


Concept. et prog. objet

\1 TD n°4 : cas d'exemple

Détails

Écrit par stéphane Domas

Catégorie : M3105 - Concept. et prog, objet avancée (/index.php/menu-cours-s3/menu-mmi3-test)

 Publication : 21 octobre 2014

 Affichages : 53

1°/ Objectif

- L'objectif de ce TD est de faire travailler les étudiants sur les principes centraux de la POO : héritage, redéfinition, abstraction et polymorphisme.
- Pour cela, on décrit une arborescence de classes ainsi que des méthodes ayant des fonctionnalités précises.
- A partir de la description de ces fonctionnalités, les étudiants doivent déterminer dans quelles classes ces méthodes sont : déclarées abstraites, définies, héritées ou redéfinies.
- La seconde partie du TD est consacrée aux entrées/sorties afin que les étudiants mettent en pratique les principes vus en cours.

2°/ A dada

2.1°/ Les classes d'équidés

- On veut modéliser un troupeau d'équidés et son évolution. Pour cela, on s'intéresse seulement à certains types d'équidés, sachant que l'on doit faire la différence entre les mâles et les femelles. Au final, on aboutit aux classes suivantes :
 - Equidé, qui modélise un équidé en général,
 - Bardots, Chevaux, Anes et Mules, sous-classes de Equidé, qui désignent des **types** d'équidé,
 - Bardot et Bardine, sous classes de Bardots,
 - Etalon et Jument, sous classes de Chevaux,
 - Ane et Anesse, sous classes de Anes,
 - Mulet et Mule, sous classes de Mules.
- Seules les classes "feuilles" (donc Etalon, Jument, Ane, Anesse, Mulet, Mule, Bardot, Bardine) seront instanciées.

2.2°/ Les règles

2.2.1°/ les attributs

- Les règles sur les attributs de ces classes sont les suivantes :
 - Tout équidé a une force, une endurance, une âge et une espérance de vie.
 - En moyenne, un équidé mâle a une force de 6 et une endurance de 4, et l'inverse pour une femelle.
 - Pour les Chevaux, la force moyenne est augmentée de 1 (donc 7 pour un mâle et 5 pour une femelle) et pour les Bardots, elle est diminuée de 1.
 - Pour les Anes et Mules, l'endurance moyenne est augmentée de 1, et pour les Chevaux, elle est diminuée de 1.
 - Pour un individu particulier (donc une instance), la force et l'endurance varie de plus/moins 2, déterminée par tirage au sort.
 - l'espérance de vie moyenne est de 20 ans, sauf pour les Anes et Mules pour lesquels c'est 25 ans.

2.2.2°/ Les méthodes

- Au cours de leur vie, les équidés vont tracter, courir et se reproduire, ce qui est modélisé par les méthodes :
 - Equide rencontre(Equide e)
 - int tracter()
 - int courir()
- Les entiers renvoyées par les deux dernières méthodes sont respectivement le nombre de kilomètres qu'un équidé peut parcourir journalièrement, et le poids qu'il peut tracter (NB : ces valeurs sont totalement arbitraire et il ne faut pas y voir une modélisation cohérente de la réalité).
- Pour chaque méthode, il existe des règles, qui sont les suivantes :
 - Les types Bardots et Mules sont stériles (donc reproduction impossible)
 - Les autres équidés se reproduisent à partir de 3 ans mais les femelles ne sont plus fertiles après 18 ans.

- Si deux équidés fertiles, de sexe opposé ET de même type, se rencontrent, ils produisent forcément un nouvel individu dont le sexe est tiré aléatoirement.
- Si les types ne sont pas les mêmes, rien ne se passe sauf pour une rencontre entre Ane et Jument qui donne un Mulet ou une Mule, ou bien entre un Etalon et une Anesse, qui donne un Bardot ou une Bardine.
- Un équidé tracte 50 fois sa force, sauf les Anes pour lesquels c'est 30.
- Un équidé peut courir 10 fois son endurance, sauf les Bardots, Mules et Anes pour lesquels, c'est 15, avec une sous-exception pour la Bardine avec 12 fois.

2.3°/ Le troupeau

- A côté des classes d'équidé, on définit la classe Troupeau représentant une collection d'équidé.
- Comme pour les TP sur les humains, le troupeau va vieillir et chaque année, un certain nombre de rencontres vont se produire.

3°/ Démarche

- Les étudiants doivent comprendre que les règles permettent de déterminer facilement quel cas est général et lequel est particulier.
- Ensuite, il faut analyser où se trouvent les cas généraux et particuliers dans l'arborescence de classe.
- La situation simple est quand une classe A suit le cas général et que une sous-classe B suit un cas particulier. On a alors une méthode définie dans A et redéfinie dans B.
- Parfois les choses se compliquent, lorsqu'il y a des cas particuliers de cas particuliers ou bien lorsqu'un cas particulier se trouve dans la super classe alors que la sous classe suit le cas général.
- La conclusion est : choisir la solution qui utilise le moins possible de redéfinitions (= moins de code).
- Un deuxième problème vient de l'utilisation ou non de l'abstraction.
- Dans certains cas, la modélisation pousse à déclarer une méthode abstraite dans une classe précise car il n'y a aucun traitement possible pour celle-ci. Cependant, comme l'objectif est d'écrire moins de code dans les sous classes, il est parfois préférable de définir quand même cette méthode au lieu de la mettre abstraite.
- En conclusion, il existe pour les exercices qui suivent plusieurs solutions correctes syntaxiquement mais qui ne se valent pas du point de vue de la compacité du code. Pour certaines questions, plusieurs solutions seront exposées pour mettre en relief cette remarque.

4°/ Exercice sur les constructeurs

4.1°/ Questions

- En fonction des règles sur les attributs, donner le code des constructeurs des classes : Equide, Chevaux, Mules, Jument et Ane.
- L'objectif est d'écrire **le moins de code possible toutes classes confondues**. Il se peut donc qu'un constructeur soit volumineux pour que les autres soient quasi vides.

4.2°/ Solutions

4.2.1°/ Solution naïve basique (= la moins bonne des solutions correctes)

- D'après l'énoncé, la force et l'endurance de chaque instance varie de +/- 2.
- Cette formulation pousse celui qui ne réfléchit pas à mettre ce tirage aléatoire dans chacune des classes feuilles.
- Cette approche est renforcée par le fait qu'il y a une différence entre les stats. des mâles et des femelles. On ne peut donc pas attribuer une force/endurance à un Equidé en général. L'espérance de vie n'est pas non plus commune.
- On aboutit donc à une solution de ce type :

```

1  class Equide {
2      int force;
3      int endurance;
4      int esperanceVie;
5      int age;
6      public Equide() {
7          age = 0;
8      }
9      ...
10 }
11
12 class Chevaux extends Equide {
13     public Chevaux() { super(); }
14     ...
15 }
16
17 class Mules extends Equide {
18     public Mules() { super(); }
19     ...
20 }
21
22 class Jument extends Chevaux {
23     public Jument() {
24         super();
25         Random loto = new Random();
26         force = 5 + (2-loto.nextInt(5)); // moyenne + tirage alea +/- 2
27         endurance = 5 + (2-loto.nextInt(5)); // moyenne + tirage alea +/- 2
28         esperanceVie = 20;
29     }
30     ...
31 }
32
33 class Ane extends Anes {
34     public Ane() {
35         super();
36         Random loto = new Random();
37         force = 6 + (2-loto.nextInt(5)); // moyenne + tirage alea +/- 2
38         endurance = 5 + (2-loto.nextInt(5)); // moyenne + tirage alea +/- 2
39         esperanceVie = 25;
40     }
41     ...
42 }

```

- On voit donc que presque tout est défini dans les classes feuilles, avec à chaque fois le copier/coller du tirage aléatoire.

Remarque : Il existe une solution un peu meilleure si on constate que l'espérance de vie peut être définie dans Equide et simplement changée dans Anes et Mules.

4.2.2°/ Solution après réflexion (= presque la meilleure)

- Si on veut réutiliser un maximum de code, il suffit de voir quels sont les traitements/instructions qui sont communs à un ensemble de classes, puis de mettre ceux-ci dans une super-classe dont hérite cet ensemble.
- Par exemple, si la force et endurance de chaque instance d'une classe feuille varie de +/- 2, cela implique que c'est valable pour n'importe quel équidé. On peut donc mettre ce tirage aléatoire dans Equide.
- Pour autant, il y a toujours le problème de la force et endurance moyenne qui varie selon le genre mais aussi le type. Idem pour l'espérance de vie.

- Une solution acceptable est de fixer une valeur qui convienne au maximum d'équide, puis de modifier leur valeur au fur et à mesure que l'on descend dans l'arborescence des classes et que l'on peut appliquer une règle
- Par exemple, on peut fixer l'espérance de vie d'un équidé à 20 et faire +5 dans les constructeurs de Anes et Mules.
- Cela conduit à la solution suivante :

```

1  class Equide {
2      int force;
3      int endurance;
4      int esperanceVie;
5      int age;
6      static Random loto;
7      public Equide() {
8          loto = new Random();
9          force = 2 - loto.nextInt(5);
10         endurance = 2 - loto.nextInt(5);
11         esperanceVie = 20; // NB: on pourrait aussi choisir 25 et modifier dans Chevaux et Bardot
12         age = 0;
13     }
14     ...
15 }
16
17 class Chevaux extends Equide {
18     public Chevaux() {
19         super();
20         force += 1; // s'applique à tous les Chevaux qu'ils soient males ou femelles
21         endurance -= 1; // idem
22     }
23     ...
24 }
25
26 class Mules extends Equide {
27     public Mules() {
28         super();
29         endurance += 1;
30         esperanceVie += 5;
31     }
32     ...
33 }
34
35 class Jument extends Chevaux {
36     public Jument() {
37         super();
38         force += 4; // moyenne des équidés femelles
39         endurance += 6; // moyenne des équidés femelles
40     }
41     ...
42 }
43
44 class Ane extends Anes {
45     public Ane() {
46         super(); // appel au constructeur de Anes qui fait +1 endurance, et +5 esperance vie
47         force += 6; // moyenne des équidés males.
48         endurance += 4; // moyenne des équidés males.
49     }
50     ...
51 }

```

- Les règles énoncées en section 2° et la question n'empêchent pas de définir des méthodes annexes (du moment que l'on est capable de donner leur code et de les expliquer)
- Dans la solution précédente, il reste pas mal de copier/coller dans les classe feuilles pour définir la force et l'endurance par rapport au sexe.
- En supposant qu'il existe une méthode qui renvoie le sexe, on pourrait encore réduire le code. Cela suppose qu'elle soit déclarée (potentiellement abstraite) dans Equide et définie dans les feuilles, car c'est uniquement dans les feuilles que l'on peut connaître le genre mâle ou femelle.
- Pour mettre en pratique ce principe, il est possible de définir une méthode `getGenre()` dans les classe feuilles. Elle renvoie 0 pour les mâles et 1 pour les femelles. Pour éviter les redéfinitions dans toutes les classes feuilles, on peut même la définir dans Equide avec une valeur par défaut (NB : cela reste dangereux si l'on doit créer de nouvelles classes feuilles et que l'on oublie de redéfinir la méthode).
- Cela conduit à la solution suivante :

```

1  class Equide {
2      static int M=0; static int F=1;
3      int force; int endurance; int esperanceVie; int age;
4      static Random loto;
5      public int getGenre() { return F; } // retourne 1 par défaut
6
7      public Equide() {
8          loto = new Random();
9          if (getGenre() == M) {
10             force = 6; endurance = 4;
11         }
12         else {
13             force = 4; endurance = 6;
14         }
15         force += 2 - loto.nextInt(5); // cette fois c'est +=
16         endurance += 2 - loto.nextInt(5); // cette fois c'est +=
17         esperanceVie = 20;
18         age = 0;
19     }
20     ...
21 }
22
23 class Chevaux extends Equide {
24     public Chevaux() {
25         super();
26         force += 1;
27         endurance -= 1;
28     }
29     ...
30 }
31
32 class Mules extends Equide {
33     public Mules() {
34         super();
35         endurance += 1;
36         esperanceVie += 5;
37     }
38     ...
39 }
40
41 class Jument extends Chevaux {
42     public Jument() {
43         super(); // appel au constructeur de Chevaux qui fait +1 force et -1 endurance
44     }
45     ...
46 }
47
48 class Ane extends Anes {
49     public Ane() {
50         super(); // appel au constructeur de Anes qui fait +1 enduracnce et +5 esperance vie
51     }
52     public int getGenre() { return M; } // redéfinition
53     ...
54 }

```

Remarque : il est possible de se passer de la méthode `getGenre()` grâce au mot clé `instanceof`. Cela conduit à faire un `if` débile qui teste toutes les possibilités de nom de classe pour les mâles (ou femelle). Ce n'est pas lisible, pas vraiment extensible pour N classes feuilles, et surtout pas du tout POO.

5°/ Exercices sur les méthodes.

5.1°/ Questions

- En fonction des règles données ci-dessus, remplir un tableau de 3 colonnes (une pour chaque méthode) et 13 lignes (une pour chaque classe) en précisant dans chaque case si, pour la classe considérée, la méthode est :
 - abstraite,
 - définie,
 - héritée,
 - redéfinie.
- Donner le code de la méthode `rencontre()` pour les classes : Mules, Chevaux, Etalon
- Donner le code de la méthode `tracter()` pour les classes : Jument, Ane, Bardine.
- Donner le code de la méthode `courir()` pour les classes : Equide, Chevaux, Mules
- Il est possible que parmi ces 9 méthodes, certaines soient abstraites et d'autres simplement héritées. Dans ces cas, on se contentera de l'indiquer.
- De nouveau, l'objectif est d'écrire le moins de code possible.

5.2°/ Solutions

5.2.1°/ déclaration

- Pour la méthode `rencontre()` :

ATTENTION ! Le piège basique à éviter est d'interpréter "reproduction impossible" comme le fait que la rencontre n'est qu'un concept pour les Mules et Bardots. Cette mauvaise interprétation conduit à déclarer la méthode `rencontre()` abstraite pour les Mules et Bardots.

L'expression signifie uniquement que la rencontre ne peut pas produire un nouvel équidé, donc que la méthode doit renvoyer `null`. De plus, si l'on a bien suivi/appris les cours, il est strictement impossible pour une classe instanciable d'avoir une méthode abstraite. En effet, la présence d'une méthode abstraite rend la classe abstraite, et de ce fait empêche son instanciation, ce qui contredit l'énoncé.

- La règle basique (sauf exceptions) des rencontres est d'avoir le même type et des sexes opposés.
- Soit `e1` et `e2` deux instances d'équidés. Il faut donc déterminer leur type et leur sexe, soit 4 données.
- Comme l'objectif est d'écrire le moins de code possible, on peut essayer de définir la méthode `rencontre()` dans les classes où l'on connaît plusieurs de ces 4 données, ce qui évite de les trouver.
- Par exemple, si `e1` est en réalité une instance de `Etalon` et que l'on (re)définit la méthode `rencontre()` dans cette classe, il n'y a évidemment pas besoin de déterminer le type et le sexe de `e1`. Il ne reste que celui de `e2`, ce qui peut s'écrire :

```
1 class Etalon extends Chevaux {
2     ...
3     public Equide rencontre(Equide e) {
4         if (e instanceof Jument) {
5             ...
6         }
7         else if (e instanceof Anesse) {
8             ...
9         }
10        else return null;
11    }
12    ...
13 }
```

Remarque : dans cette situation, l'utilisation de `instanceof` simplifie grandement le code et est donc "acceptable" (c'est en fait la hiérarchie de classe qui n'est pas très pertinente !)

- Reste à comptabiliser les cas particuliers. D'après les règles, les Mules et Bardots sont stériles, alors que Chevaux et Anes sont fertiles. Il y a égalité.
 - De ce fait, il existe deux solutions quasi équivalentes en terme de volume de code :
 - définir `rencontre()` dans `Equidé` avec comme code `{ return null; }` et la redéfinir dans `Etalon`, `Jument`, `Ane`, `Anesse` avec du code générant un nouvel individu (cf. exemple ci-dessus).
 - déclarer `rencontre()` abstraite dans `Equidé`, la définir dans `Etalon`, `Jument`, `Ane`, `Anesse` (cf. exemple ci-dessus), et la définir avec le code `{ return null; }` dans `Mules` et `Bardots`.
 - La première solution nécessite de (re)définir la méthode 5 fois, alors que la 2ème nécessite 6 définitions. La différence est vraiment minime en terme de code : juste un `return null` en plus.
 - comme la deuxième solution est plus POO, il faut donc normalement la privilégier.
 - Il existe d'autres solutions correctes mais moins bonnes. Par exemple :
 - la "pire" : une seule définition dans `Equidé` avec un monstrueux switch et plein de `instanceof`
 - la "un peu moins pire" : 2 définitions dans `Chevaux` et `Anes` avec gros switch et `instanceof`, et 2 définitions dans `Mules` et `Bardots` avec `{ return null; }`
 - la "presque bonne" : comme les bonnes solutions mais en faisant `{ return null; }` dans `Mulet`, `Mule`, `Bardot` et `Bardine`.
- Pour la méthode `tracter()` :
- Contrairement à la méthode `rencontre()`, il n'y a pratiquement qu'une seule solution valable : définir `tracter()` dans `Equidé` avec comme code `{ return force*30; }` et la redéfinir dans `Anes` avec comme code `{ return 50*force; }`
 - En effet, il n'y a qu'un cas particulier concernant 2 classes (`Ane` et `Anesse`) parmi les 13. Comme ces deux classes sont les seuls descendants de `Anes`, on peut redéfinir la méthode dans `Anes`.
- Pour la méthode `courir()` :
- L'énoncé est volontairement trompeur. En fait il y a seulement 3 cas particuliers, pour `Etalon`, `Jument` et `Bardine`.
 - La meilleure solution est donc de définir `courir()` dans `Equidé` avec comme code `{ return 15*endurance; }`, la redéfinir dans `Chevaux` avec `{ return 10*endurance; }` et dans `Bardine` avec `{ return 12*endurance; }`.

- En résumé, la meilleure solution de la première question est :

	<code>rencontre()</code>	<code>tracter()</code>	<code>courir()</code>
<code>Equidé</code>	abstraite	définie	définie
<code>Chevaux</code>	héritée	héritée	redéfinie
<code>Anes</code>	héritée	redéfinie	héritée
<code>Mules</code>	définie	héritée	héritée
<code>Bardots</code>	définie	héritée	héritée
<code>Etalon</code>	définie	héritée	héritée
<code>Jument</code>	définie	héritée	héritée
<code>Ane</code>	définie	héritée	héritée
<code>Anesse</code>	définie	héritée	héritée
<code>Mulet</code>	héritée	héritée	héritée
<code>Mule</code>	héritée	héritée	héritée
<code>Bardot</code>	héritée	héritée	héritée
<code>Bardine</code>	héritée	héritée	redéfinie

5.2.2°/ implémentation de méthodes

En reprenant le tableau ci-dessus, il est relativement facile de répondre aux questions d'implémentation.

- Pour la méthode rencontre() :

```
1  class Mules extends Equide {
2      public Equide rencontre(Equide e) { return null; }
3      ...
4  }
5
6  abstract class Chevaux extends Equide {
7      // méthode rencontre() abstraite, hérité de Equide
8      ...
9  }
10
11 class Etalon extends Chevaux {
12     public Equide rencontre(Equide e) {
13         Equide bebe=null;
14         if ((age<3) || (e.age<3) || (e.age>18)) return null; // test sur l'age
15         if (e instanceof Jument) {
16             // int sexe = tirage aléatoire
17             if (sexe==0) bebe = new Etalon(...);
18             else bebe = new Jument(...);
19         }
20         else if (e instanceof Anesse) {
21             // int sexe = tirage aléatoire
22             if (sexe==0) bebe = new Bardot(...);
23             else bebe = new Bardine(...);
24         }
25         return bebe;
26     }
27 }
```

- Pour la méthode tracter() :

```
1  class Jument extends Chevaux {
2      // methode tracter() héritée de Chevaux
3      ...
4  }
5
6  class Ane extends Anes {
7      // méthode tracter() héritée de Anes
8      ...
9  }
10
11 class Bardine extends Bardots {
12     // méthode tracter() héritée de Bardots
13     ...
14 }
```

- Pour la méthode courir() :

```

1 | abstract class Equide {
2 |     public int courir() { return 15*endurance; }
3 |     ...
4 | }
5 |
6 | abstract class Chevaux extends Equide {
7 |     public int courir() { return 10*endurance; }
8 |     ...
9 | }
10 |
11 | class Mules extends Equide {
12 |     // méthode courir() héritée de Equide
13 |     ...
14 | }

```

6°/ Les entrées/sorties

6.1°/ Questions

- Pour sauvegarder l'état d'un équidé au format texte, on définit la méthode : `void saveTxt(PrintWriter pw)`.
 - Cette méthode sauvegarde la valeur des attributs ainsi que le nom de la classe de l'équidé, selon le format : `nom_classe,force,endurance,age`
 - Dans quelle(s) classe(s) de l'arborescence de Equidé doit-on définir et éventuellement redéfinir cette méthode ?
 - Choisissez une classe où est définie (ou redéfinie) cette méthode et donner son code.
-
- Pour éviter les problèmes du format texte, on veut sauvegarder et relire l'état d'un troupeau grâce à un flux objet.
 - On suppose que la classe Troupeau contient un attribut `List<Equide> troupeau`.
 - Donner le code de la méthode `void save(String fileName)` de la classe Troupeau qui permet de sauver l'intégralité du troupeau dans un fichier `fileName`
 - Donner le code de la méthode `void load(String fileName)` de la classe Troupeau qui permet de restaurer l'intégralité du troupeau depuis le fichier `fileName`.

6.2°/ Solutions

6.2.1°/ le format texte

- Pour la méthode `saveTxt()` :
 - On remarque que les attributs sont communs à tous les Equides. L'écriture de ceux-ci peut donc se faire dans la classe Equide.
 - Le problème essentiel vient du fait qu'il faut sauvegarder le nom de la classe d'instance. Si on est dans la méthode `saveTxt()` d'Equide, on ne connaît pas ce nom. Cela implique en apparence de définir `saveTxt()` dans toutes les classes feuilles.
 - Cependant, il est possible de conserver la partie commune dans Equide. En effet, une méthode redéfinie peut appeler la super-méthode.
 - Avec ce principe, quand bien même il faut redéfinir `saveTxt()` dans toutes les classes feuilles, il n'y a pas de copier/coller de code. Par exemple, avec Etalon :

```

1  abstract class Equide {
2      public void saveTxt(PrintWriter pw) {
3          pw.print(force+", "+endurance+", "+age);
4      }
5      ...
6  }
7
8  class Etalon extends Chevaux {
9      public void saveTxt(PrintWriter pw) {
10         pw.print("Etalon,");
11         super.saveTxt(pw);
12         pw.println("");
13     }
14     ...
15 }

```

- Comme dans l'exercice sur les constructeurs, on peut définir des méthodes supplémentaires, par exemple une qui renvoie le nom de classe.
- Cela suppose de définir cette méthode dans toutes les classes feuilles, mais dans ce cas, on ne définit saveTxt() que dans Equide :

```

1  abstract class Equide {
2
3      public abstract String getClassName();
4
5      public void saveTxt(PrintWriter pw) {
6          pw.println(getClassName()+", "+force+", "+endurance+", "+age);
7      }
8      ...
9  }
10
11 class Etalon extends Chevaux {
12     public String getClassName() { return "Etalon"; }
13     ...
14 }

```

6.2.2°/ avec flux objet

- Pour écrire entièrement les membres du troupeau sous forme objet, il faut obligatoirement que les classes d'équidé soient sérialisables.
- Ensuite, vu que List<T> est sérialisable, on peut sauvegarder/lire en une seule ligne tout le troupeau.

```

1  abstract class Equide implements Serialisable {
2      ...
3  }
4
5  class Troupeau {
6
7      private List<Equide> troupe;
8
9      public void save(String fileName) {
10         ObjectOutputStream oos = null;
11         try {
12             oos = new ObjectOutputStream(new FileOutputStream(fileName));
13             oos.writeObject(troupe);
14         }
15         catch(IOException e) { ... }
16     }
17
18     public void load(String fileName) {
19         ObjectInputStream ois = null;
20         try {
21             ois = new ObjectInputStream(new FileInputStream(fileName));
22             troupe = (List<Equide>)ois.readObject();
23         }
24         catch(IOException e) { ... }
25     }
26     ...
27 }

```

7°/ Exercices sur les exceptions

7.1°/ Questions

- Comme dans les TPs, on veut qu'une rencontre impossible produise une exception, nommée MeetingException.
- Le message d'erreur renvoyé par une instance de cette classe peut prendre trois formes :
 1. "Naissance impossible : problème de stérilité", quand une instance de Mulet, Mule, Bardot ou Bardine participe à une rencontre,
 2. "Naissance impossible : les équidés sont de même sexe", lorsque des Chevaux et/ou Anes de même sexe se rencontrent,
 3. "Naissance impossible : problème d'âge", lorsque l'âge ne convient pas.
- Donner le code de la classe MeetingException.
- Que doit-on ajouter au prototype de la méthode rencontre() pour qu'elle propage cette exception au lieu de la traiter elle-même ?
- Donner le code de la méthode rencontre() de : Mules, Anesse.
- En supposant que le programme principal contienne les instructions suivantes :

```

1  public static void main(String[] args) {
2      ...
3      Equide e1 = troupe.get(id1);
4      Equide e2 = troupe.get(id2);
5      Equide bebe = null;
6
7      bebe = e1.rencontre(e2);
8      ...
9  }

```

- Que va-t-il se passer à la compilation et pourquoi ?
- Comment remédier à ce problème ?
- Afin de différencier les cas de stérilité et ceux liés à l'âge, on crée une sous classe à MeetingException, nommée SterilityException
- Cette exception doit être levée lorsque la rencontre concerne un des types stériles.

- Donner le code de cette classe, ainsi que les éventuelles modifications sur la super classe.

7.2°/ solutions

7.2.1°/ La classe MeetingException

- Il existe plusieurs solutions pour créer cette classe, la plus compliquée consistant à définir le message d'erreur lors de l'instanciation.
- Dans ce cas, l'écriture de la classe est simple mais ce sont les méthodes rencontre() qui vont être complexes car elles doivent tester tous les cas d'erreurs. Qui plus est, ces tests sont quasi identiques quelle que soit la classe. Il y a donc beaucoup de copier/coller.
- Comme pour les exercices précédents, il faut donc identifier les traitements communs et ceux qui sont particuliers pour limiter le volume de code. On peut faire les remarques suivantes :
 - les tests sur l'âge sont déjà présents dans toutes les méthodes rencontre(). Il y a donc le cas d'erreur n°3 que l'on peut identifier avant de créer l'objet exception.
 - quand on est dans la classe Mules ou Bardots, c'est forcément le cas d'erreur n°1 qui est pris.
 - il n'y a que pour les chevaux et anes qu'il faut différencier le cas n°1 du n°2 en vérifiant quelles classes sont impliquées dans la rencontre. Vu que cette différenciation se fait de la même façon dans les 4 classes, on peut la mettre dans la méthode getMessage() de MeetingException.
- Au final, on peut parfois créer une exception dont on connaît à l'avance le message, et parfois on le connaît grâce à des tests dans getMessage().
- Cela implique de donner en paramètre au constructeur le type de message (cas n°1, cas n°3, cas n°1 ou 2) ainsi que les 2 objets équidé provoquant l'exception.
- Ensuite, getMessage() utilise le type et éventuellement les 2 objets pour renvoyer le bon message d'erreur.

```

1  class MeetingException extends Exception {
2
3      int typeError; // 0 = à déterminer, 1 = stérilité, 2 = age
4      Equide e1;
5      Equide e2;
6
7      public MeetingException(int typeError, Equide e1, Equide e2) {
8          super("naissance impossible"); // message par défaut
9          this.typeError= typeError; this.e1 = e1; this.e2 = e2;
10     }
11
12     public String getMessage() {
13         String msg="";
14         if (typeError == 1) {
15             msg = "Naissance impossible : problème de stérilité";
16         }
17         else if (typeError == 2) {
18             msg = "Naissance impossible : problème d'age";
19         }
20         else {
21             if ((e1 instanceof Mulet) || (e1 instanceof Mule) || (e2 instanceof Mulet) || (e2 insta
22                 (e1 instanceof Bardot) || (e1 instanceof Bardine) || (e2 instanceof Bardot) || (e2
23                 msg = "Naissance impossible : problème de stérilité";
24             }
25             else if ( (e1.isMale() && e2.isMale()) || (e1.isFemelle() && e2.isFemelle()) ) {
26                 msg = "Naissance impossible : les équidés sont de même sexe";
27             }
28         }
29     }
30     return msg;
31 }

```

```
1 class MeetingException extends Exception {
2
3     public MeetingException(String msg) {
4         super(msg);
5     }
6 }
7
8 class Mules extends Equide {
9     ...
10    public Equide rencontre(Equide e) {
11        try {
12            throw new MeetingException("Naissance impossible : les mules sont stériles");
13        }
14        catch(MeetingException e) {
15            System.out.println(e.getMessage());
16        }
17        return null;
18    }
19    ...
20 }
21
22 class Anesse extends Anes {
23     ...
24    public Equide rencontre(Equide e) {
25        Equide bebe = null;
26        try {
27            if (e instanceof Ane) {
28                if ((age < 3) || (age>15)) throw new MeetingException("Naissance impossible : un équid");
29                else if (e.age < 3) throw new MeetingException("Naissance impossible : un équide a "+"
30                // instanciation ane/anesse
31            }
32            else if (e instanceof Etalon) {
33                // idem que ci-dessus mais naissance bardot/bardine
34            }
35            else {
36                if ((e instanceof Mulet) || (e instanceof Mule)) {
37                    throw new MeetingException("Naissance impossible : les mules sont stériles");
38                }
39                else if ((e instanceof Bardot) || (e instanceof Bardine)) {
40                    throw new MeetingException("Naissance impossible : les bardots sont stériles");
41                }
42                else if ((e instanceof Anesse) || (e instanceof Jument)) {
43                    throw new MeetingException("Naissance impossible : les équidés sont de même sexe");
44                }
45            }
46        }
47        catch(MeetingException e) {
48            System.out.println(e.getMessage());
49        }
50        return bebe;
51    }
52    ...
53 }
```

- La meilleure solution consiste à passer les équidés concernés en paramètre au constructeur. Ainsi, on peut redéfinir la méthode `getMessage()` et créer un message d'erreur en fonction du type de ces équidés. Tous les tests faits dans `rencontre()` sont concentrés dans la classe d'exception, ce qui est non seulement plus lisible mais aussi plus POO.

```

1  class MeetingException extends Exception {
2
3      protected Equide e1;
4      protected Equide e2;
5
6      public MeetingException(Equide e1, Equide e2) {
7          super("Naissance impossible"); // msg par défaut
8          this.e1 = e1;
9          this.e2 = e2;
10     }
11
12     public String getMessage() {
13         String msg = "";
14         if ((e1 instanceof Mulet) || (e1 instanceof Mule) || (e2 instanceof Mulet) || (e2 instanceof Mule)) {
15             msg = "Naissance impossible : les mules sont stériles";
16         }
17         else if ((e1 instanceof Bardot) || (e1 instanceof Bardine) || (e2 instanceof Bardot) || (e2 instanceof Bardine)) {
18             msg = "Naissance impossible : les bardots sont stériles";
19         }
20         else if ( (e1.isMale() && e2.isMale()) || (e1.isFemelle() && e2.isFemelle()) ) {
21             msg = "Naissance impossible : les équidés sont de même sexe";
22         }
23         // etc. avec les tests sur l'age.
24         return msg;
25     }
26 }
27
28 class Mules extends Equide {
29     ...
30     public Equide rencontre(Equide e) {
31         try {
32             throw new MeetingException(this,e);
33         }
34         catch(MeetingException e) {
35             System.out.println(e.getMessage());
36         }
37         return null;
38     }
39     ...
40 }
41
42 class Anesse extends Anes {
43     ...
44     public Equide rencontre(Equide e) {
45         Equide bebe = null;
46         try {
47             if (e instanceof Ane) {
48                 if ((age < 3) || (age>15)) throw new MeetingException(this,e);
49                 else if (e.age < 3) throw new MeetingException(this,e);
50                 // instantiation ane/anesse
51             }
52             else if (e instanceof Etalon) {
53                 // idem que ci-dessus mais naissance bardot/bardine
54             }
55             else {
56                 throw new MeetingException(this,e);

```



```

57     }
58     }
59     catch(MeetingException e) {
60         System.out.println(e.getMessage());
61     }
62     return null;
63 }
64 }

```

3.5.2°/ La propagation explicite

- On voit que le fait de traiter l'erreur directement dans la méthode de rencontre est un peu incohérent : on fait un try/catch autour de throw que la méthode génère elle-même. Dans ce cas, autant afficher directement un message d'erreur sans générer d'exception.
- On remarque également que la méthode renvoie null en cas d'erreur. Si on veut gérer un troupeau comme dans les TPs, il faudrait tester la valeur de retour de rencontre() et si elle n'est pas nulle, insérer le bébé dans le troupeau.
- Or, cette façon de faire est inutile : autant ne rien faire en cas de naissance impossible. Mais pour cela, il faut que ce soit la méthode qui appelle rencontre() qui traite l'exception.
- Pour cela, il faut que rencontre() **propage explicitement** l'exception quand elle a lieu : on ajoute throws MeetingException dans son entête. Par exemple :

```

1  class Mules extends Equide {
2      ...
3      public Equide rencontre(Equide e) throws MeetingException {
4          throw new MeetingException(this,e);
5          return null;
6      }
7      ...
8  }

```

3.5.3°/ La capture

- Le problème du code principal sans try/catch autour de l'appel à rencontre() est le suivant. MeetingException hérite d'Exception, à savoir une exception de type vérifié. Cela implique qu'elle doit à un moment ou un autre être capturée. Si ce n'est pas le cas, le compilateur signale une erreur car main() n'est pas définie comme propageant les MeetingException (NB : ce qui ne devrait JAMAIS être le cas).
- Pour la régler, il suffit de mettre l'appel dans un try/catch

```

1 // NE JAMAIS écrire ça :
2 public static void main(String args[]) throws MeetingExcpetion {
3
4 //MAIS :
5
6 public static void main(String args[]) {
7     ...
8     Equide e1 = troupe.get(id1);
9     Equide e2 = troupe.get(id2);
10    Equide bebe = null;
11    try {
12        bebe = e1.rencontre(e2);
13        troupe.add(bebe);
14    }
15    catch(MeetingExcpetion e) {
16        System.out.println(e.getMessage());
17    }
18    ...
19 }

```

3.5.4°/ La sous classe SterilityException

- Une solution simple consiste à déplacer les tests faits dans `getMessage()` de `MeetingException`, dans sa rédéfinition dans `SterilityException`.
- Cependant, il est possible de simplifier cette méthode en se basant simplement sur un attribut indiquant le type stérile mis en jeu, donné lors de l'instanciation. Par exemple :

```

1  class MeetingException extends Exception {
2
3      protected Equide e1;
4      protected Equide e2;
5
6      public MeetingException(Equide e1, Equide e2) {
7          super("Naissance impossible"); // msg par défaut
8          this.e1 = e1;
9          this.e2 = e2;
10     }
11
12     public String getMessage() {
13         String msg="";
14         // pas de tests sur les types stériles
15         if ( (e1.isMale() && e2.isMale()) || (e1.isFemelle() && e2.isFemelle()) ) {
16             msg = "Naissance impossible : les équidés sont de même sexe";
17         }
18         // etc. avec les tests sur l'age.
19         return msg;
20     }
21 }
22
23 class SterilityException extends MeetingException {
24
25     int typeSterile; // 1 = Mules, 2 = Bardots
26     public SterilityException(int type) {
27         super(null,null);
28         typeSterile = type;
29     }
30
31     public String getMessage() {
32         String msg="";
33         if (typeSterile == 1) {
34             msg = "Naissance impossible : les mules sont stériles";
35         }
36         else if (typeSterile == 2) {
37             msg = "Naissance impossible : les bardots sont stériles";
38         }
39         return msg;
40     }
41 }

```

- La méthode rencontre() de Mules devient :

```

1  public Equide rencontre(Equide) throws MeetingException {
2      throw new SterilityException(1);
3      return null;
4  }

```

- On remarque que le throws de l'entête n'a pas besoin d'être modifié : on utilise le polymorphisme.

