SalarieException**. Cette classe est invoquée quand on cherche à modifier ou à initialiser la propriété **m_nCategorie** de la classe **Salarie**.

13 Le constructeur de la classe **CategorieSalarieException** reçoit une instance de **Salarie** comme paramètre. Ce paramètre est transmis tel quel au constructeur de la classe de base **SalarieException**.

14 La méthode **toString** de la classe **CategorieSalarieException** permet de modifier le message d'erreur pour le personnaliser par rapport à la classe d'exception.

15 Une nouvelle classe d'exception **SalaireSalarieException** est déclarée en dérivant la classe **SalarieException**. Cette classe est invoquée quand on cherche à modifier ou à initialiser la propriété **m_dSalaire** de la classe **Salarie**.

16 Le constructeur de la classe **SalaireSalarieException** reçoit une instance de **Salarie** comme paramètre. Ce paramètre est transmis tel quel au constructeur de la classe de base **SalarieException**.

17 La méthode **toString** de la classe **SalaireSalarieException** permet de modifier le message d'erreur pour le personnaliser par rapport à la classe d'exception.

monApp.java

```
import java.util.*;

public class monApp
{
     public static void main (String[] args)
     {
          Hashtable dict = new Hashtable();
          Salarie s;
          try
          {
1              s = new Salarie(16, 1, 10, "CAUJOL", 10900.00);
               putInDictionary(dict, s);
          }
          catch (SalarieException se)
          {
2              System.err.println(se);
          }
          try
          {
1              s = new Salarie(5, 1, 10, "DUMOULIN", 15600.00);
               putInDictionary(dict, s);
          }
          catch (SalarieException se)
          {
2              System.err.println(se);
          }
          try
          {
1              s = new Salarie(29, 3, 20, "AMBERT", 5800.00);
               putInDictionary(dict, s);
          }
          catch (SalarieException se)
          {
2              System.err.println(se);
          }
          try
          {
1              s = new Commercial(17, 2, 20, "ABRU", 6000.00,
                                  120000.0,7);
               putInDictionary(dict, s);
          }
          catch (SalarieException se)
          {
2              System.err.println(se);
          }
          try
          {
3              s = new Salarie(20, 5, 20, "CITEAUX", 8000.00);
               putInDictionary(dict, s);
          }
          catch (SalarieException se)
          {
2              System.err.println(se);
          }
          try
          {
1              s = new Salarie(34, 2, 30, "CHARTIER", 7800.00);
               putInDictionary(dict, s);
          }
          catch (SalarieException se)
          {
2              System.err.println(se);
          }
          try
```

```
                {
4                    s = new Commercial(51, 2, 20, "BRUTI", -7500.00,
                                        150000.0, 11);
                     putInDictionary(dict, s);
                }
                catch (SalarieException se)
                {
2                    System.err.println(se);
                }


5               for (Enumeration e = dict.elements() ; e.hasMoreElements() ;)
                {
                     Salarie sal = (Salarie)e.nextElement();
                     System.out.println(sal);
                     System.out.println(sal.calculSalaire());
                }
        }

        public static void putInDictionary(Hashtable dict, Salarie s)
        {
             Integer key = new Integer(s.matricule());
             dict.put(key, s);
        }
}
```

Commentaires

1 En ce qui concerne la surveillance de l'instanciation des **Salarie**, le choix a été fait de surveiller à part, dans un bloc **try-catch**, l'instanciation de chaque **Salarie**. De ce fait l'application pourra continuer normalement et seuls seront pris en compte les salariés dont les propriétés sont correctes. On ne cherche pas à distinguer le type spécifique d'exception. On se contente de capturer les exceptions de classe générique **SalarieException**.

2 En cas de problème dans l'instanciation des **Salarie**, un branchement inconditionnel est effectué dans le bloc **catch**. La fonction **putInDictionary** n'est pas exécutée dans ce cas là.

3 Cette instanciation va déclencher une exception de classe **CategorieSalarieException** car le paramètre correspondant à la catégorie est égal à 5, donc différent de 1, 2 ou 3 qui sont les valeurs permises de cette propriété.

4 Cette instanciation va déclencher une exception de classe **SalaireSalarieException** car le paramètre correspondant au salaire est négatif.

5 Lors de l'itération du dictionnaire, seuls les salariés dont les propriétés étaient correctes seront affichés.

Résultat affiché

```
CategorieSalarieException: 20,5,20,CITEAUX,8000.0
La categorie ne peut etre que 1, 2 ou 3.

SalaireSalarieException: 51,2,20,BRUTI,-7500.0,0,0.0
Le salaire ne peut etre que positif.
```

```
29,3,20,AMBERT,5800.0
Le salaire de AMBERT est de 5800.0.
17,2,20,ABRU,6000.0,7,120000.0
Le salaire de ABRU est de 14400.0.
5,1,10,DUMOULIN,15600.0
Le salaire de DUMOULIN est de 15600.0.
16,1,10,CAUJOL,10900.0
Le salaire de CAUJOL est de 10900.0.
34,2,30,CHARTIER,7800.0
Le salaire de CHARTIER est de 7800.0.
```

*Ces trois fichiers se trouvent dans le répertoire **JOTP7**.*