IN201 : 7 – Héritage

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

Solution

Résumé

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

1 Contenu

Ce corrigé contient des commentaires aux questions du TP sur l'héritage. Vous pourrez trouver les sources des classes et interfaces sur le site web.

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`.

Solution :
Voici les commentaires que l'on pouvait faire sur le programme de test :

```
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 ?
```

Ici, pas de problème particulier. On crée une référence de type `Point` et on lui affecte un objet de type `Point`. Toutes les méthodes utilisées sont celles de `Point` (car l'objet référencé par `p1` est de type `Point`).

```
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 ?
```

De la même façon, il n'y aucun problème ici. Les méthodes utilisées sont celles de la classe `PointNomme`.

```
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 ?
```

On définit une référence de type `Point`. On affecte ensuite à cette référence une autre référence de type `Point`. Il n'y a donc aucun problème au niveau du typage. Les méthodes utilisées sont celles de la classe de l'objet référencé par `p1`, i.e. `Point`.

```
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 ?
```

q est une référence de type Point. pn1 est une référence de type PointNomme. D'après le principe de substitution, toute instance d'une sous-classe peut « remplacer » une instance d'une de ses super-classes. Ce principe s'applique également aux types des références. Comme PointNomme hérite de Point, on peut affecter une référence de type PointNomme à q. Le principe de liaison dynamique nous assure que ce n'est pas le type de la référence qui va déterminer l'implantation de la méthode qui sera choisie, mais le type de l'objet référencé. Donc même si q est de type Point, comme l'objet référencé est de type PointNomme, les méthodes appliquées vont être celles de la classe PointNomme.

```
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 ?
```

Java est un langage fortement typé. Toute variable ou référence possède un type et le compilateur vérifie statiquement qu'il n'y a pas d'incompatibilité entre les types. En particulier, dans le cas d'une affectation à une référence, il vérifie que l'objet ou la référence que l'on affecte est bien du type ou d'un sous-type de la référence. Or ici, qn est de type PointNomme et p1 est de type Point. Comme Point n'est pas une sous-classe de PointNomme, on ne peut pas affecter p1 à qn. Le compilateur détecte cette erreur et émet le message suivant :

```
TestPolymorphisme.java:45: incompatible types
found   : fr.supaero.figure.Point
required: fr.supaero.figure.PointNomme
        qn = p1;        // Est-ce autorise ? Pourquoi ?
             ^
```

```
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 ?
```

Il n'y a aucun problème. On utilise deux références de même type et l'objet référencé est également du même type. Ce sont donc les méthodes de PointNomme qui vont être utilisées.

```
56          double d1 = p1.distance (pn1);  // Est-ce autorise ? Pourquoi ?
```

On utilise la méthode distance sur une référence de type Point (classe qui implante la méthode distance). Il n'y a donc pas d'erreur à la compilation (de plus, comme l'objet référencé par p1 est également de type Point, il n'y aura pas d'erreur à l'exécution. On suppose donc maintenant que les références ont été correctement initialisées, en particulier qu'elles ne valent pas **null**). La référence passée en paramètre de distance est de type PointNomme. Or, le type du paramètre de distance est Point. Mais PointNomme hérite de Point, donc d'après le principe de substitution, on peut utiliser un PointNomme à la place d'un Point.

```
59          double d2 = pn1.distance (p1);  // Est-ce autorise ? Pourquoi ?
```

De la même façon, d'après le principe de polymorphisme, un PointNomme se comporte comme un Point. On peut donc appeler la méthode distance sur une référence de type PointNomme.

```
62          double d3 = pn1.distance (pn1); // Est-ce autorise ? Pourquoi ?
```

C'est une « combinaison » des deux cas précédents.

```
66          qn = q; // Est-ce autorise ? Pourquoi ?
67          System.out.print ("qn = ");     qn.afficher();  System.out.println ();
```

On affecte une référence de type `Point` à une référence de type `PointNomme`. Comme `Point` n'hérite pas de `PointNomme`, il y a une erreur à la compilation :

```
TestPolymorphisme.java:66: incompatible types
found   : fr.supaero.figure.Point
required: fr.supaero.figure.PointNomme
        qn = q; // Est-ce autorise ? Pourquoi ?
             ^
```

```
70          qn = (PointNomme) q;    // Est-ce autorise ? Pourquoi ?
71          System.out.print ("qn = ");     qn.afficher();  System.out.println ();
```

`qn` est une référence de type `PointNomme` et `q` est une référence de type `Point`. On ne peut pas affecter directement `q` à `qn`. On utilise le transtypage (grâce à `(PointNomme)`) pour « transformer » le type de `q`. Il n'y aura donc pas d'erreur à la compilation.
Dynamiquement, i.e. à l'exécution, il faut que le transtypage soit effectivement possible. Comme l'objet référencé par `q` est effectivement un objet de type `PointNomme` (c'est l'objet référencé par `pn1`), le transtypage ne posera pas de problème.

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

Le problème est presque identique au précédent. Le transtypage nous permet de compiler sans erreur. Par contre, à l'exécution, il va y avoir un problème. On veut transtyper `p1` qui est une référence de type `Point` et qui référence un objet de type `Point`. Comme `Point` n'est pas une sous-classe de `PointNomme`, on ne peut pas transtyper « effectivement » `p1`. On va donc avoir une exception de type `ClassCastException` à l'exécution :

```
Exception in thread "main" java.lang.ClassCastException:
 fr.supaero.figure.Point cannot be cast to fr.supaero.figure.PointNomme
        at fr.supaero.figure.TestPolymorphisme.main(TestPolymorphisme.java:75)
```

3 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.

3.1 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. 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.

3.2 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` ;

   Solution :

   La classe `CompteCourant` hérite de la classe `CompteSimple`. Le diagramme de classes complet est présenté sur la figure 1.

   On peut remarquer que la relation qui existe entre `CompteCourant` et `Historique` est une relation de composition : on considère en effet qu'un historique n'est attaché qu'à un seul compte courant et disparaît avec lui.

   Il est intéressant de noter que des méthodes portant le même nom dans `CompteCourant` sont des méthodes redéfinissant ou surchargeant des méthodes de `CompteSimple` :

   – les méthodes `crediter(montant: double)` et `debiter(montant: double)` *redéfinissent* les méthodes de même signature présentes dans `CompteSimple`. Elles définissent un nouveau comportement pour ces méthodes.
   – les méthodes `crediter(intitule: String, montant: double)` et `debiter(intitule: String, montant: double)` *surchargent* les méthodes `crediter` et `debiter` présentées précédemment. Ce sont des méthodes différentes (elles n'ont pas la même signature) ! Vous remarquerez dans le code source de `CompteCourant` que je n'ai utilisé le *tag* `@Override` que pour les méthodes précédentes.

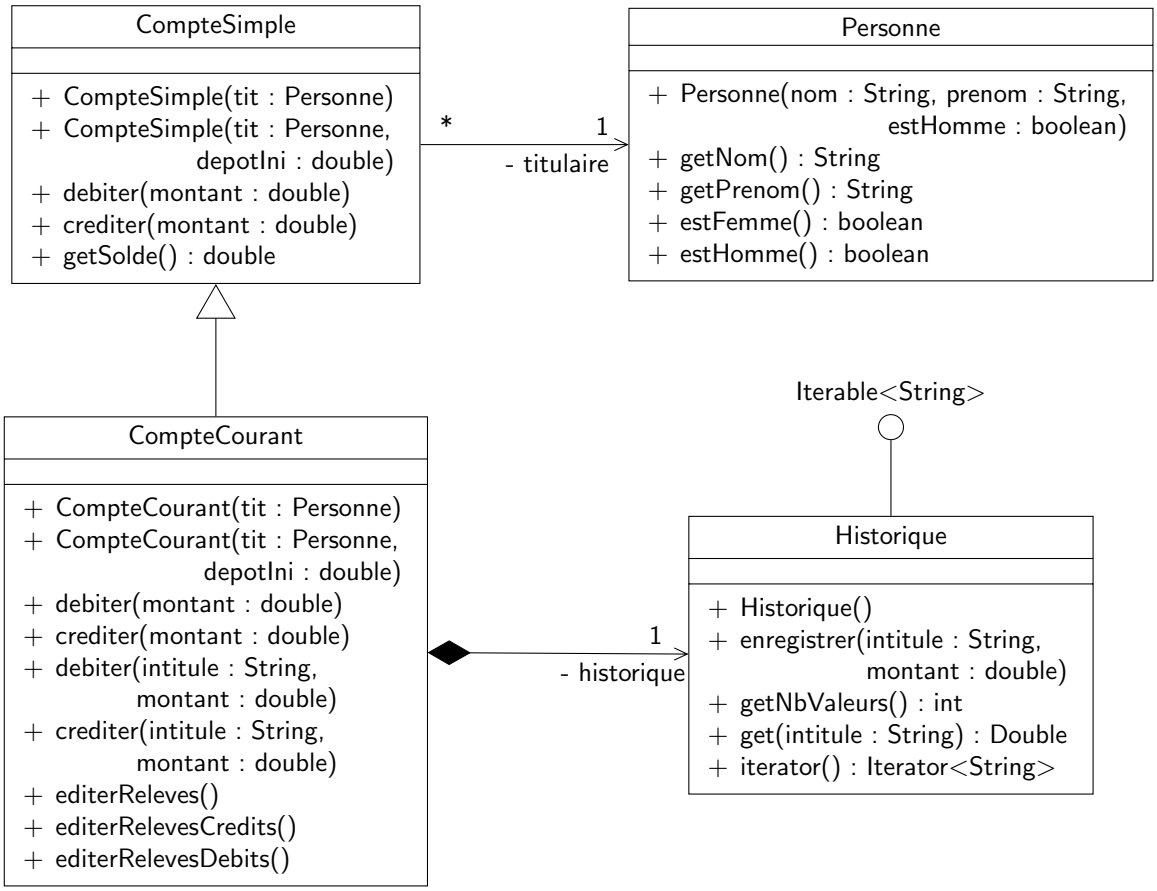

FIGURE 1 – Diagramme de classe présentant `CompteSimple`, `Personne`, `CompteCourant` et `Historique`

2. implanter la classe `CompteCourant` ;

Solution :

Le code source est disponible sur le site. Il n'y avait pas de problèmes particuliers. Vous remarquerez simplement qu'une précondition du constructeur de `CompteCourant` est que le solde passé en paramètre soit positif ou nul.

De plus, j'ai empêché la redéfinition de la méthode `crediter` utilisée dans le constructeur. Si on spécialise `CompteCourant` en une autre classe qui redéfinit elle-même `crediter`, il peut y avoir des problèmes (utilisation d'un attribut non correctement initialisé par exemple).

Je vous propose quelques approfondissements dans la section 5 sur le code que j'ai écrit :

– la construction de la classe `Historique` et l'utilisation d'une *table de correspondance* ;
– la mise en forme des relevés et l'utilisation de la méthode `printf`.

3. un programme de test vous est fourni sur le site. L'exécuter et commenter les résultats ;

Solution :

L'exécution de `ExempleComptes` donne le résultat suivant :

```
Solde de cs1 = 1000.0
Solde de cc1 = 1100.0
-------------------------------------
Titulaire :      M. Christophe Garion
+-----------------------------------+
|               |  credit |   debit |
|---------------+---------+---------|
| depot initial |  100,00 |         |
| operation 1   | 1000,00 |         |
|---------------+---------+---------|
|         solde | 1100,00 |         |
+-----------------------------------+

Solde de cs = 600.0
Solde de cc1 = 600.0
-------------------------------------
Titulaire :      M. Christophe Garion
+-----------------------------------+
|               |  credit |   debit |
|---------------+---------+---------|
| depot initial |  100,00 |         |
| operation 1   | 1000,00 |         |
| operation 2   |         |  500,00 |
|---------------+---------+---------|
|         solde |  600,00 |         |
+-----------------------------------+
```

C'est bien le résultat attendu grâce à la liaison dynamique et la redéfinition des méthodes `crediter` et `debiter`. En particulier :

– `cc1.crediter(1000)` : on exécute bien la méthode de la classe `CompteCourant`. Si on n'avait pas redéfini la méthode, l'historique n'aurait pas été mis à jour et on l'aurait vu à l'appel de `cc1.editerReleve()` ;
– `cs.debiter(500)` : le compilateur sélectionne la méthode `debiter` de la classe `CompteSimple` car il se base sur le type de la poignée `cs`. Par contre, à l'exécution, c'est bien la méthode de la classe `CompteCourant` qui est exécutée, car le type de l'objet attaché à `cs` est bien `CompteCourant` (liaison dynamique).

4. est-ce que ce programme est suffisant ? Proposer des tests supplémentaires si nécessaire ;

Solution :

Il faut vérifier que le sous-typage fonctionne correctement. On peut donc utiliser la classe de test de `CompteSimple` avec un objet de type `CompteCourant`. Supposons que l'on ait la classe de test pour `CompteSimple` présentée sur le listing 1 [1] :

Listing 1– La classe `CompteSimpleTest`

```java
package fr.supaero.banque;

import org.junit.*;
import static org.junit.Assert.*;

/**
 *  Unit Test for class CompteSimple
 *
 *
 * Created: Wed Nov 30 22:42:37 2005
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
 * @version 1.0
 */
public class CompteSimpleTest {

    private CompteSimple c;

    protected CompteSimple createCompte(Personne p, double solde) {
        return new CompteSimple(p, solde);
    }

    @Before public void setUp() {
        this.c = this.createCompte(new Personne ("Christophe", "Garion", true),
                                   1000);
    }

    @Test public void testGetSolde() {
        Assert.assertEquals(1000, this.c.getSolde(), 0.0);
    }

    @Test public void testCrediter() {
        this.c.crediter(100);
        Assert.assertEquals(1100, this.c.getSolde(), 0.0);
    }

    @Test public void testDebiter() {
        this.c.debiter(300);
        Assert.assertEquals(700, this.c.getSolde(), 0.0);
    }
}// CompteSimpleTest
```

On pourra écrire une classe de test `CompteCourantTest` pour `CompteCourant` comme présenté sur le listing 2.

Listing 2– La classe `CompteCourantTest`

```java
package fr.supaero.banque;

import org.junit.*;
import static org.junit.Assert.*;

/**
 *  Unit Test for class CompteCourant
 *
 *
 * Created: Wed Nov 30 22:28:22 2005
 *
 * @author <a href="mailto:garion@supaero.fr">Christophe Garion</a>
```

```
 * @version 1.0
 */
public class CompteCourantTest extends CompteSimpleTest {

    private CompteCourant cc;
    private Historique hist;
    private String[] tabC = { "depot cheque 1", "depot cheque 2", "depot cheque 3" };
    private double[] tabMC = { 100, 200, 300 };
    private String[] tabD = { "retrait 1", "retrait 2", "retrait 3" };
    private double[] tabMD = { 20, 20, 40 };

    protected CompteSimple createCompte(Personne p, double solde) {
        return new CompteCourant(p, solde);
    }

    @Before public void setUp() {
        super.setUp();
        this.cc = new CompteCourant(new Personne("Christophe", "Garion", true),
                                    1000);
        this.hist = this.cc.getHistorique();
    }

    @Test public void testCrediterWithIntitule() {
        this.cc.crediter("depot cheque", 100);
        Assert.assertEquals(1100, this.cc.getSolde(), 0.0);
    }

    @Test public void testDebiterWithIntitule() {
        this.cc.debiter("retrait", 300);
        Assert.assertEquals(700, this.cc.getSolde(), 0.0);
    }

    @Test public void testHistoriqueCreation() {
        for (String intitule : this.hist) {
          Assert.assertEquals("depot initial", intitule);
            Assert.assertEquals(1000, this.hist.get(intitule), 0.0);
        }
    }

    @Test public void testHistoriqueCrediter() {
        for (int i = 0; i < this.tabC.length; i++) {
            this.cc.crediter(tabC[i], tabMC[i]);
        }

        int i = 0;
        for (String intitule : this.hist) {
            if (!intitule.equals("depot initial")) {
                Assert.assertEquals(tabC[i], intitule);
                Assert.assertEquals(tabMC[i], this.hist.get(intitule), 0.0);
                i++;
            }
        }
    }

    @Test public void testHistoriqueDebiter() {
        for (int i = 0; i < this.tabD.length; i++) {
            this.cc.crediter(tabD[i], tabMD[i]);
        }
```

```
        int i = 0;
        for (String intitule : this.hist) {
            if (!intitule.equals("depot initial")) {
                Assert.assertEquals(tabD[i], intitule);
                Assert.assertEquals(tabMD[i], this.hist.get(intitule), 0.0);
                i++;
            }
        }
    }
}// CompteCourantTest
```

Cette classe de test donne bien le résultat suivant :

```
Testsuite: fr.supaero.banque.CompteCourantTest
Tests run: 8, Failures: 0, Errors: 0, Time elapsed: 0,228 sec
```

On voit que les méthodes de test définies dans `CompteSimpleTest` sont bien exécutées.

La méthode `createCompte` dans `CompteCourant` est ce que l'on appelle une méthode *factory* : elle permet de construire un objet en encapsulant un appel au constructeur de la classe. Comme elle est protégée, on la redéfinit dans `CompteCourantTest` pour qu'elle renvoie un objet de type `CompteCourant`. On peut alors « utiliser » les méthodes de test publiques de `CompteSimpteTest`. On aurait également pu déclarer l'attribut `c` de `CompteSimpteTest` comme étant protégé et modifier la méthode `setUp` de `CompteCourantTest`. Le fait de spécialiser `CompteSimpleTest` et de redéfinir `createCompte` suffit à faire exécuter tous les tests définis dans `CompteSimpleTest` avec une instance de `CompteCourant`.

Aurait-on pu masquer l'attribut `c` de `CompteSimpteTest` avec un autre attribut `c` de type `CompteCourant` dans `CompteCourantTest` ? Non, car le masquage ne « remplace » pas l'attribut ! Lors de l'appel aux méthodes de test de `CompteSimpteTest` « depuis » `CompteCourantTest`, on aurait utilisé l'attribut `c` de `CompteSimpteTest` qui n'est pas initialisé.

J'avais besoin de créer des méthodes de test spécifiques à la classe `CompteCourant` pour vérifier que les méthodes `crediter` et `debiter` prenant un intitulé en paramètre fonctionnait bien. J'ai donc ajouté un attribut de type `CompteCourant`, défini une méthode `setUp` qui fait appel à la méthode `setUp` de `CompteSimpleTest` et écrit les méthodes de test. On remarquera que l'attribut de type `CompteCourant` déclaré dans `CompteCourantTest` ne sert pas pour l'exécution des tests définis dans `CompteSimpleTest`.

Reste un problème : je ne vérifie pas ici que les informations sont correctement enregistrées dans l'historique. En effet, il n'y a pas de méthode publique permettant de récupérer cet historique. J'ai donc créé cette méthode pour pouvoir tester l'inscription dans l'historique. Je lui ai donné une visibilité de paquetage pour n'autoriser que les classes du paquetage `fr.supaero.banque` (dont fait partie `CompteCourantTest`) à l'utiliser. Remarquez que la création de cette méthode d'accès à l'historique ne respecte pas le diagramme de conception présenté sur la figure 1 : l'historique ne doit pas être accessible depuis l'extérieur de la classe `CompteCourant` (nom de rôle – `historique`). On aurait également pu utiliser les mécanismes de réflexion (cf. corrigé du TP récapitulatif) pour récupérer l'historique associé au compte sans cette méthode.

Enfin, on remarquera que le même principe a été appliqué pour les classes de test des classes `Point` et `PointNomme`.

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

   Solution :

   Le source est disponible sur le site. J'ai choisi d'utiliser une instance de la classe `ArrayList<CompteSimple>` plutôt qu'un tableau, mais le principe est le même. Il fallait penser à utiliser **`instanceof`** pour tester le type réel des objets attachés aux références de la liste (qui sont **toutes** des références de type `CompteSimple`). Il fallait également transtyper les références contenues dans la liste.

4 Conception d'une classe `Carre`

On souhaiterait spécialiser la classe `Polygone` développée précédemment en une classe `Carre`. Le constructeur de la classe `Carre` prendrait en paramètre le point représentant le coin inférieur gauche du carré et la valeur de son côté.
Cette spécialisation est-elle judicieuse ? Pourquoi ?

Solution :
Il paraît naturel de spécialiser la classe `Polygone` si on veut créer une classe `Carre`. Il faut dans un premier temps réfléchir aux conséquences de cette spécialisation.
En effet, si `Carre` est une spécialisation de `Polygone`, d'après le principe de substitution, toute instance de `Carre` peut remplacer une instance de `Polygone`. En particulier, on doit pouvoir appeler la méthode `ajouter` sur une instance de la classe `Carre` et cette méthode doit fournir au moins le même comportement que celle de la classe `Polygone`. Or, il est impossible d'ajouter un nouveau point à un carré.
De plus, la méthode `retirer` doit pouvoir également s'appliquer à un objet de type `Carre` et celui-ci doit fournir au moins le même comportement qu'une instance de `Polygone`. Or, il est intuitivement incorrect de retirer un point à un carré et rien ne nous en empêche.
On ne peut donc pas spécialiser la classe `Polygone` en une classe `Carre`.
Cette question était bien sûr une question « piège ». On peut évidemment construire en Java une classe `Carre` qui redéfinit la méthode `retirer` de telle façon que celle-ci ne retire pas le point passé en paramètre. Mais dans ce cas, *du point de vue conception*, on viole le principe de substitution.
On peut se demander également « où » est l'erreur. À mon avis, un carré est un polygone, il n'y a pas de doute... Donc c'est la classe `Polygone` qui est mal conçue. On ne devrait pas pouvoir ajouter ou retirer un point à un polygone.

5 Pour aller plus loin...

Je vais revenir dans cette section sur quelques aspects avancés de l'implantation des classes `Historique` et `CompteCourant`.

5.1 Tableaux associatifs dans `Historique`

La classe `Historique` devait servir à stocker les opérations effectuées sur un compte. Plutôt que de stocker simplement le montant des opérations, je souhaitais avoir également le descriptif de l'opération via un intitulé. On aurait donc très bien pu stocker
– les intitulés sous forme d'une instance de `ArrayList<String>`
– les montants sous forme d'une instance de `ArrayList<Double>`
Les listes étant ordonnées, on aurait eu avec le même indice l'intitulé de l'opération dans une des listes et le montant associé dans l'autre. J'ai choisi ici d'utiliser un *tableau associatif*[2] [3]. Un tableau associatif permet d'associer à un ensemble de *clés* un ensemble de *valeurs* via une fonction *injective*. On peut voir les tableaux associatifs comme une généralisation des tableaux « classiques » : pour ces derniers, l'ensemble de clés est un sous-ensemble de $\mathbb{N}$ particulier. Les tableaux associatifs sont des structures de données très importantes en informatique, ils permettent de représenter des dictionnaires, des bases de données etc.
Une implantation possible d'un tableau associatif se fait via l'utilisation d'une *table de hachage* (*hashtable* en anglais). Une table de hachage est un tableau associatif dans lequel les clés sont transformées en des valeurs entières via une fonction de *hachage*. Cette fonction doit garantir que deux clés différentes produiront deux valeurs différentes. L'intérêt des tables de hachage est que la complexité moyenne en temps des accès à un élément de la table se fait en $O(1)$.
En Java, les tables de hachage sont représentées par la classe `java.util.HashMap<K,V>`. Comme `ArrayList`, cette classe est *générique* (nous consacrerons un cours à la généricité). Pour `HashMap`,`K` représente le type des clés et `V` le type des valeurs. Ici, je veux associer aux intitulés des opérations leurs montants, donc je vais utiliser une instance de `HashMap<String, Double>` comme attribut de `Historique`.
`HashMap` fournit des méthodes de base disponible pour tous les tableaux associatifs en Java :
– `put(K cle, V valeur)` permet d'insérer un nouveau couple `(cle, valeur)` dans le tableau
– `get(K cle)` permet d'obtenir la valeur associée à la clé `cle`
J'ai utilisé ces méthodes dans `Historique`. Restait à pouvoir parcourir l'ensemble des intitulés depuis la classe `CompteCourant` pour afficher un relevé complet des opérations effectuées sur le compte. Si l'on consulte la documentation javadoc de `HashMap` [4], il existe une méthode `keySet` dans `HashMap` qui permet d'obtenir l'ensemble des clés de la table dans une instance de `Set`. Malheureusement, l'ensemble retourné ne permet pas de parcourir ses éléments par ordre d'insertion, mais par ordre naturel. Pour les chaînes de caractères (les intitulés des opérations pour nous), l'ordre naturel est l'ordre lexicographique. On se retrouve donc avec des relevés affichant les opérations par ordre alphabétique. Pour pallier ce problème,

2. Nous verrons plus loin que la solution « basique » consistant à utiliser deux listes était peut-être plus judicieuse.

j'ai donc utilisé une instance de `ArrayList` pour stocker les intitulés dans l'ordre d'insertion. La première solution était donc la bonne... J'ai toutefois conservé la table de hachage pour vous présenter cette structure de données.
Je disposais donc maintenant d'une classe `Historique` permettant de gérer correctement les couples intitulé/montant. Attention, il faut toutefois que les intitulés des opérations soient différents pour les distinguer !

5.2 Classe `Object` et `hashCode`

Justement, quelle est la fonction de hachage qui est utilisée ? Si on regarde plus précisement la classe `Object` dont toutes les classes héritent, on s'aperçoit qu'il existe une méthode `hashCode` dedans qui retourne un entier. C'est en fait cette méthode qui est appelée à chaque fois que l'on a besoin de hacher un objet pour l'utiliser comme clé dans une table de hachage. Et c'est la valeur renvoyée par cette méthode qui est utilisée par défaut dans la méthode `toString` de `Object` (cf. documentation javadoc de `Object`).
Lorsque l'on définit une classe, on est censé redéfinir la méthode `hashCode`, ainsi que les méthode `equals` et `clone` qui sont présentes dans `Object`. [2] est une bonne référence qui peut vous aider à implanter une méthode `hashCode`.

5.3 Mise en forme dans `CompteCourant`

Pour pouvoir afficher correctement les relevés de compte, il fallait pouvoir mettre en forme ces derniers. En particulier, pour chaque ligne du relevé
– la largeur de la colonne contenant l'intitulé
– la largeur de la colonne contenant les crédits
– la largeur de la colonne contenant les débits
devaient être identiques.
Vous avez vu l'an dernier dans la bibliothèque standard du langage C la fonction `printf` qui permet de formater l'affichage sur la sortie standard. Depuis Java 5, on peut également le faire sur `System.out` (ou sur toute autre instance de `PrintWriter`) via la méthode `printf(String format, Object... args)`. La syntaxe de `printf` en Java est donc très proche de celle que vous connaissez en C : on donne dans un premier argument sous forme d'une chaîne de caractères le texte « fixe » à afficher et les spécifications de format, puis ce qu'il faut formater dans les arguments suivants. La syntaxe précise des spécifications de format est donnée dans la documentation javadoc de la classe `java.util.Formatter` [4]. Par exemple :
– `"%2$20.2f"` est une spécification de format pour le deuxième[3] argument donné à `printf` (`%2$`), l'affichage devra se faire sur au minimum 20 colonnes (`20`), et on attend un réel qui sera affiché avec 2 chiffres après la virgule (`.2f`).
– `"%1$-30s"` est une spécification de format pour le premier argument donné à `printf` (`%1$`), l'affichage sera justifié à gauche (`-`), l'affichage devra se faire sur au minimum 30 colonnes (`30`), et on attend une chaîne de caractères (`s`).
J'ai donc fourni dans `Historique` des méthodes permettant de connaître la longueur maximale des intitulés, des montants des débits et des crédits et j'ai utilisé ces valeurs dans `CompteCourant` pour mettre en forme correctement l'affichage. Vous remarquerez que j'ai utilisé deux méthodes privées, `printEnteteReleve` et `printPiedReleve` pour factoriser l'affichage de l'entête et de la fin du relevé.
Si vous souhaitez utiliser la mise en forme via `printf`, faites attention : les erreurs dues à un mauvais format (par exemple utiliser `".2f"` alors que l'argument passé n'est pas un réel) ne sont pas détectées à la compilation mais apparaissent sous forme d'exceptions levées à l'exécution de votre programme ! Il faut donc bien le tester...
Il y avait encore un point de détail : il fallait parfois afficher une séquence de – et la longueur de cette séquence n'était pas fixe. On ne peut pas en Java (contrairement à d'autres langages comme Python par exemple) construire facilement une chaîne de caractères en répétant un motif[4]. J'ai utilisé la technique suivante :
– utilisation du constructeur de `String` prenant en paramètre un tableau de **`char`**. La longueur de ce tableau me permettait de fixer la longueur de la chaîne. Par exemple **`new`** `String(`**`new char`**`[15])` permet de construire une chaîne de longueur 15 ;
– la chaîne ainsi construite contient la valeur par défaut de **`char`** à chaque position. Cette valeur par défaut est `"\0"`. J'ai donc utilisé la méthode `replace` de `String` pour remplacer `"\0"` par le caractère voulu, ici `"-"`.
Évidemment, cela était fastidieux et ne vous était pas demandé. Vous disposez toutefois maintenant d'un exemple vous montrant comment faire. Vous pouvez également remarquer que la bibliothèque Commons Lang du projet Apache [1] fournit une classe `StringUtils` qui possède une méthode statique `repeat` qui fait ce que l'on veut ici. De façon générale, le projet Commons de la fondation Apache peut être très utile pour trouver des API de qualité pour résoudre un problème particulier.

3. En ne comptant pas le premier argument qui est la chaîne de caractères contenant les directives de formatage.
4. Dans l'idéal, on aimerait pouvoir écrire comme en Python `"-"* 5` pour construire la chaîne `"-----"`...

Références

[1] *Apache Commons Lang*. URL : http://commons.apache.org/lang.

[2] J. Bloch. *Effective Java : programming language guide*. Addison-Wesley Professional, 2001.

[3] T.H. Cormen et al. *Introduction à l'algorithmique*. 2e éd. Dunod, 2004.

[4] *Java API specifications*. URL : http://download.oracle.com/javase/6/docs/api/index.html.