Apnée de Programmation numéro 2

Configuration, empaquetage et distribution

1 - Fabriques d'ensembles

Pour commencer, reprenez les ensembles génériques réalisés lors du précédent TP, vous devriez avoir : une interface générique décrivant les méthodes de manipulation d'ensembles, une implémentation sous forme de liste chaînée et une implémentation sous forme de tableaux redimensionnés à la volée. De manière analogue à l'exemple de la pile générique présentée en cours, créez une fabrique abstraite d'ensembles ainsi que deux fabriques concrètes correspondant aux deux implémentations dont vous disposez. Modifiez le programme principal dont vous disposez pour tester ces fabriques.

2 - Configuration globale du jeu

Dans cette partie, nous allons utiliser des fichiers de propriétés pour configurer les implémentations d'ensemble utilisées dans notre jeu. Les fichiers de propriétés sont des fichiers contenant un ensemble de couples (clé, valeur) analogue à une table de hachage. Il ont l'avantage d'être gérés par la classe `java.util.Properties` de la bibliothèque standard. Ils peuvent être écrits selon plusieurs syntaxes décrites dans la documentation en ligne. Pour cette apnée, nous nous contenterons d'une syntaxe simple où chaque ligne est de la forme :

```
clé=valeur
```

et où toute ligne démarrant par un `#` est un commentaire. Les méthodes `load` et `store` de la classe `Properties` permettent respectivement de charger et sauvegarder des fichiers de propriétés écrits en utilisant cette syntaxe.

Dans notre jeu, nous utiliserons des ensembles de petite taille et des ensembles de grande taille. Nous utiliserons des fichiers de propriétés pour déterminer quelle implémentation utiliser pour chaque type d'ensemble. Nous commencerons par écrire un premier fichier de propriétés, nommé `defaut.cfg`, que nous placerons au même endroit que le répertoire contenant les niveaux, et qui contiendra un ensemble de propriétés par défaut :

```
# La valeur pour chacune des propriétés suivantes est : Tableau ou Liste
GrandEnsemble=Tableau
PetitEnsemble=Liste
```

Questions :

- utilisez la classe `Properties` pour charger le fichier `defaut.cfg`

- si un fichier nommé `$HOME/.armoroides` existe, lire ce fichier et utiliser les propriétés qu'il contient à la place des propriétés par défaut. Pour cela :
  - pour des raisons de portabilité, il est préférable d'aller chercher la valeur de la propriété `user.home` fournie par la classe `System` plutôt que d'aller chercher la valeur de la variable d'environnement
  - la classe `Properties` vous permet de créer un objet ayant un autre objet de la classe `Properties` jouant le rôle d'ensemble de valeurs par défaut : toute clé non définie dans l'objet sera cherché dans l'ensemble par défaut. Utilisez ce mécanisme pour permettre au fichier `$HOME/.armoroides` de ne contenir qu'une partie des propriétés ci-dessus
- ajoutez à votre fabrique abstraite une méthode `init` permettant d'initialiser deux attributs statiques `petit` et `grand` contenant respectivement une référence à une fabrique de grands ensembles et une fabrique de petits ensembles (oui, oui, vous avez bien compris, il va falloir tester la valeur de chaînes de caractères pour choisir l'implémentation)
- il est généralement de bon ton de ne pas accéder directement aux attributs d'un objet ou d'une classe afin de masquer le détail de l'implémentation. Ajoutez donc à votre fabrique abstraite deux méthodes pour accéder aux attributs `petit` et `grand`
- modifiez votre programme principal pour n'utiliser que des petits et grands ensembles et testez plusieurs contenus pour vos fichiers de propriétés afin de vérifier qu'ils fonctionnent

3 - Empaquetage

Mettez tout ce qui concerne vos ensembles dans un paquetage nommé `Ensembles` et tester que tout fonctionne. Prenez soin de ne déclarer en public que ce qui doit l'être :

- l'interface `Ensemble` générique
- la classe et les méthodes statiques de votre fabrique abstraite

4 - Distribution

L'environnement de développement java est doté d'un outil permettant de créer des archives exécutables ou non contenant les classes et/ou les fichiers source d'un programme. Ces archives java sont des fichiers, portant généralement l'extension `jar`, exécutables directement par la machine virtuelle java via l'option `-jar`. Pour créer une telle archive avec les programmes que nous avons développés jusqu'alors, vous pouvez utiliser la commande suivante :

```
jar cfe Armoroides.jar ClassePrincipale *.class Ensembles/*.class defaut.cfg Niveaux
```

Cette commande va créer le fichier d'archive `Armoroides.jar` qui contiendra les fichiers `*.class`, `Ensembles/*.class`, `defaut.cfg` ainsi que `Niveaux` et dont la classe principale est `ClassePrincipale` (celle qui contient le `main`). Cette archive est alors exécutable avec la commande suivante :

```
java -jar Armoroides.jar
```

vous devriez alors constater que vos niveaux et vos propriétés ne sont pas trouvés...

Le problème vient du fait qu'une archive `jar` n'est pas similaire à un système de fichiers : les constructeurs des classes `FileInputStream` et `PrintStream` acceptent en paramètre un chemin vers le fichier à ouvrir qui n'a plus de sens dans le cas d'une archive `jar`. Pour résoudre ce problème java est capable de désigner les resources (au sens de données textuelles, images, sons, ...) via une URL qui peut désigner aussi bien un fichier qu'une partie d'une archive. Il reste à localiser ces resources de manière indépendante du système de fichiers. Pour cela java donne accès à la partie de sa mécanique permettant de charger les classes via la classe `ClassLoader`. La méthode `ClassLoader.getSystemClassLoader` permet de récupérer l'objet responsable du chargement des classes d'un programme qui connaît la localisation des différentes ressources associées à ce programme. La série d'appels suivante permet de récupérer la ressource `defaut.cfg` qu'elle soit dans un fichier ou dans une archive `jar` :

```
ClassLoader.getSystemClassLoader().getResourceAsStream("defaut.cfg");
```

la resource est récupérée sous forme d'`InputStream` directement utilisable par un `Scanner` ou par la méthode `load` de la classe Properties. Modifiez votre code pour charger correctement toutes les resources lorsqu'elles se trouvent dans une archive `jar`.

Remarque : une archive `jar` n'est pas un système de fichier, impossible en particulier de parcourir un répertoire de cette archive, vous devez connaître et nommer chaque ressource utilisée.

Bonus - Pour les utilisateurs d'IDE

Les utilisateurs d'IDE avancées telles qu'Eclipse ou Netbeans auront sans doute remarqué que l'accès à des fichiers n'est pas trivial avec ces outils. En effet, l'IDE cache les fichiers source dans un endroit du projet différent de l'endroit où se trouvent les classes et il n'est généralement pas très bon d'aller y faire des modifications manuelles. Pour résoudre cela, vous pouvez ajouter dans les propriétés du projet des chemins à ajouter au `CLASSPATH` pour la recherche de classes :

- sous Eclipse, aller voir dans "Java Build Path"/"Source"/"Link Source"
- sous NetBeans, aller voir dans "Sources"/"Add Folder"

Ceci, combiné à la méthode de chargement présentée ci-dessus, vous ne devriez plus avoir de problème. Notez au passage que vous gagnez la fabrication gratuite de l'archive via une entrée du menu de votre IDE !