Proxy

Voyons comment écrire des proxys en Java et en Python. Un proxy (on parle aussi de procuration ou mandataire) est un objet qui est fourni à l'utilisateur à la place de l'objet réel qu'il attend. Le proxy sera donc un intermédiaire entre l'utilisateur et l'objet réel. Un proxy ajoute souvent des propriétés non fonctionnelles comme par exemple afficher des traces, compter le nombre d'accès aux opérations, gérer des appels à distance, etc.

Par exemples, la classe Collections de l'API Java définit des proxys sur les lists, map, etc. pour les rendre non modifiables (`unmodifiableList`, `unmodifiableMap`...), pour ajouter un contrôle de type à l'ajout d'un élément (`checkedList`, `checkedMap`, etc.), etc.

Ces proxys sont écrits en réalisant l'interface correspondante [1] (`List`, `Map`...) et définissent les opérations en appelant les opérations de l'objet réel si l'opération est autorisée, ou si l'élément est du type attendu ou en levant une exception sinon.

Exercice 1 : Comprendre unmodifiableList

Intéressons nous à `Collections.unmodifiableList`.

1. Écrire un programme en Java, qui crée une liste d'entiers, y ajoute les éléments 2, 3, 5 et 7 (on pourra utiliser la méthode `addAll` de Collections), affiche la liste, supprime l'élément 5, affiche la liste, utilise la méthode `unmodifiableList` pour avoir un nouvel accès sur la liste et vérifie que par cet accès on ne peut ni ajouter de nouveaux éléments, ni en supprimer.

2. Consulter le code source de la classe Collections de l'API Java 8 dans le fichier `/mnt/n7fs/ens/tp_cregut/src/Collections.java` et retrouver la définition du proxy utilisé par `unmodifiableList`.

Exercice 2 : Proxy et introspection

Définir explicitement une classe peut être lourd pour le programmeur. Aussi, Java propose une classe Proxy qui s'appuie sur l'introspection pour définir des proxys : tous les appels de méthodes seront traités par un objet réalisant l'interface InvocationHandler qui réifie l'appel d'une méthode sous la forme d'une méthode `invoke` qui prend en paramètre la méthode appelée et ses paramètres effectifs. Un paramètre supplémentaire permet d'obtenir le proxy à l'origine de l'appel. La création effective du proxy se fait grâce à la classe Proxy. Elle permet de créer dynamiquement un proxy grâce à sa fabrique statique `newProxyInstance` qui prend en paramètre le chargeur de classe à utiliser (on prend celui qui a permis de charger la classe de l'objet qui nous intéresse, par exemple `List.class.getClassLoader()`), les interfaces que ce proxy réalisera (objet de type `Class`) et l'instance de `InvocationHandler` à utiliser pour traiter les appels de méthodes.

L'intérêt de s'appuyer sur l'introspection est de pouvoir créer des proxy à l'exécution, de ne pas avoir à écrire explicitement le code des classes proxy, de pouvoir réutiliser le même `InvocationHandler` pour plusieurs proxy, etc. L'inconvénient est que le recours à l'introspection induit un surcoût en terme de temps d'exécution.

1. En fait, des classes intermédiaires sont utilisées pour factoriser le code.

Expérimentons le en définissant un proxy protection qui interdit l'appel à certaines méthodes.
1. Définir une réalisation de InvocationHandler appelée `ProtectionHandler` dont le constructeur prend en paramètre un objet et le nom des méthodes à interdire sur cet objet. Lors de l'appel d'une méthode, si son nom est dans les méthodes interdites, une exception `UnsupportedOperationException` sera levée. Sinon, l'appel sera réalisé sur l'objet passé en paramètre du constructeur.
2. À partir de l'exemple précédent, écrire un nouveau programme qui remplace `unmodifiablelist` par la création d'un proxy utilisant `ProtectionHandler`.
3. Utiliser la classe `ProtectionHandler` pour interdire d'appeler les opérations `put` et `clear` sur un tableau associatif (`Map`).
4. Pour aller un plus loin, lire la documentation d'Oracle : Dynamic Proxy Classes.

Exercice 3 : Proxy en Python

Regardons comment faire des proxys en Python.
1. *Introspection.* Python définit les fonctions hasattr(object, name), getattr(object, name), setattr(object, name, value) et delattr(object, name) qui permettent respectivement de savoir si l'objet a un attribut du nom indiqué, de récupérer la valeur de l'attribut, de la modifier et de supprimer l'attribut. Comprendre le programme suivant (dont l'exécution se fait sans erreur).

```
1  class C:
2      pass
3
4  c = C()
5  c.x = 5
6  assert c.x == 5
7  assert hasattr(c, 'x')
8  assert getattr(c, 'x') == 5
9  assert not hasattr(c, "y")
10 setattr(c, 'y', 7)
11 assert hasattr(c, "y")
12 assert getattr(c, 'y') == 7
13 assert c.y == 7
14 assert vars(c) == {'x': 5, 'y': 7}
15 delattr(c, 'x')
16 assert not hasattr(c, 'x')
17 assert vars(c) == {'y': 7}
18 print(dir(c))
```

2. *Introspection et objet.* Sur une classe, les fonctions précédentes s'appuient sur des fonctions de mêmes noms préfixées et suffixées par `__`. Par exemple, si l'attribut cherché n'existe pas sur l'objet, la méthode `__getattr__` est appelée.

Définir une classe qui se comporte comme un proxy protection en remplaçant les appels aux méthodes interdites par la levée d'une exception AttributeError.