TP 4

Manipulations

Les manipulations présentées ici ne sont réalisables que si, comme demandé, vous avez lu attentivement le document préliminaire.

Les fichiers mentionnés sont disponibles dans le semainier sur le portail.
Placez vous dans une fenêtre de commandes ("terminal") et exécutez (en les comprenant) les différentes étapes décrites ci-dessous. Dans la suite du semestre vous devrez constamment effectuer ces manipulations.

Exercice 1 : Dans cet exercice nous travaillerons avec les fichiers du projet `robot`. Ce projet impliquait trois classes : `Robot.java`, `Box.java`, `ConveyerBelt.java` à récupérer sur le portail et à placer dans le dossier `src`.

Q 1 . Mise en œuvre des paquetages Pour illustrer l'utilisation des paquetages, deux paquetages seront créés. Le premier se nommera `example` et il contiendra un sous-paquetage `example.util`.

Q 1.1. Dans le dossier `src`, créez un dossier pour le paquetage `example`. Celui-ci accueillera toutes les classes du paquetage `example`. Dans ce dossier créez un dossier `util` qui contiendra les classes du paquetage`example.util`. Rappelons qu'à un paquetage doit correspondre un dossier de même nom.

Q 1.2. Le paquetage `example` contiendra une seule classe : la classe `Robot`. Copiez le fichier `Robot.java` du premier TP dans ce dossier (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 example) et ajoutez la ligne "`package example;`" en tête du fichier.

Q 1.3. Faites ce qu'il faut pour définir les classes `Box` et `ConveyerBelt` comme classes du paquetage `example.util`.

Q 1.4. Il faut faire une nouvelle modification dans la classe `example.Robot`. Cette classe utilise les classes `Box` et `ConveyerBelt` du paquetage `example.util`. Il est donc nécessaire d'**importer** ces classes pour pouvoir les utiliser.
Ajoutez la déclaration "`import example.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 2 . 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`[1].

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 dossiers **racines** des paquetages utilisés. Ainsi si vous devez utiliser une classe du paquetage `pack1`, celle-ci se trouve nécessairement dans un dossier `pack1`. C'est le dossier 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 dossier. Le nom du fichier à compiler reprend donc ce nom de dossier. Le fichier à compiler n'est donc pas `UneClasse.java` mais `pack1/UneClasse.java`.

Q 2.1. 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 dossier courant). Si elle est définie, vérifiez que ce même "`.`" est présent dans la définition.

[1] Cette variable, un peu à l'image de la variable d'environnement `PATH` utilisée par les shells, regroupe une liste de chemins (dossiers) dans votre espace de travail. Dans cette liste, les dossiers sont indiqués par des chemins absolus ou relatifs, ils sont séparés par "`:`" sous linux/unix et par "`;`" sous Windows.
[2] c'est très probablement le cas.

Q 2.2. 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 dossier, il faut utiliser l'option `-d` de la commande (`javac`). Cette option permet de préciser le dossier ***d**estination* de la compilation.

Placez vous dans le dossier `src`.

Exécutez la commande : "`javac example/Robot.java -d ../classes`".

La classe `example.Robot` est alors compilée ainsi que les classes dont elle dépend et le résultat de la compilation est rangée dans le dossier `classes` que vous aviez créé.

Lors de la compilation, c'est un fichier qui est compilé, d'où l'usage du "`/`", séparateur de fichier.

Q 2.3. Examinez le contenu du dossier `classes`. On y retrouve la structure des paquetages contenant les fichiers compilés (extension `.class`).

Q 3 . 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[3]. 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 dossier `docs`. Rappelons que la javadoc d'une méthde doit être écrite avant le codage de celle-ci !

Q 3.1. Placez vous dans le dossier `src`.

Q 3.2. Exécutez la commande : "`javadoc example example.util -d ../docs`" ou "`javadoc -d ../docs -subpackages example`" pour générer la documentation du paquetage `example` et de tous ses sous-paquetages.

Q 3.3. Allez dans le dossier `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 3.4. Complétez la javadoc de la classe `Box.java` car elle est incomplète.

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

Q 3.5. 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 4 . Tests.

Nous allons tester les tests de la classe `Box` définis dans la classe de test `BoxTest`.

On rappelle que dans une démarche normale ces tests doivent être écrits avant l'écriture du code de la méthode, juste après en avoir écrit la javadoc.

Q 4.1. Ouvrez le fichier `test/BoxTest.java` et étudiez les tests proposés.

Q 4.2. La classe `Box` a été placée dans un paquetage, enlevez les commentaires devant la déclaration d'importation cette classe.

Q 4.3. Compilez la classe `Box` et la classe de test `BoxTest.java` (en vous plaçant à la racine du projet et en supposant l'existence du dossier `classes`) :

```
javac -classpath test-1.7.jar test/BoxTest.java
```

Q 4.4. Exécutez le test, depuis la racine du projet :

```
java -jar test-1.7.jar BoxTest
```

Tous les tests doivent être passés avec succès. C'est le cas si la barre est verte.

Q 4.5. Créez une classe de test pour tester les méthodes de la classe `Robot` :

1. pour `take`, testez les deux cas où le robot porte déjà ou non une caisse avant l'appel de la méthode,
2. pour `carryBox`, testez les deux cas de figure,
3. pour `putOn` testez que si la caisse a été posée alors, ensuite, le robot ne la porte plus. Dans les différents cas où la caisse ne peut être posée on vérifie qu'à l'issue de la méthode la caisse portée est la même qu'avant [4].

Dans chacune des méthodes il faudra créer les objets utiles au test : le robot, les caisses et les tapis roulants appropriés.

[3] Il faut ajouter l'option `-private` lors de l'exécution de la commande pour générer la documentation des éléments privés

[4] On peut constater que la méthode `putOn` est "mal écrite" car elle devrait utiliser des exceptions et non renovyer une chaîne de caractères... Vous pouvez tester cette solution.

Q 4.6. Compilez et exécutez vos tests.

Q 5 . Exécution du programme.

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 coprs de la définition de la classe `Robot`.

Q 5.2. Recompilez.

Q 5.3. Placez vous dans le dossier `classes` et exécutez la commande : "`java example.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 `example.util.Box` et `example.util.ConveyerBelt` 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 `example/util/Box.class` (par exemple) est accessible à partir de "`.`" qui est le dossier courant donc `classes` ici.

Pour vérifier cela, placez vous dans le dossier du projet : `tp4` et réessayez "`java example.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 dossier `tp4`, le fichier `example/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 example.Robot`"

Cette fois plus de problème car, les fichiers nécessaires sont accessibles depuis le dossier `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 il existe la notion d'**archive java**. L'utilitaire qui permet de les créer s'appelle `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 dossier `classes`. Exécutez la commande :

```
jar cvf ../appli.jar example
```

Vous avez alors **C**réé dans le dossier `tp4` un fichier `appli.jar` qui contient, compressés, tous les fichiers de l'arborescence dont la racine est le dossier `example`.

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

Q 6.4. Utilisation Dans le dossier `tp4`, créez un dossier `tmp`. Copiez y le fichier `appli.jar`. Placez vous dans le dossier `tmp`. Exécutez la commande : "`java example.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 example.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 dossiers ou des archives.

Q 6.5. Extraction. Toujours depuis le dossier `tmp`, exécutez la commande "`jar xvf appli.jar`". Consultez le contenu du dossier `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 dossier `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 `example.Robot` comme classe principale d'une archive.

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

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

Q 7 . Ajouter des ressources. Il est possible d'ajouter des ressources autres que les dossiers des classes à une archive `jar`. Par exemple on peut vouloir ajouter le dossier `docs` contenant la documentation. Pour conserver le niveau de dossier `docs` dans l'archive, il faut avant de l'inclure se "placer au-dessus" de ce dossier et donc **C**hanger de dossier par l'option `-C`

Depuis le dossier `classes` vous devez donc exécuter la commande :

```
jar cvfm ../appli.jar ../manifest-ex example -C .. docs -C .. test
```

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

ou depuis le dossier `tp4` (racine de `classes`) :

```
jar cvfm appli.jar manifest-ex docs test -C classes example
```

A rendre

Q 8 . Rendez via le GitLab ce TP en créant dans votre dépôt un dossier `tp4`.

Déposez-y les fichiers correspondant aux codes et aux tests réalisés dans les questions précédentes ainsi que les codes source et tests correspondant à l'exercice sur les dates étudiés en TD. Vous placerez les types définis dans un seul paquetage nommé `date`. Vous ajouterez une classe « principale » `DateMain` avec une méthode `main` qui fait quelques manipulations sur les objets `Date`. Vous respecterez les consignes présentées ci-dessous en considérant que le fichier `jar` à créer exécute la méthode `main` de `DateMain`.

Exercice 2 : Rendre un travail.
Vous rendrez vos TP via votre dépôt GitLab pour POO. Le **respect des échéances est obligatoire** et sera un critère d'évaluation.
Dans votre dépôt, vous créerez un dossier par TP à rendre. Dans ce dossier on trouvera **nécessairement** :

- le dossier `src` contenant les sources,
- le dossier `test` contenant les fichiers de test (avec le fichier `test-1.7.jar`),
- un fichier texte simple, nommé `readme.md`, dans lequel vous indiquerez
  - les noms des membres du binômes,
  - un paragraphe présentant le TP et ses objectifs
  - une description le cas échéant de ce qui n'a pas été fait ou qui ne fonctionne pas correctement (en précisant dans quel(s) cas) ou ce qui a été fait en plus dans votre programme par rapport au cahier des charges du TP et toute information complémentaire que vous jugerez utile.
  - les instructions **précises** (donner les lignes de commande à chaque fois) indiquant **précisément** comment :
    - générer et consulter la documentation,
    - compiler les sources du projet,
    - compiler et exécuter les tests,
    - générer le fichier `.jar` (sans les sources ni les docs) ,
    - exécuter le programme (avec le jar exécutable et sans le jar exécutable) en décrivant les éventuels arguments à préciser dans la ligne de commande, avec des exemples précis.

  Pour cette étape, vous pouvez ajouter à votre dépôt un `makefile`. Dans ce cas vous créerez des cibles pour la compilation, la génération de la documentation, la création du jar et l'exécution. Et vous en préciserez l'usage dans le `readme.md`.
- ne déposez pas le dossier `docs`, ni les fichiers `.class` qui peuvent ête à générés.

Vous nettoierez votre dossier en en supprimant les fichiers inutiles (`.java~` ou autres).

Cela devrait paraître évident et il devrait être inutile de le préciser mais : **vous ne devrez jamais rendre un tp sans avoir vérifier que les différentes étapes que vous présentez dans votre `readme.md` s'exécutent sans problème : compilation, génération des documentations et des tests, exécution du progamme** (vous ne devez par exemple pas avoir de *warning* lors de la génération de la documentation).