Exceptions
Corrigé

Exercice 1 : Comprendre les exceptions

Dans cet exercice, on considère le programme donné au listing 1 qui compile sans erreur. Les identifiants sont volontairement non significatifs.

Listing 1 – Les exceptions en Java

```
1  class ExempleException {
2      private void m2(String p) {
3          System.out.print("<");
4          if (p == null) {
5              throw new NullPointerException();
6          }
7          if (p.length() == 0) {
8              throw new IllegalArgumentException("Chaîne vide");
9          }
10         System.out.print(p.charAt(0));
11         System.out.print(p.charAt(1));
12         System.out.print(">");
13     }
14
15     public void m1(String p) {
16         System.out.print("[");
17         try {
18             System.out.print("(");
19             m2(p);
20             System.out.print(")");
21         } catch (NullPointerException e) {
22             System.out.print("N");
23         } catch (IllegalArgumentException e) {
24             System.out.print("I");
25         } finally {
26             System.out.print("F");
27         }
28         System.out.print("]");
29 }    }
30
31 class ClassePrincipale {
32     public static void main(String[] args) {
33         String argument = (args.length == 0) ? null : args[0];
34         new ExempleException().m1(argument);
35         System.out.println(".");
36 }    }
```

1.1. Indiquer ce qu'affiche l'exécution des commandes suivantes :

```
1  java ClassePrincipale un
2  java ClassePrincipale ""
3  java ClassePrincipale x
```

Solution : Voici le résultat des exécutions :

```
> java ClassePrincipale un
[(<un>)F].

> java ClassePrincipale ""
[(<IF].

> java ClassePrincipale x
[(<xFException in thread "main" java.lang.StringIndexOutOfBoundsException: String index
↪  out of range: 1
    at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:47)
    at java.base/java.lang.String.charAt(String.java:693)
    at ExempleException.m2(ClassePrincipale.java:11)
    at ExempleException.m1(ClassePrincipale.java:19)
    at ClassePrincipale.main(ClassePrincipale.java:34)
```

Remarques :

1. Les instructions d'une clause **`finally`** sont toujours exécutées (à partir du moment ou le **`try`** correspondant a été franchi), qu'une exception se produise ou pas, qu'elle soit récupérée ou pas.
2. Quand une exception est levée, l'exécution du bloc s'interrompt et l'exception se propage. Elle remonte de bloc en bloc (y compris appels de méthodes) jusqu'à rencontrer un gestionnaire d'exception (**`catch`**) qui lui correspond. L'exception reprend alors avec les instructions de ce gestionnaire d'exception. Si aucun gestionnaire d'exception ne récupère l'exception, la machine virtuelle arrête le programme avec un message qui commence par « Exception in thread... ».

1.2. L'ordre des **`catch`** est-il important ?

Solution : Oui, car ils sont testés dans l'ordre. Le premier qui convient est pris. Il faut donc toujours commencer par les types d'exceptions les plus précis.

Mais ce n'est pas très grave de se tromper car le compilateur signalera l'erreur.

1.3. Les exceptions `IllegalArgumentException` et `NullPointerException` sont-elles vérifiées ? En particulier, on indiquera ce qui fait qu'une exception est vérifiée en Java, qui vérifie et l'intérêt de cette notion.

Solution : Les exceptions vérifiées en Java sont les classes qui sont sous-types de `Throwable` (pour être une exception) et qui ne sont ni sous-type de `RuntimeException`, ni sous-type de `Error`.

C'est le compilateur qui vérifie : « exception vérifiée » signifie que le compilateur signalera un message d'erreur si le programmeur n'a pas dit explicitement ce qu'il faisait de l'exception : soit il la traite et il a mis un **`catch`**, soit il la laisse se propager et il doit mettre un **`throws`**.

Ici la méthode m2 n'a pas de throws et pourtant elle lève et laisse se propager (pas de **`catch`**) les exceptions mentionnées. Les deux exceptions ne sont donc pas vérifiées. Effectivement, elles sont sous-type de `RuntimeException`.

Exercice 2 : Somme des arguments de la ligne de commande

La classe Somme (listing 2) affiche la somme des nombres réels donnés en argument de la ligne de commande. Par exemple, « `java Somme 10 15.5 4` » affiche `29.5`.

Listing 2 – La classe Somme

```java
/** Calculer la somme des paramètres de la ligne de commande.  */
public class Somme {

    /* Afficher la somme des arguments de la ligne de commande */
    public static void main(String[] args) {
        double somme = 0;
        for (int i = 0; i < args.length; i++) {
            somme += Double.parseDouble(args[i]);
        }
        System.out.println(somme);
    }

}
```

2.1. L'exécution de `java Somme 10 x 3` affiche ce qui suit dans le terminal :

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "x"
    at java.base/jdk.internal.math.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2054)
    at java.base/jdk.internal.math.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.base/java.lang.Double.parseDouble(Double.java:543)
    at Somme.main(Somme.java:8)
```

Expliquer comment interpréter cet affichage pour comprendre ce qu'il s'est passé.

Solution : La première ligne nous apprend trois choses :

1. la non robustesse du programme : il s'est terminé sur une exception non récupérée (« Exception in thread "main" »), .
2. l'exception qui s'est produire : ici java.lang.NumberFormatException,
3. le message qui explique pourquoi l'exception a été levée : pour la chaîne de caractère « x ».

Les lignes suivantes correspondent aux différents appels de méthodes depuis le programme principal (ligne du bas) jusqu'à l'instruction qui a provoqué la levée d'exception (ligne du haut).

L'exception `NumberFormatException` a été levée ligne 2054 du fichier `FloatingDecimal.java`, dans la méthode `readJavaFormatString` de la classe `FloatingDecimal` du paquetage nommé `jdk.internal.math` du module java.base. En partant du bas, on a exécuté la méthode `main` de `Somme` jusqu'à la ligne 8 où on a appelé la méthode `parseDouble` de la classe `Double`, exécutée jusqu'à la ligne 543 où elle appelle la méhode `parseDouble` de la classe `FloatingDecimal`, exécutée jusuqu'à la ligne 110 où elle appelle la méthode `readJavaFormatString` de `FloatingDecimal` exécutée jusqu'à la ligne 2054 où elle lève l'exception `NumberFormatException`.

L'auteur de la classe `Somme` déduit de cette exécution que le programme n'est pas robuste car il se termine sur une exception `NumberFormatException`, due au fait que x ne respecte pas le format des réels. La méthode `parseDouble` (en réalité `readJavaFormatString`) détecte le problème, lève une exception qui provoque l'arrêt de l'exécution du programme et la remontée des appels de méthodes. Comme aucun gestionnaire d'exceptions n'a été défini, l'exécution du programme s'arrête et la trace des appels est affichée (comportement de la machine virtuelle).

2.2. Modifier la classe Somme pour afficher la somme de tous les arguments de la ligne de commande en ignorant ceux qui ne sont pas réels. On indiquera le nombre de données ignorées.

```
> java Somme 10 x 3 y
13.0
Nombre de données ignorées : 2
> java Somme 10 3
13.0
```

Solution :

```java
/** Calculer la somme des paramètres de la ligne de commande.  */
public class SommeDonneesValidesNbErreurs {

    /* Afficher la somme des arguments de la ligne de commande */
    public static void main(String[] args) {
        double somme = 0;
        int nbErreurs = 0;
        for (int i = 0; i < args.length; i++) {
            try {
                somme += Double.parseDouble(args[i]);
            } catch (NumberFormatException e) {
                nbErreurs++;
            }
        }
        System.out.println(somme);
        if (nbErreurs > 0) {
            System.out.println("Nombre données ignorées : " + nbErreurs);
        }
    }

}
```

Remarques :

1. Au lieu de `NumberFormatException` dans le **`catch`**, on pourrait mettre un type plus général comme `Exception` mais ce serait maladroit (voire une erreur) car on pourrait alors récupérer des exceptions correspondant à d'autres problèmes et les traiter comme un problème de format de nombre. Le vrai problème est alors caché et difficile ensuite à retrouver !

   Conseil : Toujours mettre dans un **`catch`** le type le plus précis qui correspond au traitement que l'on veut faire.

2. Certains étudiants souhaitent faire un test explicite pour savoir si `args[i]` correspond bien à un réel. Si ce choix n'est pas logique dans cet exercice : le première question nous montre l'exception qui se passe en cas d'argument incorrect sur la ligne de commande, il est donc logique de la récupérer si on souhaite traiter le problème. Cependant, si on veut faire un test, on arrive alors à la structure suivante :

```java
for (int i = 0; i < args.length; i++) {
    if (/* args[i] correspond à un réel */) {
        somme += Double.parseDouble(args[i]);
    } else {
```

```
            nbErreurs++;
        }
    }
```

On remarque que la structure du code est la même : au **if/else** et **try/catch** près. Avec le **try/catch**, on est optimiste et on fait la conversion. Si elle ne fonctionne pas, on fera un traitement particulier. Avec le **if** explicite, on s'assure que la conversion marchera et sinon on fait le traitement particulier.

Reste à voir comment écrire la question du **if**. Une idée souvent proposée est de faire :

```
if (args[i] instanceof Double) {
```

Bien sûr ceci ne marche pas. Un objet de type String ne peut pas être considéré comme un objet de type Double. La classe String n'est pas un sous-type du type Double. Ceci provoquera d'ailleurs une erreur de compilation ici.

L'opérateur **instanceof** ne convient pas ici car on souhaite vérifier que les caractères de la chaîne correspondent à un réel.

Pour faire ce test, on pourrait utiliser les expressions régulières et vérifier que la chaîne correspond à l'expression régulière qui décrit les nombres réels.

Ici la solution avec exception est plus intéressante car avec le test explicite on va faire deux fois la vérification du bon format (dans le cas où le format des réels est respecté par la chaîne) : avec l'expression régulière d'abord et `parseDouble` le fera aussi ensuite. L'exception évite ici un traitement redondant.

3. Utiliser une **foreach** serait plus logique ici que de gérer explicitement un indice sur le tableau args. On écrirait :

```
for (String arg : args) {
    ...
    somme += Double.parseDouble(args)
    ...
}
```

Exercice 3 : Livret A

Un Livret A est un produit bancaire dont le titulaire peut faire des opérations de dépôt dans la limite d'un plafond et des retraits dont le montant ne peut pas dépasser le solde du livret. Les sommes versées sur un Livret A donnent lieu à rémunération. Le taux d'intérêt ainsi que le plafond du Livret A sont identiques pour tous les Livrets A. Au 12 mars 2015, le taux d'intérêt est fixé à 1% et le plafond à 22 950 euros.

Tout comme pour les comptes courants, un historique des opérations est géré par la banque. Il est ainsi possible d'accéder aux dernières opérations de crédit ou de débit et d'éditer un relevé.

3.1. Compléter le diagramme de classes UML du système pour faire apparaître les Livrets A.

Solution : Un livret A ressemble à un compte courant : il a les mêmes opérations créditer, débiter et éditer le relevé. On peut obtenir son solde et son titulaire. Ceci n'est cependant pas un argument pour utiliser l'héritage dans un langage comme Java [1] qui lie héritage et sous-typage.

1. En c++, par exemple, on peut dissocier héritage et sous-typage.

La question qu'il faut se poser est : est-ce que livret A est un sous-type de compte courant ? En d'autres termes : peut-on utiliser un livret A là où on attend un compte courant ?

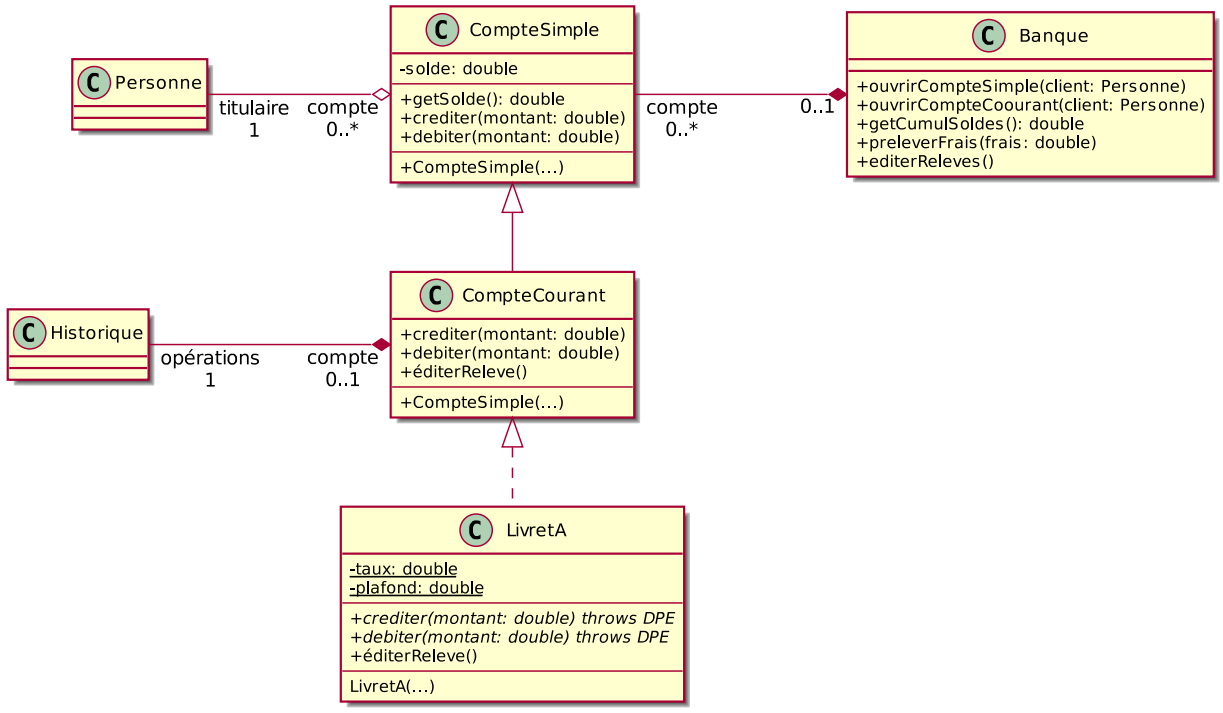
À cette réponse, il faut répondre non. Un livret A n'est pas un sous-type de compte courant. Par exemple, il ne réussira pas les programmes de test de compte simple qui fait que le solde passer en négatif.

Dans la suite, nous allons faire un raisonnement par l'absurde et faire donc hériter `LivretA` de `CompteCourant` pour constater que ce n'est effectivement pas une bonne solution.

Si on choisit l'héritage (à tord ici), on doit ensuite se demander s'il y a des méthodes de la super-classe (`CompteCourant`) à redéfinir. Ici, il faut redéfinir créditer et débiter pour tenir compte des contraintes du livret A.

Les informations supplémentaires à définir sont le taux d'intérêt et le plafond. On ne traitera pas le calcul des intérêts car il faudrait connaître les dates de valeur des opérations, ce que nous ne gérons pas...

Le taux d'intérêt est une information identique (ou commune) à tous les Livrets A. En conséquence, elle ne doit pas être attachée à un Livret A particulier mais à la classe des Livrets A. C'est donc une information de classe. Il en va de même pour le plafond. Comme ces deux informations concernent les livrets A, il est logique de les définir dans la classe LivretA, sous forme de requêtes de classe, puis d'attributs de classe (et non d'instance). En UML on souligne les membres de classe. En Java, ils ont le modifieur **`static`**.



Avant de passer à la suite, voyons tout de suite pourquoi faire hériter `LivretA` de `CompteCourant` (ou `CompteSimple`) conduit à une incohérence.

On considère la méthode `deposer` du code suivant et le programme principal. Est-ce que les contrats de `deposer` sont corrects ? Que devrait afficher l'exécution du programme principal.

```
1  /** Scénario d'utilisation des comptes.  */
2  public class ExempleDeposer {
3
4      /** Opération d'un automate qui permet à un client de déposer une somme
5       * d'argent sur un de ses comptes.
6       * @param c le compte sur lequel le client souhaite déposer de l'argent
7       * @param somme l'argent que le client veut mettre sur le compte c
8       */
9      //@ requires somme > 0;    // somme valide
10     //@ requires c != null;    // il y a bien un compte
11     //@ ensures c.getSolde() == \old(c.getSolde()) + somme;
12     //@            // somme déposée sur le compte
13     public static void deposer(CompteSimple c, double somme) {
14         c.crediter(somme);
15     }
16
17     public static void main (String args []) {
18         Personne p1 = new Personne("Xavier", "Crégut", true);
19
20         // Déposer de l'argent sur son compte courant
21         CompteCourant compte = new CompteCourant(p1, 5000);
22         deposer(compte, 20000);
23         System.out.println("solde du compte = " + compte.getSolde());
24
25         // Déposer de l'argent sur son livret A
26         LivretA livret = new LivretA(p1, 5000);
27         deposer(livret, 20000);
28         System.out.println("solde du livret = " + livret.getSolde());
29     }
30
31 }
32
```

Les contrats de `deposer` sont effectivement corrects. Le scénario avec le compte courant fonctionne très bien. En revanche, le scénario avec le livret A ne peut pas fonctionner. On est confronté à une contradiction : le contrat de `deposer` n'est pas compatible avec l'invariant du Livret A. Le solde ne peut pas être à la fois égal à 25000 et inférieur à 22950 !

Cette contradiction vient du fait que nous avons utilisé l'héritage alors qu'il n'y a pas sous-typage !

3.2. Écrire la classe LivretA.

Solution :

On a vu qu'il fallait redéfinir les méthodes créditer et débiter. Intéressons nous à créditer. Que faire si le montant fait dépasser le plafond ? On pourrait ne rien faire (où va l'argent ?), créditer dans la limite du plafond (mais que devient le reste ?)... Ce n'est pas à la méthode créditer de `LivretA` de faire ce choix. En conséquence, il est préférable de signaler que l'opération demandée ne peut pas être réalisée. Le bon moyen est d'utiliser une exception. Définissons une nouvelle exception DepassementPlafondException.

C'est une nouvelle exception, il faut donc la définir. Doit elle être vérifiée ou pas ? Dans la mesure où elle est là pour la robustesse du programme, parce qu'il est préférable de réagir si une opération de crédit d'un livret A ne fonctionne pas, nous allons la définir vérifiée et donc nous allons hérité de Exception (et non de RuntimeException). Voici sa définition.

```
/** DepassementPlafondException signale un dépassement de plafond d'un LivretA. */
public class DepassementPlafondException extends Exception {

    public DepassementPlafondException(String message) {
        super(message);
    }

}
```

On peut maintenant écrire la classe `LivretA`.

```
/** Modélisation d'un Livret A (mauvaise façon : héritage de CompteCourant)
 * @author  Xavier Crégut
 */
public class LivretA extends CompteCourant {

    //@ public invariant getSolde() >= 0;    // découverts interdits !
    //@ public invariant getSolde() <= getPlafond();    // compte plafonné
    //@ public invariant getTauxInterets() > 0;
    //@ public invariant getPlafond() > 0;

    //@ // Invariants de liaison (entre la spécification (requêtes)
    //@ // et la réalisation (attributs)
    //@ private static invariant LivretA.getTauxInterets() == LivretA.tauxInterets;
    //@ private static invariant LivretA.getPlafond() == LivretA.plafond;

    private static double tauxInterets = 0.75;    // en pourcentage
    private static double plafond = 22950;    // en euros

    /** Construction d'un Livret A
     * @param titulaire le titulaire du compte
     * @param depotInitial le montant initial du compte
     * @exception DepassementPlafondException si le dépôt initial provoque un
     *             dépassement du plafond du livret
     */
    //@ requires depotInitial >= 10;    // montant initial strictement positif
    //@ requires depotInitial <= plafond;    // respect du plafond
    //@ ensures getSolde() == depotInitial;    // solde initialisé
    //@ ensures getTitulaire() == titulaire;    // titulaire initialisé
    public LivretA(Personne titulaire, double depotInitial) {
        super(titulaire, depotInitial);
    }

    /** Le taux  d'intérêt d'un Livret A. */
    public static /*@ pure @*/ double getTauxInterets() {
        return tauxInterets;
    }
```

```
37
38      /** Plafond de dépot d'un Livret A. */
39      public static /*@ pure @*/ double getPlafond() {
40          return plafond;
41      }
42
43      /**
44       * Créditer le compte du montant (exprimé en euros) et enregistrer
45       * l'opération de crédit.
46       * @param montant montant déposé sur le compte en euros
47       */
48      //@ also
49      //@ requires montant >= 0;        // montant strictement positif
50      //@ requires montant <= getPlafond() - getSolde();
51      //@                    // ne pas dépasser le plafond
52      @Override
53      public void crediter(double montant)
54          throws DepassementPlafondException
55      {
56          if (montant + getSolde() > getPlafond()) {
57              throw new DepassementPlafondException("Dépassement du plafond :"
58                          + " dépôt de " + montant
59                          + " ==> dépassement de : "
60                          + (this.getSolde() + montant - plafond));
61          }
62
63          super.crediter(montant);
64      }
65
66      /**
67       * Débiter le compte du montant (exprimé en euros) et enregistrer
68       * l'opération de crédit.
69       * @param montant montant retiré du compte en euros
70       */
71      //@ also
72      //@ requires montant > 0;        // montant strictement positif
73      //@ requires montant <= getSolde();    // pas de découvert !
74      @Override
75      public void debiter(double montant)
76          throws DecouvertException
77      {
78          if (montant > this.getSolde()) {
79              throw new DecouvertException("Découvert : retrait de " + montant
80                      + " > solde = " + this.getSolde());
81          }
82          super.debiter(montant);
83      }
84
85  }
```

Le compilateur signale une erreur lors de la compilation sur les méthodes `crediter` et `debiter`. Pourquoi ?

Parce que ces méthodes peuvent lever une exception qui n'a pas été déclarée dans la mé-

thode correspondante de la classe `CompteCourant` (et `CompteSimple`). Il faut donc modifier les classes `CompteSimple` et `CompteCourant`. Ceci montre que l'héritage n'était pas neutre puisqu'il oblige de modifier la sémantique des opérations de `CompteSimple` et `CompteCourant`. Fallait-il réellement hériter ?

Non ! D'autant plus que s'il on ajoute les **`throws`** sur les méthodes créditer et débiter de `CompteSimple` et `CompteCourant`, il faudra reprendre tous les programmes utilisent ces méthodes pour dire ce qu'ils font de ces exceptions (qui ne peuvent pas se produire mais qui sont spécifiées dans la signature des méthodes).

Utiliser un exception non vérifiée aurait éviter que le compilateur signale une erreur mais le problème reste bien là : voir la classe ExempleDeposer ci-dessus !

La programmation par contrat nous aurait aussi permis de voir que l'héritage est une mauvaise solution ici. En effet la précondition de créditer dans `LivretA` est plus restrictive que dans `CompteCourant` : on ajoute que le montant ne doit pas provoquer un dépassement du plafond. On est donc pas dans un cas de sous-typage : la précondition d'une redéfinition doit être plus faible que la précondition de la méthode redéfinie. Ici elle est plus forte !

On renonce donc à l'héritage. **Comment écrire la classe `LivretA` sachant que son code reste proche de celui de `CompteCourant` ?**

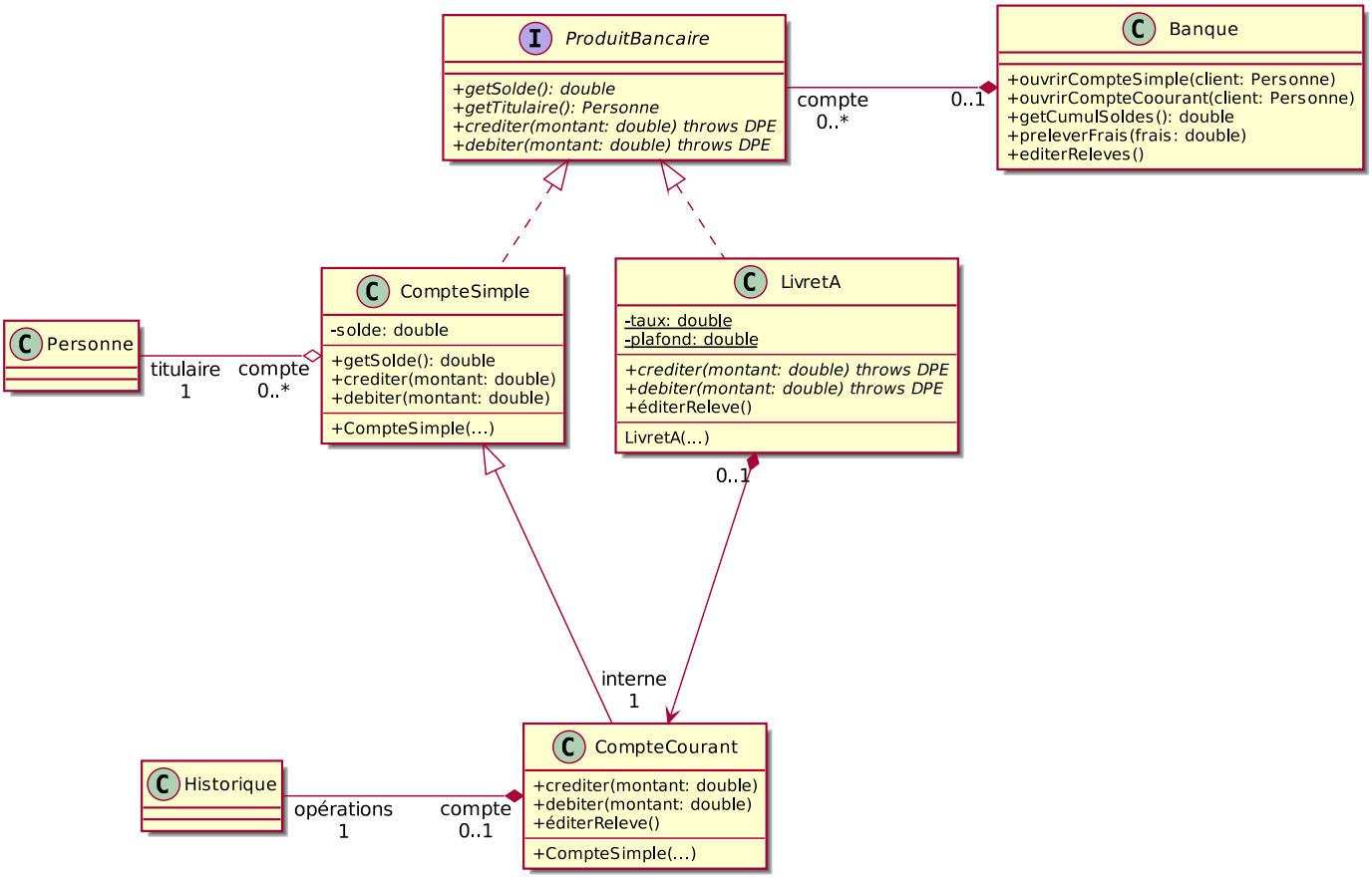
On peut s'appuyer sur (et donc utiliser) la classe `CompteCourant` en ayant recours à une relation d'utilisation, ici une relation de composition entre `LivretA` et `CompteCourant`. Les opérations de `LivretA` appelleront les opérations correspondante de `CompteCourant`. Pour créditer et débiter, on vérifiera au préalable les contraintes imposées par le `LivretA`. C'est la deuxième solution envisagée en cours pour définir la classe PointNommé sachant que la classe Point existe.

Pour la banque, peut-on stocker les comptes et les livrets dans un même tableau ?

Comme le suggère l'énoncé de l'exercice sur le livret A, on pourrait définir une notion `ProduitBancaire`. Que peut-on définir dessus ? Ce qui est présent à la fois dans `CompteSimple` et `LivretA`. Il y a bien sûr le solde, le titulaire. Il y a aussi créditer et débiter qui ont la même signature dans les comptes et les livrets. Cependant, même si `ProduitBancaire` est une interface, il faudra donner leur signification, en particulier les conditions d'application.

On n'est pas bloqué, mais il faut prendre la description la plus contraignante et donc spécifier qu'il peut y avoir une exception de type dépassement de plafond.

Dans le classe `CompteSimple`, on pourra adapter la spécification pour dire qu'aucune exception ne sera levée (on fonctionne dans plus de cas). La spécification initiale de `CompteSimple` (et `CompteCourant`) est inchangée. Dans `LivretA` on gardera bien sûr la spécification de l'exception !



3.3. Proposer un programme de test de la classe `LivretA`.

Solution : Le programme de test doit vérifier que les limites du livret sont respectées et que les exceptions DepassementPlafondException et DecouvertException sont levées dans les cas contraire.

Cette question était surtout là pour démontrer que l'héritage de `CompteCourant` n'était pas une bonne solution. En effet, une sous-classe doit réussir les programmes de test de sa super-classe. Donc `LivretA` devrait réussir ceux de `CompteCourant` et donc `CompteSimple`. Or, un test de compte simple fait passer le solde en négatif ce qui est impossible sur un LivretA. Un LivretA n'est donc pas un sous-type de `CompteSimple` (ni `CompteCourant`).