

TP 4

Manipulations

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

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`, `Caisse.java`, `TapisRoulant.java` à récupérer sur le portail et à placer dans le répertoire `src`.

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

Q 1.1. 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 1.2. 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 1.3. Faites ce qu’il faut pour définir les classes `Caisse` et `TapisRoulant` comme classes du paquetage `exemple.util`.

Q 1.4. 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 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 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 2.1. Visualisez votre variable `CLASSPATH` (commande `echo $CLASSPATH`). Si la variable n’est pas définie², rien de s’affiche. JAVA considérera alors qu’elle vaut simplement “.” (le répertoire courant). Si elle est définie, vérifiez que ce même “.” est présent dans la définition.

¹Cette 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.

²c’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 répertoire, il faut utiliser l'option `-d` de la commande (`javac`). Cette option permet de préciser le répertoire *destination* 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.

Q 2.3. Examinez le contenu du répertoire `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 générée³. 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`. Rappelons que la javadoc d'une méthode doit être écrite avant le codage de celle-ci !

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

Q 3.2. 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 3.3. 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 3.4. 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 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 `Caisse` définis dans la classe de test `CaisseTest`.

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/CaisseTest.java` et étudiez les tests proposés.

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

Q 4.3. Compilez la classe de test (en vous plaçant à la racine du projet) :

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

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

`java -jar test.jar CaisseTest`

Tous les tests doivent être passés avec succès.

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

1. pour `prend`, testez les deux cas où le robot porte déjà ou non une caisse avant l'appel de la méthode,
2. pour `porteCaisse`, testez les deux cas de figure,
3. pour `poseSur` testez que si la caisse peut être posée alors ensuite le robot ne porte plus de caisse. 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.

Q 4.6. Compilez et exécutez vos tests.

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

⁴On peut constater que la méthode `poseSur` est "mal écrite" car elle devrait utiliser des exceptions et non des affichages... Nous verrons plus tard comment mettre en place, et tester, ces exceptions.

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 corps de la définition de la classe `Robot`.

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 paquets. 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 exécution.

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 répertoire `classes`. Exécutez la commande :

```
jar cvf ../appli.jar exemple
```

Vous avez alors Créé 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 donc 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'eXtraire les fichiers de l'archive.

Q 6.6. jar exécutable. Un point intéressant est la possibilité 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 **Manifeste** 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. 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 répertoires des classes à une archive **jar**. Par exemple on peut vouloir ajouter le répertoire **docs** contenant la documentation. Pour conserver le niveau de répertoire **docs** dans l'archive, il faut avant de l'inclure se “placer au-dessus” de ce répertoire et donc **Changer de répertoire** par l'option **-C**

Depuis le répertoire **classes** vous devez donc exécuter la commande :

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

(**-C .. docs** signifie “Changer vers le répertoire .. et ajouter **docs** et son contenu à l'archive”)

ou depuis le répertoire **tp4** (racine de **classes**) :

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

Exercice 2 : Rendre un travail.

Vous rendrez vos TP via l'application **PROF** sous la forme d'une archive (**zip** ou **tar**) dont le nom sera de la forme **votreNom-nomDuTP.zip** (ou **.tar**). Le **respect des échéances est obligatoire** et sera un critère d'évaluation.

L'archive à rendre **contiendra systématiquement un répertoire dont le nom sera votre login** et qui regroupera les fichiers :

- le fichier **jar** exécutable de votre TP,
- le répertoire **src** contenant les sources,
- le répertoire **test** contenant les fichiers de test,
- le répertoire **docs** contenant la documentation,
- un fichier texte simple, nommé **lisezmoi.txt**, dans lequel vous indiquerez
 - vos nom et prénom,
 - les instructions **précises** sur comment exécuter le programme décrivant les arguments à préciser dans la ligne de commande,
 - 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 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.

Cela devrait paraître évident et il devrez être inutile de le préciser mais : **vous ne devrez jamais rendre un tp sans avoir vérifier que votre jar exécutable s'exécute effectivement sans problème**. Pour limiter toute situation particulière liée à votre environnement de travail il est conseillé de tester ce **jar** depuis un répertoire différent du répertoire de travail du projet.

Q 1 . Rendez sur **PROF** l'archive correspondant à ce tp avec comme **jar** exécutable associé le fichier **appli.jar**.