IN201 : 5 – Héritage

Author : Christophe Garion <garion@isae.fr>
Public : SUPAERO 2A
Date :

Résumé

Ce TP a pour but de vous faire comprendre les mécanismes liés à l'héritage, comme le polymorphisme et la liaison tardive.

1 Objectifs

Les objectifs du TP sont les suivants :
- comprendre l'héritage ;
- concevoir et implanter une classe spécialisant une classe existante ;
- comprendre le polymorphisme et la liaison dynamique.

Pour réaliser le TP, vous allez devoir créer un projet Java sous Eclipse en utilisant votre dépôt Subversion. Si vous avez configuré votre dépôt pour qu'il soit disponible dans la vue *SVN Repository Exploring* d'Eclipse, vous créerez votre projet en faisant un *checkout* **à partir du répertoire TP5** de votre dépôt en cliquant droit dessus. N'oubliez pas de configurer votre *build path* pour pouvoir utiliser les éventuelles archives JAR placées dans le répertoire `lib`.

2 Utilisation de la classe `PointNomme`

Récupérer la classe `PointNomme` et son programme de test `TestPolymorphisme` sur le site. Le programme de test comporte des erreurs. Répondre aux questions contenues dans le source de `TestPolymorphisme`* (cf. listing 1).

N'utilisez pas Eclipse dans un premier temps et essayez de répondre aux questions sur le listing fourni. Vous pourrez ensuite vérifier vos réponses en utilisant Eclipse.

Listing 1– La classe `TestPolymorphisme`

```
 1 package fr.isae.geometry;
 2
 3 /**
 4  * Tester le polymorphisme et la liaison dynamique.
 5  *
 6  * @author <a href="mailto:cregut@enseeiht.fr">Xavier Cregut</a>
 7  * @version 1.0
 8  */
 9 public class TestPolymorphisme {
10
11     /** Methode principale */
12     public static void main(String args[]) {
13         // Creer et afficher un point p1
14         Point p1 = new Point(3, 4);             // Est-ce autorise ? Pourquoi ?
15         p1.translater(10,10);   // Quel est le translater qui est execute ?
16         System.out.print("p1 = ");      p1.afficher (); System.out.println ();
17                         // Qu'est ce qui est affiche ?
18
19         // Creer et afficher un point nomme pn1
20         PointNomme pn1 = new PointNomme (30, 40, "PN1");
21                         // Est-ce autorise ? Pourquoi ?
```

```
22          pn1.translater (10,10); // Quel est le translater qui est execute ?
23          System.out.print ("pn1 = ");    pn1.afficher(); System.out.println ();
24                          // Qu'est ce qui est affiche ?
25
26          // Definir une poignee sur un point
27          Point q;
28
29          // Attacher un point a q et l'afficher
30          q = p1;         // Est-ce autorise ? Pourquoi ?
31          System.out.println ("> q = p1;");
32          System.out.print ("q = ");      q.afficher();   System.out.println ();
33                          // Qu'est ce qui est affiche ?
34
35          // Attacher un point nomme a q et l'afficher
36          q = pn1;                // Est-ce autorise ? Pourquoi ?
37          System.out.println ("> q = pn1;");
38          System.out.print ("q = ");      q.afficher();   System.out.println ();
39                          // Qu'est ce qui est affiche ?
40
41          // Definir une poignee sur un point nomme
42          PointNomme qn;
43
44          // Attacher un point a q et l'afficher
45          qn = p1;        // Est-ce autorise ? Pourquoi ?
46          System.out.println ("> qn = p1;");
47          System.out.print ("qn = ");     qn.afficher();  System.out.println ();
48                          // Qu'est ce qui est affiche ?
49
50          // Attacher un point nomme a qn et l'afficher
51          qn = pn1;       // Est-ce autorise ? Pourquoi ?
52          System.out.println ("> qn = pn1;");
53          System.out.print ("qn = ");     qn.afficher();  System.out.println ();
54                          // Qu'est ce qui est affiche ?
55
56          double d1 = p1.distance (pn1);  // Est-ce autorise ? Pourquoi ?
57          System.out.println ("distance = " + d1);
58
59          double d2 = pn1.distance (p1);  // Est-ce autorise ? Pourquoi ?
60          System.out.println ("distance = " + d2);
61
62          double d3 = pn1.distance (pn1); // Est-ce autorise ? Pourquoi ?
63          System.out.println ("distance = " + d3);
64
65          System.out.println ("> qn = q;");
66          qn = q; // Est-ce autorise ? Pourquoi ?
67          System.out.print ("qn = ");     qn.afficher();  System.out.println ();
68
69          System.out.println ("> qn = (PointNomme) q;");
70          qn = (PointNomme) q;    // Est-ce autorise ? Pourquoi ?
71          System.out.print ("qn = ");     qn.afficher();  System.out.println ();
72
73          System.out.println ("> qn = (PointNomme) p1;");
```

```
74          qn = (PointNomme) p1;   // Est-ce autorise ? Pourquoi ?
75          System.out.print ("qn = ");     qn.afficher();  System.out.println ();
76      }
77  }
```

3 De la méthode equals de Point

Vous trouverez dans votre dépôt les codes sources des classes `Point` et `PointNomme` appartenant au paquetage `fr.isae.geometry`. Le code source de la classe `Point` est celui qui était fourni lors du TP sur les orbites constituées de `Point`. En particulier, la méthode `equals` développée dans cette classe n'était pas « la bonne » (cf. cours sur l'héritage et exercice sur `equals` et JUnit).

1. modifier la signature de la méthode `equals` pour qu'elle redéfinisse la méthode `equals` de `Object`.
2. implanter la méthode `equals` de `Point`. Vérifier dans un test JUnit simple que la méthode fonctionne bien dans tous les cas.
3. redéfinir également la méthode `equals` de `PointNomme`. Peut-on utiliser la méthode `equals` de `Point` dans l'implantation de cette méthode ?
4. créer un test JUnit simple qui utilise une instance `p` de `Point` et une instance `pn` de `PointNomme` ayant des coordonnées identiques et utiliser l'assertion qui semble correcte parmi `assertTrue` et `assertFalse` pour les appels suivants :
   - `p.equals(pn)`
   - `pn.equals(p)`

   Que peut-on dire du résultat ?

4 Gestion de comptes bancaires

On désire modéliser un système bancaire comportant des comptes simples, des comptes avec historique, des comptes rémunérés. Pour cela, on dispose de classes déjà développées que l'on va utiliser.

4.1 Les classes CompteSimple et Personne

Dans un premier temps, on va utiliser des classes déjà développées :
- la classe `CompteSimple` qui représente un compte « basique » ;
- la classe `Personne` qui représente une personne physique ;
- la classe `Historique` permettant de représenter un historique pour un compte.

Les *bytecodes* de ces classes sont disponibles sur le site, ainsi que leur documentation javadoc. Le diagramme de classe présenté sur la figure 1 vous présente les relations existant entre ces deux classes, ainsi que la classe `Historique` de façon simplifiée[1]. Vous remarquerez que la classe `Historique` réalise `Iterable` (cf. corrigé du TP sur les interfaces), on peut donc parcourir ici les intitulés des opérations enregistrées dans l'historique avec une boucle **for**. Il faudra se référer à la documentation `javadoc` fournie pour avoir l'ensemble des méthodes applicables sur les objets des classes (en particulier les accesseurs et modifieurs induits par les associations entre les classes).
Une classe de test de `CompteSimple`, `CompteSimpleTest`, est disponible sur le site.

1. En particulier, des méthodes permettant de mettre en forme plus facilement les relevés de compte dans `CompteCourant` sont disponibles dans `Historique`

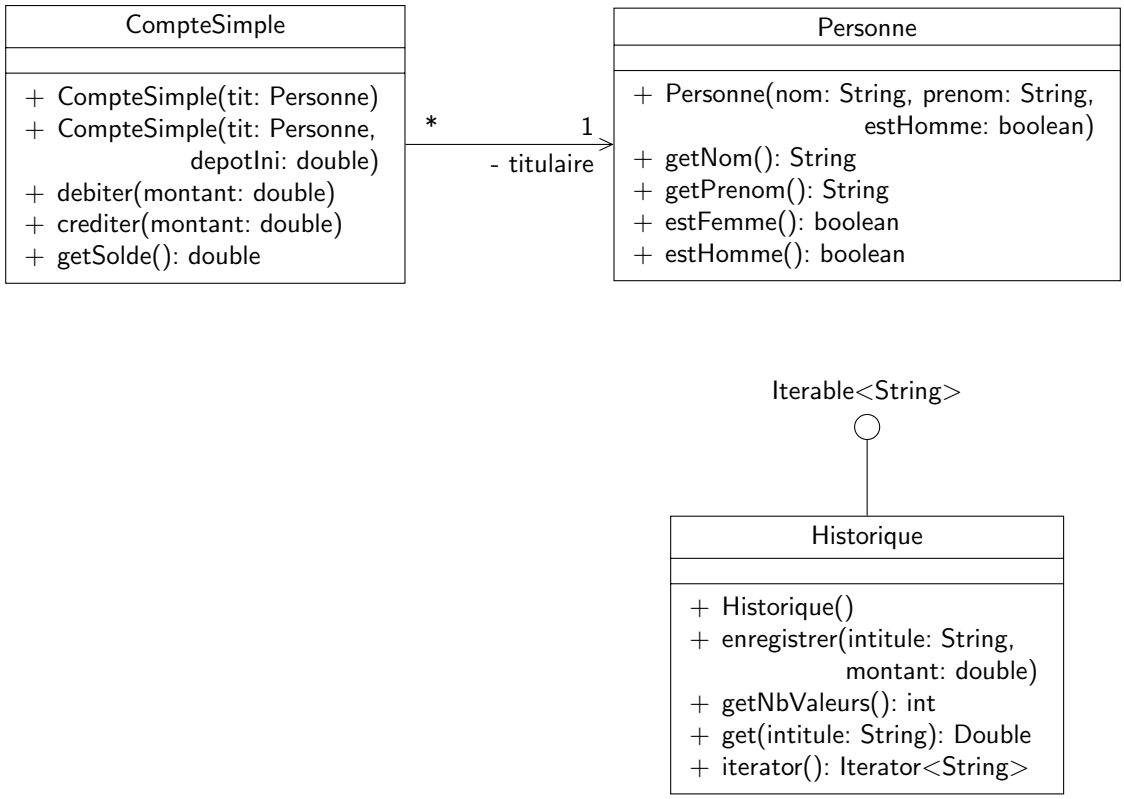

FIGURE 1 – Diagramme de classe présentant `CompteSimple`, `Personne` et `Historique`

4.2 Conception et implantation de la classe `CompteCourant`

Une banque conserve pour chaque compte l'historique des opérations qui le concernent (on ne s'intéresse ici qu'aux débits et aux crédits). On souhaite modéliser un tel compte qu'on appelle *compte courant*. En plus des opérations d'un compte simple, un compte courant offre des opérations pour afficher l'ensemble des opérations effectuées (`editerReleve`), ou seulement les opérations de crédit (`afficherReleveCredits`) ou de débit (`afficherReleveDebits`) et permet de créditer ou de débiter le compte en ajoutant un intitulé à ces opérations.
Pour représenter l'historique, on utilisera la classe `Historique` fournie. Pour enregistrer une opération, on conservera le signe de l'opération (crédit ou débit) dans l'historique.

1. compléter le diagramme ci-dessus pour inclure une classe `CompteCourant` qui possède un historique. On identifiera les méthodes *redéfinissant* des méthodes de `CompteSimple` et les méthodes *surchargeant* des méthodes de `CompteSimple`.
2. implanter la classe `CompteCourant`.
3. un programme de test vous est fourni sur le site. L'exécuter et commenter les résultats.
4. est-ce que ce programme est suffisant ? Proposer une classe de tests JUnit `CompteCourantTest`. Cette classe de test doit-elle utiliser la classe `CompteSimpleTest` ? Si oui, pourquoi ? On pourra modifier éventuellement `CompteSimpleTest`.

Pour qu'Eclipse prenne en compte les codes sources disponibles dans le répertoires `tests`, il faut ajouter ce répertoire parmi les sources de votre projet via le menu « Configure Build Path » onglet « Sources ».

5. écrire un programme de test construisant une liste d'instances de `CompteSimple` et afficher les relevés des instances de `CompteCourant` figurant dans le tableau.

5 Et maintenant les LDD...

On souhaiterait spécialiser la classe `CompteSimple` en une classe `LDD`. On rappelle qu'un Livret de Développement Durable (LDD) est un compte rémunéré dont le solde est plafonné à 12000€ (un versement ne peut avoir pour conséquence de

porter le montant inscrit sur le compte au dela de 12000€).
Cette spécialisation est-elle judicieuse ? Pourquoi ?

License CC BY-NC-SA 3.0