IN201 : 5 – 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 http://www.tofgarion.net/lectures/IN201.

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`. On peut toutefois revenir sur l'appel à `afficher` sur `pn1`. `afficher` est une méthode publique de la classe `Point` :

```
public void afficher() {
    System.out.println(this);
}
```

Le paramètre de la méthode `println` doit être de type `String`. Lorsqu'une référence est utilisée, le compilateur ajoute un appel à `toString` sur la référence. Le code réellement exécuté est donc `System.out.println(this.toString())`. `toString` est une méthode de la classe `Object`, mais elle a été redéfinie dans `Point` :

```
@Override public String toString() {
    return ("(" + this.x + "," + this.y + ")");
}
```

Lors de l'appel `pn1.afficher()`, le compilateur va vérifier qu'il y a une méthode `afficher` définie dans `PointNomme`. C'est le cas, car la méthode `afficher` de `Point` étant publique, elle est héritée par `PointNomme`. Comme elle n'est pas redéfinie, c'est elle qui sera exécutée. Lors de l'exécution de `afficher`, il y a un appel à `toString`. Or `toString` est également redéfinie dans `PointNomme` :

```
    @Override public String toString() {
        return ("" + getNom() + ":" + super.toString());
    }
```

Le principe de liaison dynamique nous indique que c'est la méthode `toString` de `PointNomme` qui sera utilisée ici, même si c'est la méthode `afficher` définie dans `Point` qui est utilisée.

```
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`, en particulier la méthode `toString` permettant l'affichage de `q` (cf. explications précédentes).

```
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: error: incompatible types
        qn = p1;        // Est-ce autorise ? Pourquoi ?
             ^
  required: PointNomme
  found:    Point
```

```
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: error: incompatible types
        qn = q; // Est-ce autorise ? Pourquoi ?
             ^
  required: PointNomme
  found:    Point
```

```
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.isae.geometry.Point cannot be cast to fr.isae.geometry.PointNomme
        at fr.isae.geometry.TestPolymorphisme.main(TestPolymorphisme.java:75)
```

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.

   Solution :
   La méthode equals définie initialement dans la classe Point était la suivante :

   ```
       public boolean equals(Point p) {
           return (p != null) && (this.x == p.x) && (this.y == p.y);
       }
   ```

   On pouvait vérifier que cette méthode n'était pas une redéfinition de la méthode equals de Object en ajoutant @Override devant la déclaration de la méthode. On obtient alors une erreur à la compilation :

   ```
   Point.java:193: error: method does not override or implement a method from a supertype
       @Override public boolean equals(Point p) {
       ^
   1 error
   ```

   En effet, la méthode equals de Object prend un Object en paramètre. La méthode définie dans Point est donc une méthode surchargée et pas une méthode redéfinie.
   Pour redéfinir la méthode equals de Point, il suffisait de changer le type du paramètre de equals dans Point (et sa documentation javadoc !) :

   ```
       @Override public boolean equals(Object o) {
   ```

2. implanter la méthode equals de Point. Vérifier dans un test JUnit simple que la méthode fonctionne bien dans tous les cas.

   Solution :
   L'implantation de la méthode equals est la suivante :

   ```
       @Override public boolean equals(Object o) {
           if (! (o instanceof Point)) {
               return false;
           }

           return (this.x == ((Point) o).x) && (this.y == ((Point) o).y);
       }
   ```

   Quelques remarques :
   - if fallait dans un premier temps vérifier si o était une instance de Point ou pas. Si ce n'est pas le cas, on renvoie **false** ;
   - il fallait ensuite comparer les coordonnées des deux points. Pour cela, il faut transtyper o qui est de type apparent Object en Point. On peut le faire, car on a testé précédemment si o était bien un point.

   Une méthode de test JUnit simple que l'on pouvait utiliser est par exemple la suivante (p1 et p2 sont deux instances de Point qui sont initialisées avec des coordonnées différentes) :

```
@Test public void testEquals() {
    assertTrue(p1.equals(p1));
    assertFalse(p1.equals(p2));
    assertFalse(p2.equals(p1));

    p2.setX(p1.getX());
    p2.setY(p1.getY());

    assertTrue(p1.equals(p2));
    assertTrue(p2.equals(p1));

    assertFalse(p1.equals(new Object()));
    assertFalse(p1.equals(new java.util.ArrayList<Point>()));
    assertFalse(p1.equals(null));
}
```

On voit en particulier que si o est **null**, le fait d'utiliser **instanceof** permet de renvoyer faux.

3. redéfinir également la méthode equals de PointNomme. Peut-on utiliser la méthode equals de Point dans l'implantation de cette méthode ?

   Solution :
   On peut tout à fait réutiliser la méthode equals de Point dans la méthode equals de PointNomme, c'est même à mon avis une très bonne solution : on évite de copier/coller du code et on utilise l'égalite logique déja définie entre points ! Par contre, la vérification de typage effectuée dans la méthode equals de Point ne sert à rien.
   Voici en tout cas la méthode :

```
@Override public boolean equals(Object o) {
    if (! (o instanceof PointNomme)) {
        return false;
    }

    return super.equals(o) && (this.nom.equals(((PointNomme) o).nom));
}
```

   Il fallait faire attention à utiliser equals pour comparer des chaînes de caractères.

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 ?

   Solution :
   Voici la méthode de test (j'ai utilisé des points créés localement, il faudrait normalement définir des acteurs sous forme d'attributs pour que cela soit plus propre) :

```
@Test public void testEqualsSupp() {
    Point p = new Point(1.0, 2.0);
    PointNomme pn = new PointNomme(1.0, 2.0, "A");

    assertTrue(p.equals(pn));
    assertFalse(pn.equals(p));
}
```

   Quelques explications :
   – p est bien « égal » à pn car on utilise la méthode equals de Point qui ne compare que les coordonnées des points. Attention, **instanceof** ne vérifie pas que l'objet est instance de la classe Point, mais que l'on peut transtyper la référence vers Point ce qui fonctionne évidemment avec PointNomme.
   – par contre, pn n'est pas « égal » à p car on ne peut pas transtyper p vers PointNomme, donc la méthode equals de PointNomme renvoie **false**.

Il peut paraître étrange que « p = pn » mais que « pn ≠ p », mais l'égalité logique n'est pas définie comme une égalité mathématique. Comme chaque classe peut définir sa propre « égalité », la propriété de symétrie ne s'applique pas ici. Je vous propose de regarder la section 6.1 pour approfondir le sujet.

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. 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.

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`.

   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.

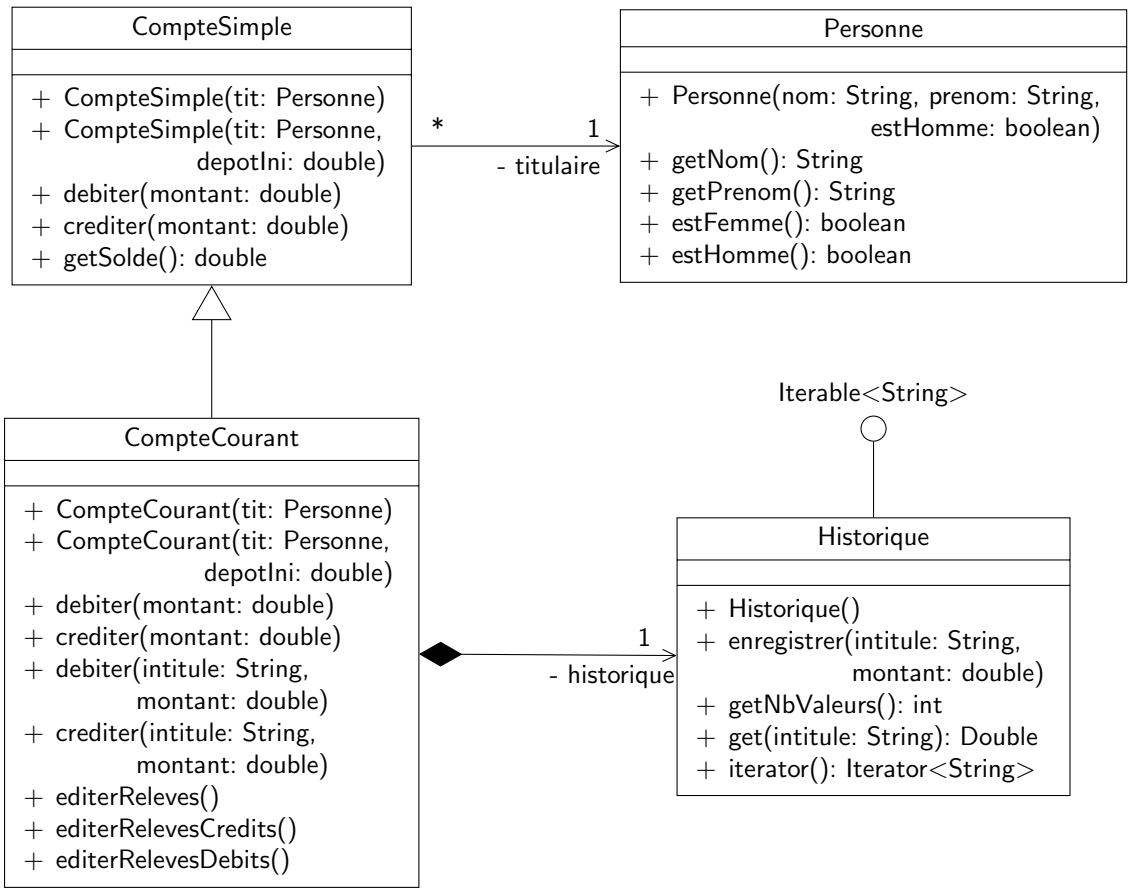

FIGURE 1 – Diagramme de classe présentant `CompteSimple`, `Personne`, `CompteCourant` et `Historique`

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).
   Si vous regardez le code des redéfinitions de `crediter` et `debiter`, je fais simplement un appel aux méthodes `crediter` et `debiter` prenant un intitulé en paramètre en leur passant un intitulé qui est une chaîne de caractères vide. Cela permet de factoriser du code. Les méthodes `crediter` et `debiter` prenant un intitulé en paramètre appelent quant à elles les méthodes `crediter` et `debiter` de `CompteSimple`. Par exemple voici les deux méthodes `crediter` de `CompteCourant` :

```
@Override public void crediter(double montant) {
    this.crediter("", montant);
}
```

```
public final void crediter(String intitule, double montant) {
    super.crediter(montant);
    this.historique.enregistrer(intitule, montant);
}
```

   Je vous propose quelques approfondissements dans la section 6 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 une classe de tests JUnit `CompteCourantTest`. Cette classe de test doit-elle utiliser la classe `CompteSimpleTest` ? Si oui, pourquoi ? On pourra modifier éventuellement `CompteSimpleTest`.

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`. Il faut modifier la classe de test `CompteSimpleTest` pour pouvoir changer l'acteur de test utilisé dans `CompteSimpleTest` (cf. exercice fait dans le cours sur l'héritage). 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.isae.bank;

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@isae.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);
    }
}
```

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

Listing 2– La classe `CompteCourantTest`

```
package fr.isae.bank;

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@isae.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++;
            }
        }
```

```
    }
}
```

Cette classe de test donne bien le résultat suivant :

```
Testsuite: fr.isae.bank.CompteCourantTest
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec

Testcase: testDebiterWithIntitule took 0 sec
Testcase: testHistoriqueDebiter took 0 sec
Testcase: testCrediterWithIntitule took 0 sec
Testcase: testHistoriqueCreation took 0 sec
Testcase: testHistoriqueCrediter took 0 sec
Testcase: testCrediter took 0 sec
Testcase: testGetSolde took 0 sec
Testcase: testDebiter took 0 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.isae.bank` (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 une liste d'instances de `CompteSimple` et afficher les relevés des instances de `CompteCourant` figurant dans le tableau.

   Solution :

   Le source est disponible sur le site. 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.

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 ?

Solution :
Il paraît naturel de spécialiser la classe `CompteSimple` si on veut créer une classe `LDD`. Il faut dans un premier temps réfléchir aux conséquences de cette spécialisation.
En effet, si `LDD` est une spécialisation de `CompteSimple`, d'après le principe de substitution, toute instance de `LDD` peut remplacer une instance de `CompteSimple`. En particulier, on doit pouvoir appeler la méthode `crediter` sur une instance de la classe `LDD` et cette méthode doit fournir au moins le même comportement que celle de la classe `CompteSimple`. Or si le montant de crédit a pour conséquence de porter le solde du LDD à plus de 12000€, on ne doit pas accepter l'opération. Par exemple, un utilisateur créant une liste de `CompteSimple` et qui crédite tous les comptes de 13000€ [2] s'attend à ce que tous les comptes soient crédités de ce montant, même si certains des comptes sont des LDD (la documentation de `CompteSimple` nous dit que les montants sont crédités).
On ne peut donc pas spécialiser la classe `CompteSimple` en une classe `LDD`.
Cette question était bien sûr une question « piège ». On peut évidemment construire en Java une classe `LDD` qui redéfinit la méthode `crediter` de telle façon que celle-ci ne crédite pas le LDD avec le montant passé en paramètre si celui-ci implique un dépassement du solde. 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 LDD est un compte (particulier certes), il n'y a pas de doute... Donc c'est la classe `CompteSimple` qui est mal conçue : il faudrait prévoir dans `crediter` des cas anormaux. Nous verrons dans la suite du cours que cela peut être géré par le mécanisme d'*exception*.

6 Pour aller plus loin...

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

6.1 La méthode `equals` de `PointNomme`

Rappelons l'implantation de la méthode `equals` que nous avons choisie dans `PointNomme` :

```
@Override public boolean equals(Object o) {
    if (! (o instanceof PointNomme)) {
        return false;
    }

    return super.equals(o) && (this.nom.equals(((PointNomme) o).nom));
}
```

On voit immédiatement (cf. section 3) que cette méthode `equals` n'est pas symétrique : une instance de `Point` peut être « égale » à une instance de `PointNomme`, mais la même instance de `PointNomme` ne sera pas égale à cette instance de `Point`. Si l'on consulte la documentation Javadoc de la classe `Object` [1], on constate toutefois que la méthode `equals` de `Object` doit vérifier les propriétés suivantes :

propriété	**`equals` de `PointNomme`**
réflexivité	✔
symétrie	✘
transitivité	✔
cohérence	✔
appel avec référence **`null`**	✔

Le principe de substitution nous impose de vérifier ces propriétés qui ont été définies au niveau de la classe `Object` dont `PointNomme` hérite. On pourrait proposer l'implantation suivante pour `equals` :

```
@Override public boolean equals(Object o) {
    if (! (o instanceof Point)) {
        return false;
    }

    // si o est un Point, on utilise la methode equals de Point !
```

```
        if (! (o instanceof PointNomme)) {
            return o.equals(this);
        }

        return super.equals(o) && (this.nom.equals(((PointNomme) o).nom));
    }
```

La première vérification via **instanceof** utilise maintenant `Point` et plus `PointNomme`. On vérifie ensuite si o n'est pas une instance de `Point`. Si c'est le cas, on utilise la méthode `equals` de `Point` en « renversant » l'appel. On vérifie facilement que la propriété de symétrie est maintenant vérifiée. Par contre on a perdu la transitivité :

```
Point      p   = new Point(1, 1);
PointNomme pn1 = new PointNomme(1, 1, "A");
PointNomme pn2 = new PointNomme(1, 1, "B");

System.out.println(pn1.equals(p)); // true, c'est equals de Point
                                   // qui est utilise

System.out.println(p.equals(pn2)); // true, c'est equals de Point
                                   // qui est utilise

System.out.println(pn1.equals(pn2)); // false, c'est equals de PointNomme
                                     // qui est utilise
```

Y-a-t'il un moyen de s'en sortir ? **Non**. C'est un problème fondamental concernant les relations d'équivalence dans les langages orientés objet. On peut éventuellement s'en sortir en utilisant une composition entre `PointNomme` et `Point` au lieu de l'héritage, mais on perd alors le sous-typage (cf. [2] pour plus de détails).
Attention, n'oubliez pas qu'on a essayé ici de pouvoir comparer un point avec un point nommé, ce qui n'est pas forcément naturel... En particulier, on peut regarder de plus près la méthode `hashCode` de `Object` (qui est utilisée dans les tableaux associatifs, cf. section suivante). La méthode `hashCode` est une méthode renvoyant un entier que l'on appelle le *hash* ou le *hachage* de l'objet sur lequel elle est appelée. Le *hash* d'un objet correspond à un nombre permettant d'identifier cet objet. Par exemple, c'est la valeur renvoyée par cette méthode qui est utilisée par défaut dans la méthode `toString` de `Object` (cf. documentation javadoc de `Object`).
Si l'on regarde le contrat de `hashCode` dans `Object`, on voit que deux objets égaux au sens de `equals` doivent avoir le même *hash*. Or, le calcul du *hash* d'un objet dépend (classiquement) de la valeur des attributs de cet objet (cf. [2] pour plus de détails). Donc normalement une instance de `Point` et une instance de `PointNomme` n'auront pas le même *hash* même si elles ont des coordonnées identiques. On devrait donc systématiquement renvoyer **false** si on utilise un objet de type `PointNomme` en paramètre de la méthode `equals` de `Point`.
Maintenant, comme on utilise **instanceof** dans le premier test de la méthode `equals` de `Point`, on ne filtre pas « assez bien » le type réel de l'objet passé en paramètre de la méthode (**instanceof** va renvoyer **true** avec un sous-type de `Point`). Il faudrait renvoyer **false** dès que le type réel de l'objet passé en paramètre n'est pas « exactement » `Point`. Pour cela, on peut s'appuyer sur la méthode `getClass` de la classe `Object`. Cette méthode renvoie une instance de la classe `Class` qui représente la classe réelle de l'objet sur laquelle elle a été appelée. On va donc écrire la méthode `equals` de `Point` de la façon suivante :

```
    @Override public boolean equals(Object o) {
        if ((o == null) ||
            ! (this.getClass().equals(o.getClass()))) {
            return false;
        }

        return (this.x == ((Point) o).x) && (this.y == ((Point) o).y);
    }
```

Par contre, on perd la possibilité d'utiliser **super.**`equals` dans `PointNomme` :

```
    @Override public boolean equals(Object o) {
        if ((o == null) ||
            ! (this.getClass().equals(o.getClass()))) {
            return false;
        }
```

```
        return (this.getX() == ((PointNomme) o).getX()) &&
            (this.getY() == ((PointNomme) o).getY()) &&
            (this.nom.equals(((PointNomme) o).nom));
    }
```

On vérifiera facilement que cette fois-ci, tout fonctionne correctement (au prix de quelques efforts ☺).

Lorsque l'on définit une classe, on est censé redéfinir la méthode `hashCode`, ainsi que les méthodes `equals` et `clone` qui sont présentes dans `Object`. [2] est une bonne référence qui peut vous aider à implanter une méthode `hashCode` et à redéfinir proprement `clone` et `equals`.

6.2 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*[3] [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` [1], 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, 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 !

Quelle est la méthode qui est utilisée comme fonction de hachage ? C'est en fait la méthode `hashCode` 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 (cf. section précédente).

6.3 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

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

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` [1]. Par exemple :

- `"%2$20.2f"` est une spécification de format pour le deuxième[4] argument donné à `printf` (`%2$`), 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[5]. 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 [4] 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.

Références

[1] Oracle. *Java API specifications*. 2013. URL : http://docs.oracle.com/javase/7/docs/api/index.html.

[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] Apache Software Foundation. *Apache Commons Lang*. 2013. URL : http://commons.apache.org/lang.

License CC BY-NC-SA 3.0

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