TP 5

Lecture préparatoire

Organiser l'espace de travail.

Pour chaque nouveau projet (ou TP pour vous) il faut créer un dossier de travail (appelé «*dossier racine du projet*»). Par exemple, pour cette fois, vous devez créer un dossier `tp5`[1].
Dans le cadre d'un projet on peut discerner différents types de ressources : les **codes sources** de vos programmes (les fichiers `.java`), les **programmes de test** qui permettent de valider votre code, la **documentation** sur le code, **les classes** générées (les fichiers `.class` par la compilation), etc. (en fonction du projet il pourrait y avoir d'autres dossiers).
Afin de structurer son espace de travail il faut séparer ces différents types d'information dans différents dossiers :

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

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-dossiers pour ranger ses documents.
Il n'y a pas de règle formelle 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.

Définition d'un paquetage. La définition d'un paquetage en Java est **implicite**. C'est-à-dire qu'il n'y a pas de fichier de définition d'un paquetage. Ce sont aux éléments du paquetage de se déclarer comme y appartenant.
Pour définir un paquetage de nom ***`pack1`***, il faut :

1. définir (dans le dossier `src/`) un dossier de même nom que le paquetage (donc `pack1/` ici) ;
2. placer les fichiers (`.java`) de définition des classes appartenant à ce paquetage dans ce dossier ;
3. ajouter dans chaque fichier `.java` la ligne `package` ***`pack1`***`;`.
   Cette ligne doit être la **première ligne de code** effectif du fichier (il peut y avoir des commentaires avant).

Un « sous-paquetage » se traduit par un sous-dossier placé dans le dossier « principale ». Par exemple un paquetage `pack1.subpack`, se traduit par un dossier `subpack/` placé dans le dossier `pack1/`. Le reste est inchangé.

Nom de classe et importation. Lorsqu'une classe appartient à un paquetage, le « **vrai** » (et le seul) nom de la classe est construit en concaténant le nom du paquetage qui la contient avec le nom de la classe qui apparaît dans la déclaration «`public class SomeClass ...`» (le « nom court » de la classe).
Donc, si cette classe appartient à un paquetage `pack1`, son nom est « `pack1.SomeClass` ». Par exemple, la classe `String` du paquetage `java.lang` s'appelle en réalité « `java.lang.String` ».
On doit donc a priori utiliser le nom `pack1.SomeClass` pour définir un attribut, une variable ou un paramètre de ce type. Pour alléger l'écriture de code, il est possible d'**importer** un paquetage. Il suffit d'ajouter dans le code de la classe définissant cette référence, la ligne :
`import pack1.SomeClass`
Cette ligne se place dans le fichier avant la déclaration de la classe «`public class OtherClass ...`» (mais après l'annonce du paquetage).
Grâce à cette importation il est possible dans le code de la classe `OtherClass` d'utiliser simplement le nom court `SomeClass`.
La syntaxe `import pack1.*;` permet d'importer toutes les classes du paquetage `pack1`.
Dans un paquetage il n'est pas nécessaire d'importer les classes du même paquetage. Cela est implicite.
NB : Le paquetage principal de Java s'appelle `java.lang`, il est toujours importé, par défaut[2].

[1] Dans le dossier correspondant à votre dépôt local git.
[2] Ce qui explique que l'on ait pu utiliser `String` sans souci au lieu `java.lang.String`.

Tests.

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

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

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

   Vous devrez peut-être également ajouter d'autres `import` vers les classes nécessaires pour les tests, en particulier la classe testée.
3. ajoutez dans le corps de la classe de test les lignes

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

   en adaptant bien sûr *`SomeClassTest`* avec votre nom de classe de test.
4. créez les méthodes de test[3].

   Ces méthodes doivent être précédées de l'annotation `@Test` et nécessairement avoir pour signature :

   `public void` *`testingMethod`*`()`

   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)`/`assertFalse(v)` vérifie que la valeur `v` fournie en paramètre vaut `true`/`false`,
   - `assertEquals(expected, actual)` vérifie l'égalité de deux valeurs passées en paramètre, en utilisant `equals()` pour les objets. La première valeur «`expected`» est la valeur de référence, c.-à-d. celle que l'on doit obtenir, la seconde «`actual`» correspond à la valeur testée, c.-à-d. la valeur que l'on a caculée et dont on veut vérifier la correction.

     Lorsque les valeurs `expected` et `actual` sont des nombres flottants, compte-tenu de l'imprécision des calculs sur ces nombres, il faut rajouter un troisième paramètre, `epsilon`, qui représente l'intervalle de précision tolérée pour l'égalité. Dans ce cas, le test `assertEquals(expected, actual, epsilon)` sera considéré comme un succès si |`expected - actual`| `<` `epsilon`.
   - `assertNull(ref)`/ `assertNotNull(ref)` vérifie que lé référence fournie en paramètre est `null`/ n'est pas `null`
   - `assertSame(expected, actual)` vérifie en utilisant `==` que les deux références fournies en paramètre correspondent au même objet (`assertNotSame(expected, actual)` existe également). La valeur «`expected`» est la valeur de référence, «`actual`» est la valeur testée.
   - `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.

La zone Documents du portail contient un document avec des informations complémentaires sur les tests.

Compilation et exécution des tests.

> ***Rappel :*** *pour réaliser ces tests unitaires nous utilisons la bibliothèque `JUnit4` qui est très utilisée pour les tests unitaires en java. Les tests tels que vous les apprenez sont donc bien des tests `JUnit4`. Cependant le fichier `test4poo.jar` en est une adaptation locale pour vous permettre de disposer de la fenêtre graphique des résultats.*

Le fichier `test4poo.jar` fourni vous permet de compiler et d'exécuter vos tests sous réserve que vous ayez respecté la structure des dossiers indiqués, en particulier les noms des dossiers `classes` et `test`.

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

`javac -classpath test4poo.jar test/`*`NomDeClasseTest`*`.java`

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

`java -jar test4poo.jar` *`NomDeClasseTest`*

[3] Le nom de ces méthodes est libre, mais ce nom doit représenter ce qui est testé. Il n'est pas rare d'avoir des noms de méthode de test plutôt longs.