Coté cours – Application médicaments pour le laboratoire GSB

Mission 3/4 : Bases de la POO Java, des classes et des objets

Propriétés	Description
Intitulé long	Utilisation et exploitation d'une application Java à partir de données libres de droit
Formation concernée	BTS Services Informatiques aux Organisations
Matière	Bases de la programmation
Présentation	L'objectif est de comprendre les bases de la POO, construction d'objets et appels de méthodes, en évoluant dans le code d'une application Java déjà implémentée.
Notions	D4.1 – Conception et réalisation d'une solution applicative A4.1.6 Gestion d'environnements de développement et de test A4.1.7 Développement, utilisation ou adaptation de composants logiciels Savoir-faire Programmer à l'aide d'un langage de programmation structurée Programmer en utilisant des classes d'objets fournies Utiliser un environnement de développement Appliquer des normes de développement Savoirs associés Concepts de base de la programmation objet Normes de développement
Transversalité	U22- Mathématiques, algorithmique appliquée
Pré-requis	Base de la programmation : variable, boucle, condition, fonction
Outils	Interface de développement intégrée (IDE) Android Studio, git
Mots-clés	Initiation à la programmation orientée objet, classe, objet, constructeur, new, signature et argument d'une méthode, paramètre d'une méthode, classe métier, classe technique, classe DAO, adresse mémoire, référence, hash code, git
Durée	6 heures
Auteur(es)	Fabrice Missonnier, relectures Hervé Le Guern, Olivier Capuozzo et Yann Barrot
Version	v 1.0
Date de publication	Mai 2019

Habituellement, le code des applications Android développées en interne par le laboratoire GSB n'est pas très bien structuré car il mélange, dans les mêmes classes Java, l'affichage à l'écran (l'interface graphique) et l'accès aux données (les médicaments que l'on souhaite afficher).

Kévin est très attentif au travail en groupe. De culture *devops*, il mise beaucoup sur les méthodes agiles pour travailler de manière plus efficace avec ses collègues. Lorsqu'on regarde la définition sur Wikipedia, un *devops* est un informaticien capable d'intervenir sur toutes les étapes de la création d'un logiciel, depuis le développement, l'intégration, les tests, la livraison jusqu'au déploiement, l'exploitation et la maintenance des infrastructures.

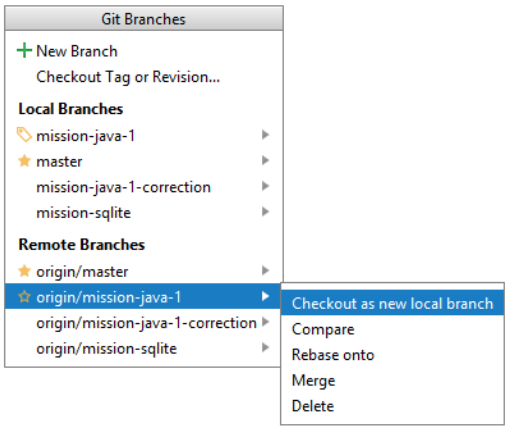
Les principes *devops* soutiennent des cycles de développement courts, une augmentation de la fréquence des déploiements et des livraisons.

La culture du *devops* et les méthodologies Agiles sont particulièrement intéressantes dans le contexte de GSB. L'informatique n'étant pas son cœur de métier, les équipes de développement ont intérêt à travailler sur des cycles très courts de développement. N'étant pas nombreux, il est aussi important pour les développeurs de connaitre l'ensemble des étapes de la création d'un logiciel.

Kévin souhaite que les médicaments soient instanciés sous forme d'objets dans le code source de l'application. C'est l'objectif de cette mission.

Organisation du code source du projet

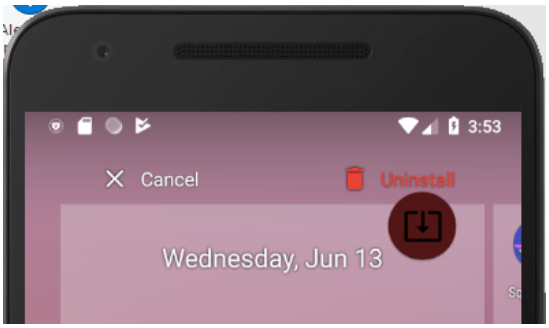
1- La première étape est de récupérer le code source programmé par Kévin. Si le projet a déjà été cloné, il faut simplement récupérer la branche `mission-java-1` (Menu *VCS/Git/Branches*) :



Avant d'exécuter le projet, il est préférable de nettoyer le code déjà compilé dans les missions précédentes.

2- Sous Android Studio, exécuter la commande *Clean project* dans le menu *Build.*

3- Supprimer l'application si elle a déjà été déployée sur la machine virtuelle Android (faire glisser l'icône dans le menu supprimer) :



Lorsqu'on exécute un projet sous Android Studio, il n'y a pas de méthode *main* comme en Java ou en C#, mais une « fenêtre » principale qui sera directement exécutée (une *Activity*). Elle est définie dans le fichier `manifests/AndroidManifest.xml` :

```
<activity android:name=".vue.MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

4- Exécuter l'application (Run 'app').

L'interface graphique doit afficher le résultat suivant :

Le code source d'une fenêtre est séparé dans deux fichiers, un fichier `xml` qui fournit le *design* de la page, et un fichier Java qui récupère les traitements lorsqu'un utilisateur effectue une action. C'est la partie interface, aussi nommée « vue » de l'application.

Vous allez intervenir sur la deuxième partie de l'application, celle qui gère les données.

I. Créer des objets, appeler des constructeurs

Un objet est un élément identifiable du monde réel (une voiture précise, un stylo donné, une entreprise, le temps, etc.). **Il est caractérisé par ce qu'il est** (ses données, son état), et **ce qu'il sait faire** (son comportement).

La classe est le type de l'objet : l'objet doliprane 500mg est une instance de la classe `Medicament`. Un objet a une vie : il est créé, a sa vie, meurt. Il peut potentiellement être hiberné et ressusciter…

Le code qui instancie les objets est dans la classe `DAO` du package `modele.dao`. Le nom de la classe a été choisi car c'est un standard en programmation. DAO signifie *Data Access Objet* (couche d'accès aux données) :

https://fr.wikipedia.org/wiki/Objet_d%27acc%C3%A8s_aux_donn%C3%A9es

Le programmeur va créer dans cette classe DAO les objets métiers dont il a besoin dans l'application.

Par exemple, pour créer un médicament, on utilise le code suivant

```
SimpleDateFormat in = new SimpleDateFormat( pattern: "YYYY-MM-dd HH:MM:SS", Locale.FRENCH);
Date convertedDate = new Date();
String dateS = "";
try {
    dateS = "2011-06-16 00:06:00.000";
    convertedDate = in.parse(dateS);
} catch (ParseException e) {
    e.printStackTrace();
}

Medicament m = new Medicament( codeCIS: "60216885", denomination: "RIVASTIGMINE ACTAVIS 1,5 mg, gélule", formePharma: "gélule",
        voieAdmin: "orale", statutAdminAMM: "Autorisation active", typeProcAMM: "Procédure centralisée",
        etatCommercialisation: "Non commercialisée", convertedDate, statusBDM: "Warning disponibilité",
        numAutEurop: "EU/1/11/693", titulaire: "ACTAVIS GROUP PTC (ISLANDE)", surveillance: false, conditionPrescription: "liste I");
```

Ces lignes de code permettent de construire un objet `m` typé en `Medicament`. Cette classe `Medicament` a déjà été écrite.

Les classes, en Java, sont définies dans un fichier portant leurs propres noms : par exemple, le fichier `Medicament.java` contient la classe `Medicament`. Ces classes, par convention, sont programmées dans un *package* `metier`.

On retrouve dans la classe `Medicament` le code du **constructeur** dont la **signature** correspond à la ligne :

```
public Medicament(String codeCIS, String denomination, String formePharma, String voieAdmin, String statutAdminAMM, String typeProcAMM,
                  String etatCommercialisation, Date dateAMM, String statusBDM, String numAutEurop, String titulaire, boolean surveillance,
                  String conditionPrescription) {
```

Dans la classe DAO, lors de l'instanciation de l'objet :

- l'argument `60216885` est placé dans le paramètre `codeCIS` du constructeur de la classe `Medicament` ;
- l'argument `RIVASTIGIME ACTAVIS 1,5mg, gélule` est placé dans le paramètre `denomination` du constructeur de la classe Medicament ;
- l'argument `gélule` est placé dans le paramètre `formePharma` du constructeur de la classe `Medicament` ;
- etc.

Le mot clé `public` indique qu'il est possible d'utiliser ce constructeur à partir d'une autre classe. Les arguments doivent correspondre exactement aux paramètres définis dans la signature du constructeur.

Dans l'exemple, pour créer un médicament, il faut 13 **arguments** qui sont passés aux **paramètres** du constructeur. **Chaque paramètre est typé** (`String` ou `boolean`).

Lors de la création d'un nouvel objet, Android Studio aide en notifiant le nom du paramètre destination correspondant à l'argument source. **Attention, ce n'est qu'une information, pas du code Java :**

```
Medicament m = new Medicament( codeCIS: "67873887",  denomination: "PREGABALINE BIOGARAN 150 mg, gélule",
```

Toujours par convention, en Java, le **nom de la classe commence par une majuscule**, **les variables (objets) commencent par une minuscule**.

La signature permet, pour le programmeur qui utilise la classe `Medicament`, de connaitre le nom du constructeur ainsi que les données effectives qu'il doit donner à l'appel. Android Studio, comme tous les environnements de développement, utilise cette signature pour l'aide à la programmation.

Le projet comporte 9 **classes « métiers »** déjà programmées :

Ces classes correspondent au domaine d'activité de l'application, au métier de GSB. Les autres classes (fenêtres de démarrage, accès aux données, etc.) sont appelées **classes « techniques »**.

Voici 4 médicaments issus de la base de données :

```
codeCIS = 60216885
denomination = RIVASTIGMINE ACTAVIS 1,5 mg,
gélule
formePharma = gélule
voieAdmin = orale
statutAdminAMM = Autorisation active
typeProcAMM = Procédure centralisée
etatCommercialisation = Non commercialisée
dateAMM = 2011-06-16 00:06:00.000
statusBDM = Warning disponibilité
numAutEurop = EU/1/11/693
titulaire = ACTAVIS GROUP PTC (ISLANDE)
surveillance = false
conditionPrescription = liste I
```

```
codeCIS = 68008765
denomination = RIVASTIGMINE ACTAVIS 4,5 mg,
gélule
formePharma = gélule
voieAdmin = orale
statutAdminAMM = Autorisation active
typeProcAMM = Procédure centralisée
etatCommercialisation = Non commercialisée
dateAMM = 2011-06-16 00:06:00.000
statusBDM = Warning disponibilité
numAutEurop = EU/1/11/693
titulaire = ACTAVIS GROUP PTC (ISLANDE)
surveillance = false
conditionPrescription = liste I
```

```
codeCIS = 61721627
denomination = RIVASTIGMINE ZYDUS 3 mg,
gélule
formePharma = gélule
voieAdmin = orale
statutAdminAMM = Autorisation active
typeProcAMM = Procédure nationale
etatCommercialisation = Commercialisée
dateAMM = 2010-05-14 00:05:00.000
statusBDM =
numAutEurop =
titulaire = ZYDUS FRANCE
surveillance = false
conditionPrescription = liste I
```

```
codeCIS = 61322336
denomination = RIVASTIGMINE BIOGARAN 4,6
mg/24 h, dispositif transdermique
formePharma = dispositif
voieAdmin = transdermique
statutAdminAMM = Autorisation active
typeProcAMM = Procédure nationale
etatCommercialisation = Commercialisée
dateAMM = 2014-10-30 00:10:00.000
statusBDM =
numAutEurop =
titulaire = BIOGARAN
surveillance = false
conditionPrescription = liste I
```

1- En vous aidant du code déjà programmé dans la méthode `instancierLesObjets()` de la classe `DAO`, créez les objets `Medicament` listés ci-dessus. Ajoutez les médicaments au tableau `lesMedicaments` et constatez leur affichage sur l'interface lors de l'exécution de l'application.

2- Testez l'exécution.

3- Toujours dans la méthode `instancierLesObjets()`, affichez les 4 médicaments créés dans la console *Logcat*, par exemple

```
Log.d("Médicament", m.toString());
```

4- Toutes les données passées en paramètres d'un nouveau médicament sont-elles affichées à l'écran du smartphone, dans la liste déroulante ? Sont-elles tout de même stockées en mémoire ?

Un objet `m` est créé avec l'instruction `new` qui appelle une **méthode** particulière de l'objet, le **constructeur**. Cette ligne permet d'initialiser les différents attributs de l'objet. Il est possible, mais pas obligatoire, de programmer un constructeur vide qui initialisera l'ensemble des attributs avec des valeurs par défaut : une chaîne vide pour les *String*, 0 pour un entier, etc. Kévin a choisi, pour le médicament, de ne pas programmer de constructeur vide.

Par exemple, si on tente de créer un médicament avec un constructeur vide (*no argument*), Android Studio indique qu'il ne connait qu'un constructeur avec treize paramètres :

```
Medicament ml = new Medicament();
```

Medicament() in **Medicament** cannot be applied to:

Expected Parameters:	Actual Arguments:
codeCIS:	String
denomination:	String
formePharma:	String
voieAdmin:	String
statutAdminAMM:	String
typeProcAMM:	String
etatCommercialisation:	String
dateAMM:	Date
statusBDM:	String
numAutEurop:	String
titulaire:	String
surveillance:	boolean
conditionPrescription:	String

Une **méthode** `toString()` a aussi été développée dans le code de la classe `Medicament`. Elle retourne une chaîne de caractères où sont concaténées toutes les valeurs de ses attributs. Après avoir créé l'objet `m`, on peut lui **appliquer une méthode** avec la « notation pointée », des parenthèses et des arguments (si nécessaire) :

```
String s = ml.toString();
```

Un même médicament peut être vendu sous plusieurs conditionnements (ou présentation). Chaque conditionnement a un code court (CIP sur 7 chiffres) ou un code long (CIP sur 13 chiffres). Le conditionnement fait référence à un médicament (codeCIS). Voici 4 exemples de présentations que l'on trouve dans la base de données :

```
codeCIP7 = 2218950
libellePres = plaquette(s) thermoformée(s)
aluminium PVC de 28 gélule(s)
statutPres = Présentation active
etatCommercialisation = Déclaration d'arrêt
de commercialisation
dateCommercialisation = 2016-04-30
00:04:00.000
codeCIP13 = 3400922189505
agrementCollectivites = non
tauxRbt = 15%
prixEuro = 11,10
texteRbt =
codeCIS = 60216885
```

```
codeCIP7 = 3821111
libellePres = plaquette(s) thermoformée(s)
PVC PVDC aluminium de 28 gélule(s)
statutPres = Présentation active
etatCommercialisation = Déclaration de
commercialisation
dateCommercialisation = 2012-08-01
00:08:00.000
codeCIP13 = 3400938211115
agrementCollectivites = oui
tauxRbt = 15%
prixEuro = 11,10
texteRbt =
codeCIS = 61721627
```

```
codeCIP7 = 2218967
libellePres = plaquette(s) thermoformée(s)
aluminium PVC de 56 gélule(s)
statutPres = Présentation active
etatCommercialisation = Déclaration d'arrêt
de commercialisation
dateCommercialisation = 2014-01-07
00:01:00.000
codeCIP13 = 3400922189673
agrementCollectivites = non
tauxRbt = 15%
prixEuro = 21,68
texteRbt =
codeCIS = 60216885
```

```
codeCIP7 = 3821192
libellePres = plaquette(s) thermoformée(s)
PVC PVDC aluminium de 56 gélule(s)
statutPres = Présentation active
etatCommercialisation = Déclaration de
commercialisation
dateCommercialisation = 2012-08-01
00:08:00.000
codeCIP13 = 3400938211924
agrementCollectivites = oui
tauxRbt = 15%
prixEuro = 21,68
texteRbt =
codeCIS = 61721627
```

5- Dans la méthode `instancierLesObjets()`, créer les objets de type `Presentation` en utilisant le constructeur de la classe dont la signature est

```
public Presentation(String codeCIP7, String libellePres, String statutPres, String etatCommercialisation,
        Date dateCommercialisation, String codeCIP13, String agrementCollectivites, String tauxRbt,
        String prixEuro, String texteRbt) {
```

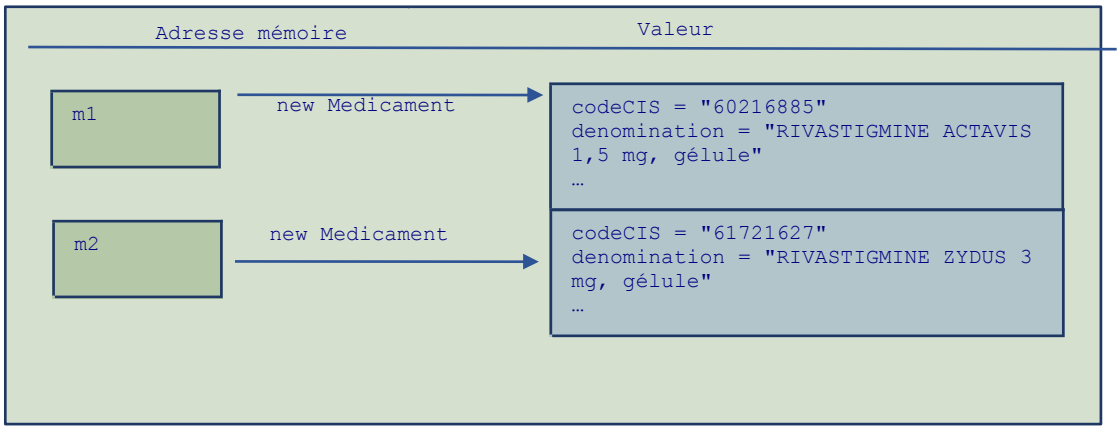
6- Afficher ces présentations dans la console *Logcat*.

7- Afficher ces présentations en appuyant sur le bouton « détail du médicament » de l'interface graphique. Testez pour chaque médicament. L'application rend-elle le bon résultat ? Pourquoi ?

Le résultat de cette dernière question n'est pas satisfaisant car le lien entre les objets médicaments et présentations n'a pas été programmé. C'est l'objectif de la deuxième partie de cette mission.

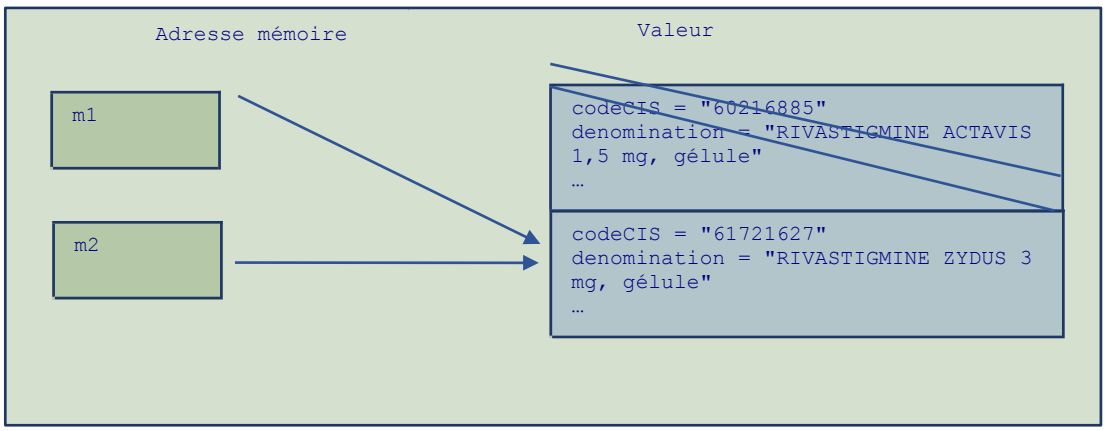
II. Différencier une classe d'un objet

En POO, un objet n'a pas de clé comme dans un modèle relationnel. C'est son emplacement en mémoire qui va le différencier d'un autre objet. Par exemple, lorsqu'on instancie deux objets `m1` et `m2` de type `Medicament` avec un `new`, `m1` et `m2` vont contenir deux « adresses » distinctes en mémoire. Le `new` alloue dynamiquement l'espace mémoire pour y stocker les valeurs correspondant aux attributs du `Medicament` :



Ainsi, comparer `m1 == m2` équivaut à comparer deux adresses mémoire, aussi appelées **références** (ou **pointeurs** dans d'autres langages comme le C ou le C++).

Si on utilise l'affectation `m1 = m2` après avoir créé les deux objets avec l'instruction `new`, les deux adresses pointeront sur le même espace de valeur :



La valeur créée par le premier `new` n'a plus de condition d'existence car plus aucune adresse n'y accède. Cet espace mémoire sera libéré par la JVM (le *garbage collector*).

Pour modifier une valeur d'un attribut d'un objet, on utilise une méthode particulière, **le mutateur**, décrite dans la troisième partie du document. Ici, pour modifier la valeur de `codeCIS` de `m1` et la remplacer par celle de `m2`, on utilisera l'instruction

```
m1.setCodeCIS("60216885");
```

8- Programmer l'inverse, en modifiant le code CIS de `m2` avec la valeur du code CIS de `m1`. Tester l'exécution du code en observant le log dans la console.

9- Affecter l'adresse de `m2` dans `m1`. Tester l'exécution du code en observant le *log* dans la console. Le résultat correspond-il bien au schéma ci-dessus ?

La méthode `toString()` a été programmée dans la classe `Medicament`. Elle permet de retourner, à un instant donné, la représentation sous format texte d'un objet donné.

10- Mettre en commentaire cette méthode dans la classe `Medicament`. (encadrez-la par `/* */`) Tester l'exécution du code.

A l'exécution, une méthode `toString()` générique, déjà programmée dans les API, affiche une valeur en hexadécimal sur 32 bits après le @.

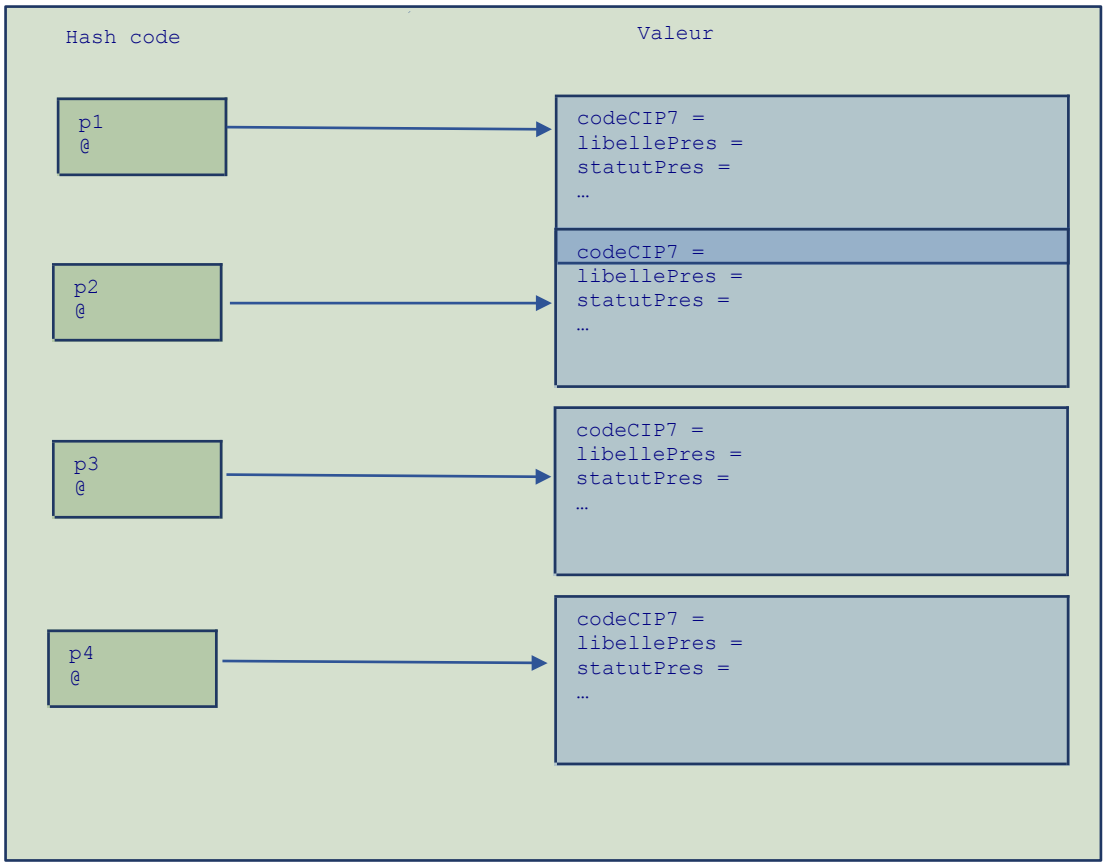
Note : l'identité des objets en Java ne correspond pas directement à une adresse mémoire mais à un *hash code*. Le *hash code* est un nombre entier sur 32 bits qui permet à la JVM de trier l'ensemble des objets et pour y accéder plus rapidement. Ce n'est pas l'adresse mémoire qui est affichée mais le *hash code* de l'objet. A l'exécution, la JVM trie les nouveaux objets en mémoire et utilise ce hash code pour y accéder plus rapidement. A la différence de l'adresse mémoire, on considère que

- deux objets n'ayant pas le même hash code sont forcément différents,
- deux objets ayant le même hash code ne sont pas forcément égaux, il faut donc comparer leurs « vraies » adresses avec le `==` (ou la méthode `equals()`).

11- Comparer les hash codes des objets.

Il n'existe pas de méthode dans le langage Java pour connaître l'adresse physique en mémoire d'un objet. Java étant précompilé pour être portable sur (normalement) tous les appareils Android, cette adresse mémoire ne sera pas définie de la même manière selon le matériel utilisé. Le *garbage collector* peut, d'ailleurs, changer l'adresse des objets lors de l'exécution du programme.

12- Afficher les hash codes des objets de type `Presentation` programmés dans la question 5 et compléter le schéma suivant :



Dans le code source de la classe `Presentation`, un attribut `leMedicament` de type `Medicament` a été programmé. Un médicament peut avoir plusieurs présentations mais une présentation correspond toujours à un et un seul médicament. On va donc ajouter une référence de type `Medicament` dans la présentation (un « pointeur »).

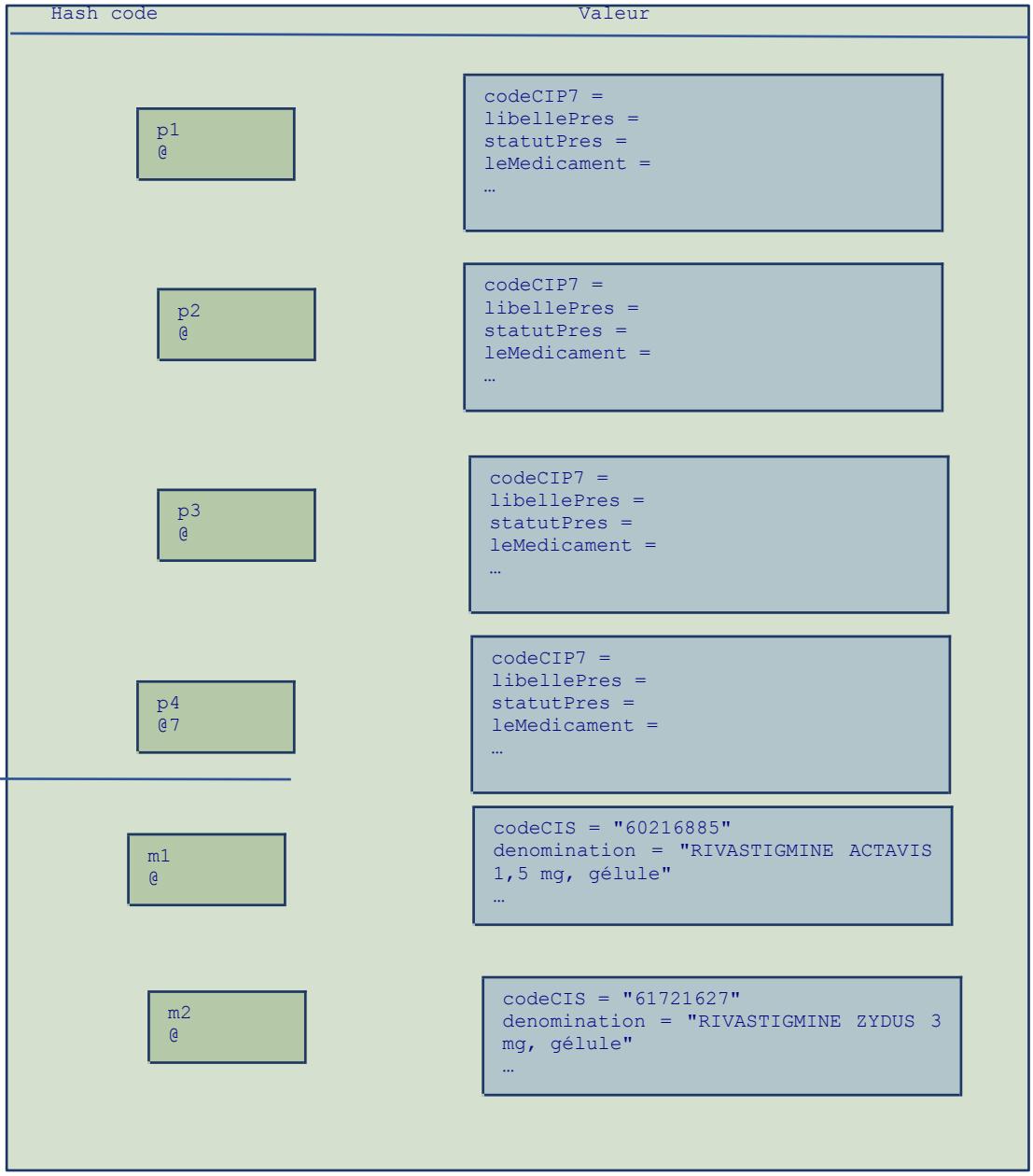
Le code ci-dessous permet de lier un objet de type `Medicament` (référencé par `m1`) à un objet de type `Presentation` (référencé par `p1`) :

```
p1.setLeMedicament(m1)
```

Dans le schéma précédent, il manque cet attribut.

13- Associer les présentations à chaque médicament en respectant les valeurs fournies dans la base de données.

14- Compléter le schéma suivant.



15- Dans la méthode `instancierLesObjets()`, construire les objets suivants, en n'oubliant pas de les lier :

```
Medicament
codeCIS = 68787715
denomination = PARACETAMOL BIOGARAN 1 g, comprimé
formePharma = comprimé
voieAdmin = orale
statutAdminAMM = Autorisation active
typeProcAMM = Procédure nationale
etatCommercialisation = Commercialisée
dateAMM = 2004-03-10 00:03:00.000
statusBDM =
numAutEurop =
titulaire = BIOGARAN
surveillance = false
conditionPrescription =

Presentation
codeCIP7 = 3638186
libellePres = plaquette(s) thermoformée(s) PVC-Aluminium de 8 comprimé(s)
statutPres = Présentation active
etatCommercialisation = Déclaration de commercialisation
dateCommercialisation = 2004-09-07 00:09:00.000
codeCIP13 = 3400936381865
agrementCollectivites = oui
tauxRbt = 65%
prixEuro = 2,10
texteRbt =
codeCIS = 68787715

Dosage
codeCIS = 68787715
codeS = 02202
designElemPharma = comprimé
dosage = 1 g
reference = un comprimé
nature = SA
numSAFT = 1


Substance
codeS = 02202
denomination = PARACÉTAMOL


AvisHASASMR
codeDossierHAS = CT-15118
motifEval = Inscription (CT)
dateAvis = 2016-04-20 00:04:00.000
codeCIS = 68787715
valeurASMR = V
libelleASMR = Cette spécialité est un complément de gamme qui n?apporte pas d?amélioration du
service médical rendu (ASMR V) par rapport aux présentations déjà inscrites.
lienAvisCT = http://www.has-sante.fr/portail/jcms/c_2627669
```

III. Manipuler des objets à l'aide de méthodes

Lorsque l'objet est créé, on peut exploiter les valeurs stockées dans ses attributs et manipuler son comportement par l'intermédiaire de **méthodes**. L'attribut d'une classe est, par convention, **privé** : le programmeur qui utilise la classe `Medicament`, par exemple, n'en a pas connaissance. C'est le principe **d'encapsulation**.

La seule solution pour accéder aux valeurs des attributs est de programmer des **méthodes publiques** dans la classe.

Les **méthodes** qui retournent les valeurs des attributs sont des **accesseurs (*getter*)**. Elles commencent, par convention, par le mot clé *get.*

Les **méthodes** qui modifient la valeur des attributs sont des **mutateurs (*setter*).** Elles commencent, toujours par convention, par le mot clé *set.*

Chaque attribut peut avoir un accesseur et un mutateur. Si un attribut n'a ni accesseur, ni mutateur, il aura un usage uniquement interne à la classe.

Par exemple, dans un médicament, on souhaite pouvoir récupérer son code CIS, mais pas le modifier. Le programmeur va donc écrire la méthode `getCodeCIS()`, mais pas `setCodeCIS()`.

Pour récupérer le code CIS du médicament `m1` créé avec un `new`, on utilisera la notation pointée :

```
String s = m1.getCodeCIS();
```

16- A la fin de la méthode `instancierLesObjets` de la classe `DAO`, modifier les valeurs des attributs des deux présentations créées précédemment (les changements à effectuer sont en italique) :

```
codeCIP7 = 2218950
libellePres = plaquette(s) thermoformée(s) plastique de 14 gélule(s)
statutPres = Présentation active
etatCommercialisation = Déclaration de commercialisation
dateCommercialisation = 2016-04-30 00:04:00.000
codeCIP13 = 3400922189505
agrementCollectivites = non
tauxRbt = 0%
prixEuro = 14.5
texteRbt = plus de prise en charge SS
codeCIS = 60216885

codeCIP7 = 2218967
libellePres = plaquette(s) thermoformée(s) plastique de 56 gélule(s)
statutPres = Présentation active
etatCommercialisation = Déclaration de commercialisation
dateCommercialisation = 2014-01-07 00:01:00.000
codeCIP13 = 3400922189673
agrementCollectivites = non
tauxRbt = 0%
prixEuro = 28.1
texteRbt = plus de prise en charge SS
codeCIS = 60216885
```

17- Vérifier que les présentations ont bien été modifiées en traçant les variables présentation et médicament dans le *log*. Vérifier aussi que la modification a bien été prise en compte dans l'interface graphique. Il faut décommenter la méthode `toString()` de `Presentation`.

Kévin a aussi programmé trois méthodes dans la classe `Presentation` dont voici les signatures :

agrementCollectivites

```
public boolean agrementCollectivites()
```

Retourne un booleen true si l'agrément collectivités est à oui (c'est une chaîne de caractère dans l'objet), false sinon

Returns:
`boolean`

tauxRemboursementEnValeur

```
public double tauxRemboursementEnValeur()
```

Retourne la valeur du taux de remboursement non plus en pourcentage mais en valeur. Exemple : 10% retourne 0.1

Returns:
`double`

texteRemboursementVide

```
public boolean texteRemboursementVide()
```

Retourne vrai si le texte de remboursement est vide, faux sinon

Returns:
`boolean`

18- Tester ces méthodes pour les quatre objets de type `Presentation` précédemment créés.