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 nommée `FabriqueEnsembleInitiale` 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`. Pour trouver et lire le fichier vous pouvez procéder de manière analogue à ce qui a été vu en apnnée 1 (paragraphe étiqueté **Attention** en fin de sujet) : la méthode `donnees` de la classe `Chargeur` permet de charger un fichier de manière indépendante de sa localisation (système de fichiers ou archive java). La classe `JeuInitial` contient un exemple d'utilisation de cette méthode.
- 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 projet une classe `Configuration` contenant une méthode `lisString`. Cette méthode devra charger l'ensemble de propriétés de la question précédente, si cela n'est pas déjà fait, et renvoyer la valeur de la propriété dont le nom est passé en paramètre. Si cette propriété n'existe pas, cette fonction devra lever une exception de type `NoSuchElementException`.
- ajoutez à la classe `Configuration` des méthodes `lisFloat`, `lisInt` et `lisBoolean` permettant de convertir dans le type idoine la valeur trouvée à l'aide de `lisString`.
- nous allons maintenant associer des objets de notre programme à des valeurs de notre fichier de configuration. Par exemple, nous souhaitons qu'une valeur `Liste` pour la propriété `PetitEnsemble` nous permette de retrouver une fabrique d'ensembles implémentés avec une liste chaînée. Mais nous ne voulons pas que notre classe `Configuration` dépende de nos fabriques car elles seront amenées à changer au cours du prochain TP. Nous allons procéder en deux temps :
    - tout d'abord nous allons ajouter à notre classe `Configuration` une méthode `associe` permettant d'associer à un couple "propriété/valeur" une référence à un objet. Dans notre exemple, nous voulons associer au couple `"PetitEnsemble/Liste"` une référence à une fabrique d'ensembles implémentés par une liste chaînée. Comme, dans la classe `Configuration`, nous ne voulons avoir aucun a priori sur les objets manipulés par le programme, nous y stockerons des références à `Object`.
    - ensuite nous allons ajouter à notre classe une méthode `trouve` qui, étant donné un nom de propriété, va d'abord trouver la valeur associé à ce nom (à l'aide de `lisString`), puis va retrouver la référence à un objet associé au couple "nom/valeur" ainsi obtenu et renvoyer cette référence.
    - la classe `Configuration` ne stockant que des références à `Object`, il va falloir effectuer un cast sur la valeur de retour de `trouve` pour retrouver le bon type associé à notre objet. Si nous voulons simplifier l'interface utilisateur, nous pouvons cacher ce cast dans la méthode `trouve` : il suffit de la rendre générique et de lui faire retourner une référence castée dans le type qu'elle reçoit en paramètre. Et avec l'inférence que fait java il n'est même pas utile de passer ce type explicitement.

  Vous avez sans doute compris que les méthodes `associe` et `trouve` cachent une table de hachage. N'hésitez pas à utiliser la `HashMap` de la bibliothèque standard (dans `java.util`) pour l'implémenter.

3 - Logging

`java` offre des facilités pour ajouter et collecter des messages de suivi de l'exécution. Tout cela se trouve dans la classe `Logger` de `java.util.logging`. Son utilisation est très simple :

- on récupère un objet de la classe `Logger` à partir de son nom (par exemple `"Armoroides.Logger"`) grâce à la méthode `Logger.getLogger`. Si le *logger* en question n'existe pas, il est fabriqué par `getLogger`.
- on peut alors utiliser le *logger* récupéré pour diffuser des messages à l'aide des méthodes `info`, `warning`, `severe`, ... ou plus généralement `log` qui prend un niveau d'importance en paramètre.
- le niveau d'importance à partir duquel les messages sont affichés peut être décidé à l'aide de `setLevel` dans le *logger*, ainsi il est très facile d'activer/desactiver ces messages.

Utilisez tout cela pour ajouter à la classe `Configuration` une méthode `logger` renvoyant un objet de la classe `Logger` dont le niveau d'importance a été réglé à la valeur donnée à la propriété `LogLevel` du fichier de configuration.

4 - Test des parties 1 à 3

Dans l'archive Apnee_Config.zip vous trouverez un programme nommé `Etape3` vous permettant de tester les parties 1 à 3 de cette apnée. Il vient compléter les fichiers que vous avez obtenu suite à l'apnée 1 et au TP1.

5 - Empaquetage

Il est maintenant temps de ranger :

- Les classes et interfaces de l'apnée 1 dans un paquetage nommé `Modèle.LectureNiveau`
- celles du TP1 dans un paquetage nommé `Modèle.Ensembles`
- La configuration et les niveau de test qui vont avec dans un paquetage nommé `Global`
- tous les fichiers `Etape*.java` dans le paquetage par défaut

Prenez soin de ne déclarer en public que ce qui doit l'être et testez que tout fonctionne.

6 - 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 Etape3 Etape[1-3]*.class Global/*.class Modele/LectureNiveau/*.class Modele/Ensembles/*.class Niveaux defaut.cfg
```

Cette commande va créer le fichier d'archive `Armoroides.jar` qui contiendra tous les fichiers `.class` de notre programme ainsi que les autres resources (niveaux et fichier de configuration). Cette archive est alors exécutable avec la commande suivante :

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

Remarque : la méthode `donnees` de la classe chargeur permet de lire le contenu d'une resource, qu'elle soit dans le système de fichiers ou une archive `jar`. Mais 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 !