TP

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, génération de documentation ("javadoc") et création d'archives ("jar").
Les fichiers mentionnés sont disponibles dans le semainier sur le portail.

Exercice 1 : Placez vous dans une fenêtre de commandes et exécutez les étapes décrites ci-dessous :

Q 1 . Organisez l'"espace de travail".

Q 1.1. Il faut créer un répertoire de travail pour chaque nouveau projet. Ce projet s'appellera tp4, créez donc un répertoire `tp4`[1].

Q 1.2. Dans le cadre d'un projet on peut discerner différents types de données :

les sources les codes source de vos programmes (les fichiers `.java`),

les librairies les bibliothèques annexes que vous utilisez pour votre programme,

la documentation il peut s'agir de la documentation sur le code ou plus généralement de la documentation sur leprojet (cahier des charges, comte-rendu de réunion, analyse, etc),

les classes les classes générées (les fichiers `.class`, càd le bytecode définissant les classes),

etc. 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,

docs pour la documentation,

classes pour les classes.

Dans nos travaux, le répertoire `lib` servira peu et la documentation se résumera à la documentation des sources.

Créez pour le projet tp4 les quatre répertoires indiqués.

Q 2 . 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-répertoire 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** 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 enb fait `java.lang.String`.

[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...

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.

Dans cet exercice nous travaillerons avec les fichiers du projet robot . Ce projet impliquait trois classes : `Robot.java`, `Caisse.java`, `TapisRoulant.java` (cf. portail).

Pour illustrer l'utilisation des paquetages, deux paquetages seront créés. Le premier se nommera `exemple` et il contiendra un sous-paquetage `exemple.util`.

Q 2.1. Dans le répertoire `src`, créez un répertoire pour le paquetage `exemple`. Celui-ci accueillera toutes les classes du paquetage `exemple`. Dans ce répertoire créez un répertoire `util` qui contiendra les classes du répertoire `exemple.util`. Rappelons qu'à un paquetage doit correspondre un répertoire de même nom.

Q 2.2. Le paquetage `exemple` contiendra une seule classe : la classe `Robot`. Copiez le fichier `Robot.java` du premier TP dans ce répertoire (il se trouve sur le portail avec les autres fichiers nécessaires à ce TP).
Pour que la classe appartienne au paquetage, il faut le déclarer en tête du fichier source. Editez le fichier (avec l'éditeur de votre choix, `emacs` par exemple) et ajoutez la ligne "`package exemple;`" en tête du fichier.

Q 2.3. Faites ce qu'il faut pour définir les classes `Caisse` et `TapisRoulant` comme classes du paquetage `exemple.util`.

Q 2.4. Il faut faire une nouvelle modification dans la classe `exemple.Robot`. Cette classe utilise les classes `Caisse` et `TapisRoulant` du paquetage `exemple.util`. Il est donc nécessaire d'**importer** ces classes pour pouvoir les utiliser.
Ajoutez la déclaration "`import exemple.util.*;`" en tête du fichier, après la déclaration du paquetage et avant la déclaration de la classe.
Le paquetage `java.lang` de JAVA est toujours importé par défaut. C'est pourquoi pour utiliser la classe `java.lang.String`, par exemple, on peut se contenter du nom court `String`.

Q 3 . Compilation. L'introduction (indispensable) de la notion de paquetage va un peu modifier la procédure de compilation et d'exécution de code.

Que ce soit pour la compilation ou l'exécution, il est en effet nécessaire de préciser où se trouvent les paquetages utilisés.

JAVA utilise pour cela un variable d'environnement appelée `CLASSPATH`. Cette variable, un peu à l'image de la variable d'environnement `PATH` utilisée par les shells, regroupe une liste de chemins (répertoires) dans votre espace de travail. Dans cette liste, les répertoires sont indiqués par des chemins absolus ou relatifs, ils sont séparés par "`:`" sous linux/unix et par "`;`" sous Windows.

Cette liste désigne les chemins de votre espace de fichiers dans lesquels JAVA doit chercher les classes nécessaires à la compilation ou à l'exécution. Ces chemins doivent désigner les répertoires **racines** des paquetages utilisés. Ainsi si vous devez utiliser une classe du paquetage pack1, celle-ci se trouve nécessairement dans un répertoire `pack1`. C'est le répertoire racine de `pack1` qui doit être présent dans la variable `CLASSPATH` (de manière absolue ou relative).

Ensuite pour compiler une classe qui se trouve dans un paquetage, il faut tenir compte du fait qu'à ce paquetage correspond un répertoire. Le nom du fichier à compiler reprend donc ce nom de répertoire. Le fichier à compiler n'est donc pas `UneClasse.java` mais `pack1/UneClasse.java`.

Q 3.1. Visualisez votre variable `CLASSPATH` (commande `echo $CLASSPATH`). Si la variable n'est pas définie[2], rien de s'affiche. JAVA considèrera alors qu'elle vaut simplement "`.`" (le répertoire courant). Si elle est définie, vérifiez que ce même "`.`" est présent dans la définition.

Q 3.2. Nous avons en début de TP créé un dossier `classes` pour y ranger les fichiers compilés. Pour préciser au compilateur qu'il doit déposer le résultat de la compilation dans ce répertoire, il faut utiliser l'option `-d` de la commande (`javac`). Cette option permet de préciser le répertoire ***d**estination* de la compilation.
Placez vous dans le répertoire `src`.
Exécutez la commande : "`javac exemple/Robot.java -d ../classes`".
La classe `exemple.Robot` est alors compilée ainsi que les classes dont elle dépend et le résultat de la compilation est rangée dans le répertoire `classes` que vous aviez créé.
Lors de la compilation, c'est un fichier qui est compilé, d'où l'usage du "`/`", séparateur de fichier.

[2] c'est très probablement le cas.

Q 3.3. Examinez le contenu du répertoire `classes`. On y retrouve la structure des paquetages contenant les fichiers compilés (extension `.class`).

Q 4 . Génération de la documentation.

L'utilitaire `javadoc` permet de générer au format html une documentation telle que celle vue à l'occasion du TP 3. La documentation générée dépend d'informations contenues dans le fichier source. Il s'agit de commentaires compris entre les délimiteurs `/**` et `*/` et placés avant les éléments à commenter : classe, attributs, constructeurs, méthodes. Par défaut seuls les éléments publics apparaissent dans la documentation générée. Des éléments particuliers, appelés *tag*, peuvent être précisés dans cette documentation, par exemple : `@param` pour décrire un paramètre, `@return` pour préciser une valeur de retour d'une méthode, etc. Vous pouvez vous inspirer des codes fournis.

Nous rangerons cette documentation dans le répertoire `docs`.

Q 4.1. Placez vous dans le répertoire `src`.

Q 4.2. Exécutez la commande : "`javadoc exemple exemple.util -d ../docs`" ou "`javadoc -d ../docs -subpackages exemple`" pour générer la documentation du paquetage `exemple` et de tous ses sous-paquetages.

Q 4.3. Allez dans le répertoire `docs`. Vous pouvez y retrouver la structure des paquetages. Ouvrez le fichier `index.html` dans un navigateur, vous accédez alors à la documentation du projet.

Q 4.4. Complétez la javadoc de la classe `Caisse.java` car elle est incomplète.

Regénérez ensuite la documentation et consultez les modifications que vous avez apportées.

Q 4.5. Tapez la commande "`javadoc`" : une rapide description des options possibles est affichée... Un exemple : l'option `-author` qui permet de prendre en compte les sections `@author`.

Q 5 . Exécution.

Q 5.1. Il est tout d'abord nécessaire de disposer d'une méthode statique `main`. Insérez le contenu du fichier `unMain.txt` (fourni sur le portail) dans le fichier `Robot.java`

Q 5.2. Recompilez.

Q 5.3. Placez vous dans le répertoire `classes` et exécutez la commande : "`java exemple.Robot`".

Lors de l'exécution c'est une classe qui est utilisée, d'où le "`.`" séparateur des noms de paquetages. Notez la différence avec la commande de compilation.

Q 5.4. Lors de l'exécution du `main`, les classes `exemple.util.Caisse` et `exemple.util.TapisRoulant` sont également utilisées. Leurs définitions seront donc chargées par la JVM. Ce chargement sera (a été) possible si les définitions de ces classes étaient accessibles depuis les chemins définis par `CLASSPATH`. C'était le cas ici si `CLASSPATH` contient le chemin "`.`", car `exemple/util/Caisse.class` (par exemple) est accessible à partir de "`.`" qui est le répertoire courant donc `classes` ici.

Pour vérifier cela, placez vous dans le répertoire du projet : `tp4` et réessayez "`java exemple.Robot`". Un message d'erreur apparaît "`NoClassDefFoundError`" qui, on le comprend, signifie qu'aucune définition de classe n'a été trouvée. En effet depuis le répertoire `tp4`, le fichier `exemple/Robot.class` c'est pas accessible si votre `CLASSPATH` vaut "`.`" (qui est sa valeur par défaut rappelons le).

Il est possible de définir une valeur spécifique de `CLASSPATH` pour une exécution donnée. Cela se fait grâce à l'option `-classpath` de la commande `java` (cette option existe aussi pour `javac` et a le même effet).

Essayez : "`java -classpath classes exemple.Robot`"

Cette fois plus de problème car, les fichiers nécessaires sont accessibles depuis le répertoire `classes` présent dans `CLASSPATH` précisé pour cette exéction.

Q 6 . Gestion d'archives. Un programme JAVA est un ensemble de classes et ne consiste donc pas en un seul fichier comme c'est le cas d'un exécutable. Pour permettre une diffusion plus facile, la notion d'archive java existe. L'utilitaire qui permet de les créer est `jar` (pour *Java ARchive*) et opère sensiblement comme la commande système `tar`.

Q 6.1. Exécutez la commande "`jar`". Vous voyez apparaître un rapide descriptif de la commande et de ses options.

Q 6.2. Création. Placez vous dans le répertoire `classes`. Exécutez la commande : `jar cvf ../appli.jar exemple`. Vous avez alors **C**réé dans le répertoire `tp4` un fichier `appli.jar` qui contient, compressés, tous les fichiers de l'arborescence dont la racine est le répertoire `exemple`.

Q 6.3. Consultation. Placez vous dans le répertoire `tp4`. Exécutez la commande : `jar tvf appli.jar`. Vous visualisez le contenu de l'archive `appli.jar`.

Q 6.4. Utilisation Dans le répertoire `tp4`, créez un répertoire `tmp`. Copiez y le fichier `appli.jar`. Placez vous dans le répertoire `tmp`. Exécutez la commande : "`java exemple.Robot`", vous obtenez un message d'erreur, le même que tout à l'heure, car la classe indiquée n'est pas trouvée par `java` dans les localisations définies par la variable `CLASSPATH`.
Exécutez la commande : "`java -classpath appli.jar exemple.Robot`", cette fois pas de problème, l'archive a été déclarée comme un emplacement où chercher les classes lors de l'exécution.

Ceci complète la définition donnée pour `CLASSPATH` qui peut dont contenir la liste des emplacements où trouver le code des classes à utiliser, ces emplacements étant ou des répertoires ou des archives.

Q 6.5. Extraction. Toujours depuis le répertoire `tmp`, exécutez la commande "`jar xvf appli.jar`". Consultez le contenu du répertoire `tmp` : vous y trouvez maintenant les fichiers classes du projet `robot`.

L'option `x` permet en effet d'e**X**traire les fichiers de l'archive.

Q 6.6. jar exécutable. Un point intéressant est la possiblité de faire des `jar` exécutables, c'est-à-dire qui définissent automatiquement le `main` à exécuter. Cela facilite le lancement d'une application JAVA. Pour cela il faut rajouter des informations à l'archive, ces informations sont définies dans un fichier particulier appelé `manifest`.

Placez vous dans le répertoire `tp4`. Copiez y le fichier `manifest-ex` (fourni sur le portail) qui est un exemple de fichier de définition de *manifest*. Jetez y un œil : il définit la classe `exemple.Robot` comme classe principale d'une archive.

Placez vous dans le répertoire `classes`, exécutez la commande "`jar cvfm ../appli.jar ../manifest-ex exemple`".

Vous avez créé la même archive que précédemment mais en y ajoutant les informations contenues dans le fichier **M**anifeste mentionné. Ces informations sont stockées dans le fichier `META-INF/MANIFEST.MF` de l'archive.

Allez dans le répertoire `tp4` et exécutez la commande : "`java -jar appli.jar`", le manifeste de l'archive est automatiquement utilisé pour déterminer le "main" à exécuter. `CLASSPATH` intègre automatiquement les fichiers de l'archive, il est donc inutile de préciser quoi que ce soit ici.

Q 7. Rendre un projet. Lorsque vous devrez rendre un projet et/ou tp, il faudra le rendre sous la forme d'un "jar exécutable" contenant le source et la documentation. Il faut donc à l'opération précédente de création d'archive, ajouter les répertoires `src` et `docs`. Pour conserver l'arborescence, il convient de se placer "au-dessus" de ces répertoires et donc de **C**hanger de répertoire par l'option `-C`

Depuis le répertoire `classes` vous devez exécuter la commande :

```
jar cvfm ../appli.jar ../manifest-ex exemple -C ..  src -C ..  docs
```

(`-C ..  src` signifie "**C**hanger vers le répertoire `..` et ajouter `src` et son contenu à l'archive")

ou depuis le répertoire racine de `classes` :

```
jar cvfm appli.jar manifest-ex src docs -C classes exemple
```