Chapitre 12

Héritage

Ce chapitre du cours traite de concepts relatifs à la programmation objet (hiérarchie de classe, héritage, extension, masquage) et sera illustré par un exemple de représentation de comptes bancaires et d'opérations liées à la manipulation de ceux-ci (retrait, dépôt, consultation).

12.1 Une classe simple pour représenter des comptes bancaires

Décrivons maintenant la classe permettant de représenter un compte bancaire. Il faut, au minimum, connaître le propriétaire (que l'on représentera simplement par son nom), le numéro du compte (que l'on représentera par tableau de caractères) et la somme disponible sur le compte (représentée par un double). Pour les opérations sur un compte, on se limite aux opérations de retrait (on décrémente si possible le compte d'une somme donnée), le dépôt (on augmente le compte d'une somme donnée) et la consultation (on retourne la somme disponible sur un compte). Dans le cas où un retrait est impossible on lèvera une exception de type `provisionInsuffisanteErreur` définie séparément. En plus du constructeur par défaut, nous fournirons également un constructeur plus complet permettant de construire un compte en fixant le numéro, le propriétaire et le solde initial. Enfin nous proposons également une méthode permettant d'effectuer un virement d'un compte vers un autre.

Ceci conduit à la construction de la classe suivante :

Listing 12.1 – (lien vers le code brut)

```
1  public class CompteBancaire{
2      String nomProprietaire ;
3      char[] numero ;
4
5      double solde ;
6
7      public CompteBancaire(){
8      }
9
10     public CompteBancaire(String proprio, char[] num, double montant){
11         this.nomProprietaire = proprio ;
12         this.numero = num ;
13         this.solde = montant ;
14     }
15
16     public double getSoldeCourant (){
```

```
17          return this.solde ;
18      }
19
20      public void deposer(double montant){
21          this.solde = solde + montant ;
22      }
23
24      public void retirer(double montant){
25          System.out.println("Appel de retrait sur compte simple");
26          if (this.solde < montant){
27              throw new provisionInsuffisanteErreur() ;
28          }else{
29              solde = solde - montant ;
30          }
31      }
32
33      public void virerVers(CompteBancaire c, double montant){
34          c.retirer(montant);
35          this.deposer(montant);
36      }
37  }
38
39  class provisionInsuffisanteErreur extends Error{
40  }
```

On peut alors utiliser cette classe dans le programme suivant :

Listing 12.2 – (lien vers le code brut)

```
1   class Test1ComptesBancaires {
2       public static void main(String[] args){
3           CompteBancaire c1 ;
4           CompteBancaire c2 ;
5           CompteBancaire c3 ;
6           String num1 = "123456789";
7           String num2 = "145775544";
8           String num3 = "A4545AA54";
9           c1 = new CompteBancaire("Paul", num1.toCharArray(), 1000.00);
10          c2 = new CompteBancaire("Paul", num2.toCharArray(), 2300.00);
11          c3 = new CompteBancaire("Henri", num3.toCharArray(), 5000.00);
12          c1.deposer(100.00);
13          c2.virerVers(c1, 1000.00);
14          Terminal.ecrireDouble(c2.getSoldeCourant());
15      }
16  }
```

12.2 Extension des données, notion d’héritage et syntaxe

Supposons maintenant que l’on souhaite prendre en considération la possibilité d’avoir un découvert sur un compte (un retrait conduit à un solde négatif) ou la possibilité d’associer une rémunération des dépôts sur un compte (compte rémunéré) ; ces cas ne sont pas prévus dans la classe que nous avons

définie. Une première possibilité est de modifier le code source de la classe précédente. Ceci peut impacter tout programme (ou toute classe) qui utilise cette classe et conduit à une difficulté : comment définir un compte rémunéré sans découvert ou un compte avec découvert possible mais non rémunéré. Il faudrait dans ce cas dupliquer à la main ces classes pour les particulariser ; il n'est pas difficile d'imaginer l'ensemble des modifications qu'entraînerait cette solution sur les programmes utilisant la classe `CompteBancaire`.

Les langages de programmation modernes offrent différentes solutions à ce problème ; le but est chaque fois de permettre une extension ou une spécialisation des types de base. Avec la programmation par objets ceci se réalise naturellement avec la notion **d'héritage**. Le principe consiste à définir une nouvelle classe à partir d'une classe existante en "ajoutant" des données à cette classe et en réutilisant de façon implicite toutes les données de la classe de base. Si A est la classe de base et si B est la classe construite à partir de A on dit que B "hérite" de A ou encore que B "spécialise" A ou encore que B "dérive" de A. On dira également que A est la classe "mère" de B et que B est une sous-classe, ou une classe "fille" de A. Dans le cas où une classe hérite d'une seule classe (c'est le cas en Java) on parle d'héritage simple (au lieu d'héritage multiple) et la relation mère-fille définit un arbre comme représenté par le dessin suivant où $B1$ et $B2$ héritent de A et où C hérite de $B1$ (et donc également par transitivité de A). On notera qu'en Java toute classe hérite de la classe `Object`.

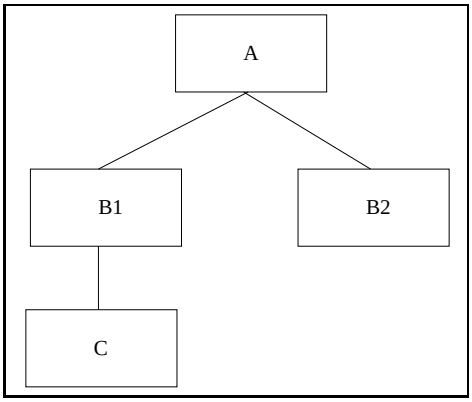

FIG. 12.1 – Hiérarchie de classes

Pour être plus précis, lors de la construction d'une nouvelle classe par extension d'une classe existante, on distingue trois sortes de méthodes (ou de variables) :

- les méthodes (et les variables) qui sont propres à la nouvelle classe ; il s'agit là d'extension. Ces méthodes (données) n'existent que pour les objets de la nouvelle classe et portent un nom qui n'est pas utilisé dans la classe mère ou, un nom déjà utilisé mais avec des paramètres en nombre ou de type différents (il y a alors une surcharge du nom de la méthode).
- les méthodes (et les variables) qui sont issues de la classe mère ; il s'agit là d'héritage. Ces méthodes (données) sont les même pour les objets de la classe mère et pour les objets de la classe héritée. Par défaut, toute méthode (variable) est héritée. On notera que lors de l'utilisation d'une méthode héritée sur un objet de la classe fille il y a une conversion implicite de l'objet de la classe fille en un objet de la classe mère. Le code d'une méthode (ou la donnée) héritée est soit partagé lorsque la méthode est statique soit propre à chaque instance dans le cas contraire (cas par défaut).
- les méthodes (ou les variables, mais cela est plus rare) qui redéfinissent des méthodes (variables) existantes dans la classe mère ; il s'agit là de masquage. Ces méthodes ont le même nom et le même profil (nombre et type des paramètres, valeur de retour) que les méthodes qu'elles

redéfinissent. Dans ce cas, la nouvelle méthode se substitue à la méthode de la classe mère. Ceci permet de prendre en compte l'enrichissement (ou la spécialisation) que la classe fille apporte par rapport à la classe mère en spécialisant le code de la méthode. Lorsque dans une classe dérivée on souhaite désigner une méthode (ou une donnée) de la classe mère (et en particulier lorsque celle-ci est masquée) on désigne celle-ci en préfixant le nom de la méthode (ou de la variable) par `super`. Ainsi, si B dérive de A et si x est un nom utilisé dans A et B, la dénomination `super.x` dans une méthode de B désignera x relatif à A et non x relatif à B (qui serait désigné par `this.x`). Dans le cas particulier du constructeur, un appel à `super` désigne un appel au constructeur approprié (celui dont le nombre et le type des paramètres conviennent) de la classe mère.

La syntaxe utilisée en Java pour définir qu'une classe B étend une classe A consiste à préciser lors de la déclaration de la nouvelle classe cette extension avec la notation `class B extends A`. Tout ce qui sera défini au niveau de cette nouvelle classe B sera propre à B ; tout ce qui n'est pas redéfini sera hérité de A, c'est-à-dire qu'un objet instance de B pourra accéder à ces données ou méthodes comme un objet instance de A.

Appliquons ces principes à la construction de deux classes dérivées de la classe `CompteBancaire`. La première permettra de représenter des comptes bancaires avec découvert autorisé ; la seconde prendra en compte une possible rémunération du compte. Pour le premier type de compte, il faut préciser quel est le découvert maximum autorisé. Pour le second type de compte, il faudra définir quel est la rémunération (taux) et, par exemple, quel est le seuil minimal à partir duquel la rémunération s'applique.

12.3 Extension de la classe `CompteBancaire` en `CompteAvecDecouvert`

Pour cette nouvelle classe nous introduisons une nouvelle donnée : le montant maximum du découvert. Cela nous conduit à modifier la méthode `retrait` de telle sorte qu'un retrait puisse être effectuer même si le solde n'est pas suffisant (il faudra néanmoins que le solde reste plus grand que l'opposé du découvert maximal autorisé). Cette méthode `retrait` masquera la méthode de même nom de la classe mère. Par ailleurs, on introduira une nouvelle méthode `fixeDecouvertMaximal` qui permet de modifier le montant du découvert maximum autorisé. Les autres données et méthodes seront reprises telles quelles (par le mécanisme d'héritage). Pour la définition du constructeur on utilise le constructeur de la classe mère (par un appel à `super`). Notons que `super` est toujours appelé dans un constructeur soit de manière implicite en début d'exécution du constructeur soit de manière explicite par un appel à `super`.

Listing 12.3 – (lien vers le code brut)

```
1  public class CompteAvecDecouvert extends CompteBancaire{
2      double decouvertMax ;
3
4      public void fixerDecouvertMaximal(double montant){
5          this.decouvertMax = montant ;
6      }
7
8      public CompteAvecDecouvert (String proprio,
9                                  char[] num,
10                                 double montant,
```

```
11                                  double decouvertMax){
12          super(proprio, num, montant) ;
13          this.decouvertMax = decouvertMax;
14      }
15
16      public void retirer(double montant){
17          // Terminal.ecrireStringln("Appel de retrait sur compte avec decouvert");
18          if (this.solde - montant < -decouvertMax){
19              throw new provisionInsuffisanteErreur() ;
20          }else{
21              solde = solde - montant ;
22          }
23      }
24  }
```

On peut alors completer le programme de test par les instructions suivantes :

Listing 12.4 – (lien vers le code brut)

```
1           CompteAvecDecouvert d1;
2           String num4 = "DD545AA54";
3
4           d1 = new CompteAvecDecouvert("Jacques", num4.toCharArray(),
5                                        6000.00, 0.00) ;
6           d1.retirer(7000.00);
7           Terminal.ecrireStringln("retrait bien passee ;" +
8                                   "nouveau solde = " + d1.getSoldeCourant() );
9           d1.fixeDecouvertMaximal(2000.00);
10          d1.retirer(7000.00);
11          Terminal.ecrireStringln("retrait bien passee ;"+
12                                  "nouveau solde = " + d1.getSoldeCourant() );
13          d1.deposer(5000.00);
```

12.4 Extension de la classe `CompteBancaire` en `CompteAvecRemuneration`

Pour cette extension, il faut au minimum connaitre le taux de rémunération du compte à partir duquel la rémunération s'applique. On suppose que les intérêts sont versés à part et intégrés sur le compte une fois l'an. Afin de simplifier l'écriture de cette classe nous supposerons qu'il existe une méthode qui calcule les intérêts (le code est un peu complexe et nécessite la connaissance des dates auquelles sont faites les opérations afin de calculer les durées pendant lesquels l'inétrêt s'applique).

Listing 12.5 – (lien vers le code brut)

```
1   public class CompteRemunere extends CompteBancaire{
2       double taux ;
3       double interets;
4
5       public void fixerTaux(double montant){
6           this.taux = montant ;
7       }
8
```

```
9       public CompteRemunere (String proprio, char[] num, double montant,
10                             double taux){
11          super(proprio, num, montant) ;
12          fixerTaux(montant);
13          interets = 0.0;
14      }
15
16      public void retirer(double montant){
17          // Terminal.ecrireStringln("Appel de retrait sur compte remunere");
18          if (this.solde < montant){
19              throw new provisionInsuffisanteErreur() ;
20          }else{
21              solde = solde - montant ;
22          }
23      }
24
25      public void calculerInteret(){
26          interets = interets + 1.0; // bien sur, le code reel est plus complexe
27      }
28
29  }
```

Je propose de supprimer l'exemple qui suit, qui correspond à une mauvaise utilisation de l'héritage (on duplique du code)

On peut alors définir un compte rémunéré avec découvert autorisé. Il suffit pour cela de dériver de la classe `CompteRemunere` la classe `CompteRemunereAvecDecouvert`. Cela se fait simplement de la façon suivante (il suffit d'introduire une nouvelle variable et un nouveau constructeur et de redéfinir la méthode de retrait) :

Listing 12.6 – (lien vers le code brut)

```
1   public class CompteRemunereAvecDecouvert extends CompteRemunere{
2       double decouvertMax ;
3
4       public void fixeDecouvertMaximal(double montant){
5           this.decouvertMax = montant ;
6       }
7
8       public CompteRemunereAvecDecouvert (String proprio,
9                                           char[] num,
10                                          double montant,
11                                          double taux,
12                                          double decouvertMax){
13          super(proprio, num, montant, taux) ;
14          // ici super designe la classe CompteRemunere
15          fixeDecouvertMaximal(decouvertMax);
16      }
17
18      public void retrait(double montant){
19          System.out.println(
20            "Appel de retrait sur compte remunere avec decouvert");
21          if (this.solde - montant < -decouvertMax){
22              throw new provisionInsuffisanteErreur() ;
```

```
23          }else{
24              solde = solde - montant ;
25          }
26      }
27  }
```

On obtient alors l'arbre de classes suivant :

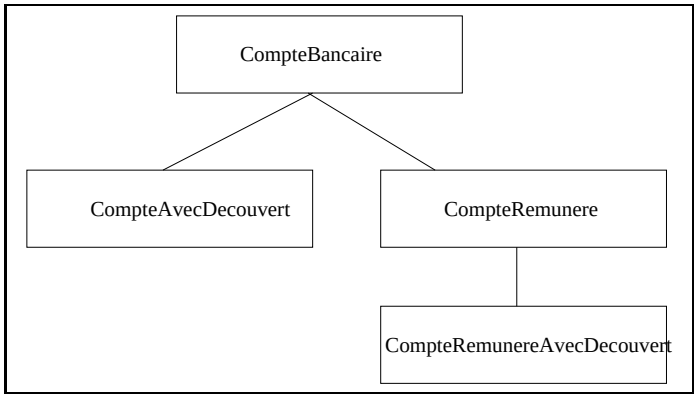

FIG. 12.2 – Hiérarchie des classes définies dans ce cours

Cette nouvelle classe peut être utilisée de la façon suivante :

Listing 12.7 – (lien vers le code brut)

```
1           dd = new CompteRemunereAvecDecouvert("Pierre", num5.toCharArray(),
2                                                6000.00, 0.00, 0.05) ;
3
4           dd.depot(1000.00);
5           dd.retrait(8000.00);
6           System.out.println(
7                   "retrait 1 bien passee sur dd; nouveau solde = "
8                   + dd.soldeCourant() );
9           dd.fixeDecouvertMaximal(2000.00);
10          dd.retrait(8000.00);
11          System.out.println(
12                  "retrait 2 bien passee sur dd; nouveau solde = "
13                  + dd.soldeCourant() );
14          dd.depot(5000.00);
15          dd.fixeTaux(0.06);
```

On notera que, la méthode `retrait` utilisée lors de ces instructions est celle qui est redéfinie dans la nouvelle classe. La méthode `fixeTaux` est par contre la méthode hérité de la classe `CompteRemunere` et les méthodes `depot` et `soldeCourant` sont celles héritées de la classe mère `CompteBancaire`. On le voit, l'héritage est transitif.

12.5 Transtypage, classe déclarée et classe d'exécution

En programmation fortement typée le mélange des types n'est permis qu'avec le respect de règles strictes. Il est par exemple interdit d'affecter une valeur réelle à une variable booléenne. Il existe

néanmoins des possibilités de convertir une valeur d'un type donnée en une valeur d'un autre type ; on parle de transtypage ou de conversion de type (cast). Cette conversion peut être implicite lorsqu'elle est fait automatiquement par le compilateur (conversion directe ou insertion de code de conversion) ou explicite par l'emploi d'une notation appropriée. Ce mécanisme a déjà été étudié en début d'année pour les types élémentaires de Java.

La règle liée à la programmation objet est que toute classe est compatible avec ses sous-classes c'est à dire que si B est une sous-classe de A alors toute variable de type A peut être affectée par une valeur de type B (par une instruction d'affectation ou lors d'un passage de paramètre). Par exemple, si $c1$ est un compte bancaire et si $d1$ est un compte bancaire avec découvert autorisé alors l'affectation $c1 = d1$ est correcte. Par contre, l'affectation $d1 = c1$ sera incorrecte. De même une instruction `c1.virement(d1, 100.0)` sera correcte et il y aura une conversion implicite de $d1$ en une valeur de type `compteBancaire` lors de l'appel de la méthode `virement`. On peut bien entendu également exécuter l'instruction `d1.virement(c1, 100.0)`.

Lors de l'appel `c1.virement(d1, 100.0)`, le paramètre formel de la méthode virement désignant le compte sur lequel on va opérer un retrait (nommons le c) est associé au paramètre effectif $d1$. Une conversion a lieu. Cependant, c qui est de classe déclarée `CompteBancaire` sera de classe réelle `CompteAvecDecouvert` lors de l'exécution de cet appel (il restera associé au paramètre effectif $d1$ qui est une instance de la classe `CompteAvecDecouvert`. Ainsi, l'instruction `c1.retrait(montant)` de cette méthode fera appel à la méthode `retrait` de la classe `CompteAvecDecouvert` qui est la classe réelle de c lors de cet appel. Par contre, l'appel de `d1.virement(c1, 100.0)`, conduit à un appel de la méthode méthode `retrait` de la classe `CompteBancaire` car cette fois-ci, le paramètre formel c est associé à la variable $c1$ qui est un objet de la classe `CompteBancaire` et donc, la classe réelle de c lors de cette exécution est `CompteBancaire`. Les instructions suivants permettent de visualiser ces principes (les méthodes `retrait` affichent des messages différents).

Listing 12.8 – (lien vers le code brut)

```
1        d1.depot(5000.00);
2        c1.virement(d1, 1000.00);
3        d1.virement(c1, 1000.00);
```

12.6 Liaison dynamique

Dans les exemples précédents, les méthodes appelées à l'exécution du programme sont connues à la compilation.

Supposons maintenant que l'on déclare un compte bancaire c sans l'initialiser :

Listing 12.9 – (lien vers le code brut)

```
1        CompteBancaire c;
2        String num = "AAA4AA54";
```

Les règles de typages de Java font qu'il est possible d'associer à c par création un compte ordinaire ou un compte avec découvert. En effet, toute classe est compatible avec ses sous-classes, et donc, c qui est de la classe `CompteBancaire` est compatible avec la classe `CompteAvecDecouvert`.

Les deux instructions

Listing 12.10 – (lien vers le code brut)

```
1     c = new CompteBancaire("Marie", num.toCharArray(), 10000.00) ;
```

Listing 12.11 – (lien vers le code brut)

```
1     c = new CompteAvecDecouvert ("Marie", num.toCharArray(), 10000.00, 0.0) ;
```

sont donc correctes ; dans le premier cas, c sera de classe réelle `CompteBancaire` et dans le second cas de classe réelle `CompteAvecDecouvert`. Donc, l'instruction `c.retrait(100.00)` appelera soit la méthode `retrait` des comptes avec découvert (second cas) soit la méthode `retrait` des comptes ordinaires. Cette liaison entre appel et méthode peut se faire lors de la compilation (dans le cas où le compilateur a suffisamment d'informations pour calculer cette liaison) ou simplement lors de l'exécution du programme ; on parle alors de liaison tardive. La suite d'instructions suivantes illustre ce mécanisme : selon la réponse faite par l'utilisateur, le programme associe à c un compte ordinaire ou un compte avec découvert.

Listing 12.12 – (lien vers le code brut)

```
1  public class TestLiaisonTardive{
2
3      public static void main(String[] args){
4          CompteBancaire c;
5          String num = "AAA4AA54";
6
7          Terminal.ecrireString("Voulez vous creer un compte decouvert O/N : ");
8          char reponse;
9          reponse = Terminal.lireChar();
10         if ((reponse != 'O') && (reponse != 'o')){
11             Terminal.ecrireString("Creation d'un compte ordinaire");
12
13             c = new CompteBancaire("Marie", num.toCharArray(), 10000.00) ;
14         }
15         else{
16             Terminal.ecrireString("Creation d'un compte avec decouvert");
17
18             Terminal.ecrireString("Quel est le couvert maximal autorise : ");
19             double max;
20             max = Terminal.lireDouble();
21             c = new CompteAvecDecouvert ("Marie", num.toCharArray(),
22                                          10000.00, max) ;
23         }
24         System.out.println("solde avant retrait " + c.soldeCourant() );
25         System.out.println("Tentative de retrait de 11000.00");
26         c.retrait(11000.00);
27         System.out.println("le retrait c'est bien passee; nouveau solde = "
28                            + c.soldeCourant() );
29     }
30 }
```