TP 4

Lecture préliminaire

Dans ce TP nous allons voir un ensemble de démarches à suivre lors de tout "projet". Il s'agit notamment de savoir organiser son espace de travail de développeur et d'utiliser les outils fournis avec JAVA : compilation, tests, génération de documentation ("`javadoc`") et création d'archives ("`jar`").

Organiser l'espace de travail.

1. Il faut créer un répertoire de travail pour chaque nouveau projet. Ce projet s'appellera cette fois `tp4`, vous devez donc créer donc un répertoire `tp4`[1].

2. Dans le cadre d'un projet on peut discerner différents types de ressources : les **codes sources** de vos programmes (les fichiers `.java`), les **bibliothèques** (ou "**librairies**") annexes que vous utilisez pour votre programme, les **programmes de test** qui permettent de valider votre code, la **documentation** – il peut s'agir de la documentation sur le code ou plus généralement de la documentation sur le projet (cahier des charges, comte-rendu de réunion, analyse, etc), **les classes** générées (les fichiers `.class`, càd le bytecode résultant de la compilation des sources), etc. car en fonction du projet il peut y avoir d'autres répertoires.

   Afin de structurer son espace de travail il faut séparer ces différents types d'information dans différents répertoires :

   - `src` pour les sources ,
   - `lib` pour les librairies,
   - `test` pour les fichiers de test
   - `docs` pour la documentation et
   - `classes` pour les classes.

   Dans les TP de cette année, le répertoire `lib` servira peu et la documentation se résumera à la documentation des sources.

   Dans le répertoire du projet (ici `tp4`) vous devez créer les cinq répertoires indiqués ci-dessus.

Paquetages. Les paquetages permettent de structurer en différentes unités les éléments d'un projet. Un paquetage regroupe des classes. Il s'agit ici d'une décomposition logique du projet, de la même manière que l'on peut structurer un espace de fichiers en sous-répertoires pour ranger ses documents.
Il n'y a pas de règle stricte définissant ce qu'il faut mettre dans un paquetage ou quand il faut créer un nouveau paquetage. Ce qu'il faut c'est essayer de conserver une cohérence au sein d'un même paquetage : cohérence fonctionnelle ou d'usage. Par exemple on peut créer un paquetage qui regroupe les éléments principaux d'une application, puis un autre avec les classes annexes et enfin un autre avec celles qui concernent l'interface graphique. Mais ce n'est qu'un exemple, pas une règle.
La définition d'un paquetage en JAVA est implicite. C'est-à-dire qu'il n'y a pas de fichier de description du paquetage. Ce sont aux éléments du paquetage de se déclarer comme y appartenant.
En JAVA à un paquetage correspond nécessairement un répertoire de même nom dans l'espace de fichiers. Dans ce répertoire doivent se trouver les fichiers des classes qui appartiennent à ce paquetage. De plus le code source de ces classes doit commencer par une ligne de la forme "`package` *`nomDuPaquetage`*`;`". Cette ligne doit être la **première ligne de code** effectif du fichier (il peut y avoir des commentaires avant).
La notion de sous-paquetage n'est pas différente de celle de paquetage. Simplement le répertoire contenant le paquetage doit être un sous répertoire de celui du paquetage "principal" et le nom du sous-paquetage est de la forme `nomPaquetage.sousPaquetage` (exemple le paquetage `java.lang`).
Important ! Lorsqu'une classe appartient à un paquetage, le **vrai (et le seul)** nom de la classe n'est pas celui qui apparaît dans la déclaration `public class UneClasse ...` (et qui est aussi le nom du fichier). Mais si cette classe appartient à un paquetage `pack1`, son nom est `pack1.UneClasse`. Par exemple la classe `String` du paquetage `java.lang` s'appelle donc `java.lang.String`.
Ainsi lorsque du code fait référence à cette classe, par exemple pour déclarer un attribut, il faut utiliser le nom "long" `pack1.UneClasse`. Il est possible pour alléger le code de pouvoir utiliser uniquement le nom "court" `UneClasse` à condition d'avoir *importer* le paquetage. Cela se fait en ajoutant dans le code utilisant la référence, la ligne `import pack1.UneClasse` pour n'importer que cette classe du paquetage ou `import pack1.*;` pour importer toutes les classes du paquetage. Cette ligne se place avant la déclaration de la classe.

[1] où vous voulez dans votre espace utilisateur, c'est à vous de gérer cet espace, mais pensez à le structurer pour y retrouver facilement vos différents travaux...

Tests. Nous avons déjà abordé lors du TP 3 l'importance des tests et la nécessité de définir avant l'écritre de chaque méthode les tests qui permettront de valider ce code.
Pour écrire une classe de test vous procéderez ainsi (le fichier `test/CaisseTest.java` fourni est un exemple) :

1. créez un fichier dans le répertoire `test` (le nom est libre mais il est d'usage de l'appeler *`NomDeClasseTestée`*`Test`),
2. ajoutez en entête de ce fichier les lignes :

   ```
   import org.junit.*;
   import static org.junit.Assert.*;
   ```

   Vous devrez certainement ajouter les `import` vers les classes nécessaires pour le test,
3. ajoutez dans le corps de la classe de test les lignes

   ```
   public static junit.framework.Test suite() {
      return new junit.framework.JUnit4TestAdapter(NomDeClasseTest.class);
   }
   ```

   en adaptant *`NomDeClasseTest`* bien évidemment.
4. créez les méthodes de test[2].

   Ces méthodes doivent :

   - être précédées de l'annotation `@Test`
   - avoir pour signature : `public void` *`nomMethode`*`()`

   Elles contiennent le code exécuté pour le test, en particulier les **assertions de test** qui doivent être vérifiées pour que le test soit réussi :

   - `assertTrue(v)` vérifie que la valeur fournie en paramètre vaut `true`,
   - `assertFalse(v)` vérifie que la valeur fournie en paramètre vaut `false`,
   - `assertEquals(ref1, ref2)` vérifie l'égalité de deux valeurs passées en paramètre, en utilisant `equals()` pour les objets,
   - `assertNull(ref)` (respectivement `assertNotNull(ref)`) vérifie que lé référence fournie en paramètre est (respectivement n'est pas) `null`
   - `assertSame(ref1, ref2)` vérifie en utilisant `==` que les deux références fournies en paramètre correspondent au même objet (`assertNotSame(ref1, ref2)` existe également)
   - `fail()` échoue toujours

   Le travail le plus compliqué (et c'est une tâche réellement difficile) est de bien construire les tests réalisés afin qu'ils permettent de s'assurer avec un maximum de certitude du bon fonctionement de la méthode.

Compilation et exécution des tests.
Le fichier `test.jar` fourni vous permet de compiler et d'exécuter vos tests sous réserve que vous ayez respecté la structure des répertoires indiqués, en particulier les répertoires `classes` et `test`.

Il faut commencer par compiler la classe dont vous testez les fonctionnalités, en plaçant le fichier compilé (`.class`) dans le dossier `classes`.
Ensuite, la **compilation** de la classe de test se fait à partir de la racine du projet à l'aide de la commande[3] :

```
javac -classpath test.jar test/NomDeClasseTest.java
```

Attention : une des erreurs possibles est d'oublier d'importer les classes utilisées dans le code de la classe de test, en particulier la classe que vous testez.

L'**exécution** du test se fait à partir de la racine du projet à l'aide de la commande :

```
java -jar test.jar NomDeClasseTest
```

[2] Le nom de ces méthodes est libre, mais ce nom reprend souvent pour facilité le nom de la méthode testée ou ce nom suivi de `Test`.
[3] Si vous utilisez la version 1.7 de java, récupérez l'archive `test-1.7.jar` sur le portail, zone "Documents".