TP 5

Paquetages, compilation, exécution, documentation, tests et archives.

Les manipulations présentées ici ne sont réalisables que si vous avez étudié le document préparatoire.

Placez-vous dans un terminal et exécutez (en les comprenant) les différentes étapes décrites ci-dessous. Dans la suite du semestre vous devrez systématiquement effectuer des manipulations similaires pour chacun de vos TP.
Les commandes présentées sont faites pour toutes être exécutées depuis le dossier de base du TP (`tp5` ici).

Exercice 1 :

Q 1 . Récupérez sur le portail l'archive contenant les fichiers pour ce TP et décompressez-la dans le dossier local de votre dépôt.

Placez-vous dans le dossier `tp5` qui a été créé. Il contient les dossiers `src/` et `test/`, ainsi que deux fichiers utiles pour la suite.

Q 2 . Mise en œuvre des paquetages.

Pour illustrer l'utilisation des paquetages, deux paquetages seront créés. Le premier se nommera `factory` et le second `factory.util` (il s'agit donc d'un « sous-paquetage » du premier).

Q 2.1. Rappelons qu'à un paquetage doit correspondre un dossier de même nom.

Dans le dossier `src/`, créez un dossier pour le paquetage `factory`. Celui-ci accueillera toutes les classes du paquetage `factory`.

Dans le dossier `factory` créez un dossier `util` qui accueillera les classes du paquetage `factory.util`.

Q 2.2. On veut que la classe `Robot` appartienne au paquetage `factory`.

Pour définir la classe dans ce paquetage, vous devez donc :

1. Déplacer le fichier `Robot.java` dans le dossier `factory`.
2. Ajouter la ligne «`package factory;`» au début du fichier `Robot.java`.

Q 2.3. En suivant la même démarche, faites le nécessaire pour définir les classes `Box` et `ConveyerBelt` comme des classes appartenant au paquetage `factory.util`[1].

Q 2.4. Importation. La classe `factory.Robot` utilise les classes `Box` et `ConveyerBelt`. Mais ces classes appartiennent maintenant à un autre paquetage : `factory.util`. Il est donc nécessaire d'**importer** ces classes.

Au début du fichier `Robot.java`, vous devez donc ajouter la déclaration « `import factory.util.*;` ». Cet ajout doit être fait après la déclaration du paquetage et avant la déclaration de la classe.

Q 3 . Génération de la documentation.

L'utilitaire `javadoc` permet de générer au format html une documentation telle que celle vue lors des premiers TP. 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[2].

Des *tags* permettent de préciser les différentes informations approtées dans la 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`.

Q 3.1. Pour générer la documentation pour les deux paquetages, vous devez exécuter la commande suivante (depuis le dossier `tp5`) :

```
···/tp5> javadoc -sourcepath src -d docs factory factory.util
```

L'option «`-sourcepath src`» précise le dossier qui contient les sources des paquetages concernés (ici le dossier `src/`), l'option `-d docs` indique le dossier **d**estination des fichiers générés (ici le dossier `docs/`).

La commande suivante produit le même résultat :

```
···/tp5> javadoc -sourcepath src -d docs -subpackages factory
```

[1] Rappel : le sous-dossier doit donc s'appeler `util`
[2] Il faut ajouter l'option «`-private`» lors de l'exécution de la commande pour générer la documentation des éléments privés

CC BY NC SA

en générant la documentation du paquetage `factory` et de tous ses sous-paquetages (ici `factory.util`).

Q 3.2. Allez dans le dossier `tp5/docs/`, qui a été créé. Vous pouvez y retrouver la structure des paquetages.

Ouvrez le fichier `index.html` dans un navigateur, vous accédez alors à la documentation du projet, que vous pouvez parcourir.

Q 3.3. Lorsque vous avez généré la documentation, la trace d'exécution a mentionné un *warning* car la documentation de la méthode `getWeight()` de la classe `factory.util.Box` est incomplète.

Lors de vos rendus, **vous devez veiller à ce que la génération de la documentation ne déclenche aucun *warning* ni erreur**.

Complétez la javadoc de la méthode `getWeight()` dans le fichier `src/factory/util/Box.java` car elle est incomplète.

Regénérez ensuite la documentation comme ci-dessus, en vous assurant qu'il n'y a plus d'alerte, et consultez dans votre navigateur les modifications que vous avez apportées.

Q 3.4. Entrez dans le terminal la commande «`javadoc --help`» : une rapide description des options possibles est affichée. Par exemple : l'option `-author` qui permet de prendre en compte les sections `@author`.

Q 4 . Compilation.

Lors de la compilation, il est nécessaire de préciser où se trouvent les fichiers concernés. Il faut alors indiquer où se trouve les dossiers correspondant aux paquetages.

Q 4.1. Pour compiler la classe `factory.Robot`, vous devez exécuter la commande suivante depuis `tp5/` :

```
···/tp5> javac -sourcepath src -d classes src/factory/Robot.java
```

Le résultat de la compilation (les fichiers «`.class`») est rangé dans le dossier `classes` qui a été créé, dans lequel vous retrouvez la structure des paquetages. On remarque que `Box` et `ConveyerBelt` ont aussi été compilés.

On peut aussi forcer la compilation de toutes les classes du paquetage ainsi :

```
···/tp5> javac -sourcepath src -d classes src/factory/*.java
```

Lors de la compilation, c'est le fichier source qui est compilé, d'où l'usage du «`/`», séparateur de fichier.

Q 5 . Tests.

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

Q 5.1. Ouvrez le fichier `test/BoxTest.java` et étudiez les tests proposés.

La classe `Box` appartient au paquetage `factory.util`. Notez donc qu'il est nécessaire d'importer cette classe : «`import factory.util.Box;`».

Q 5.2. Il faut compiler la classe testée avant de compiler les tests. Ici, la classe `factory.util.Box` a déjà été compilée lorsque vous avez compilé la classe `factory.Robot` ci-dessus.

Pour compiler la classe de test `BoxTest.java`, vous devez exécuter la commande :

```
···/tp5> javac -classpath test4poo.jar test/BoxTest.java
```

On peut aussi compiler toutes les classes de test du dossier `test` ainsi :

```
···/tp5> javac -classpath test4poo.jar test/*.java
```

Q 5.3. Pour exécuter les tests contenu dans la classe `BoxTest`, vous devez exécuter la commande :

```
···/tp5> java -jar test4poo.jar BoxTest
```

Tous les tests doivent être passés avec succès. C'est le cas si la barre est verte, comme ici avec tous les tests (2) passés avec succès.

Q 6 . Première écriture de méthodes de tests.

Vous allez compléter la classe `RobotTest` fournie dans l'achive dont le rôle est de tester la bonne implémentation des méthodes de la classe `factory.Robot`.

En étudiant le fichier `RobotTest.java` vous pouvez noter :

- l'entête avec les différents `import` nécessaires à JUnit, mais aussi les importations des paquetages du projet utilisés pour les tests [3].
- la méthode de test fournie en exemple, préfixée de `@Test` (`robotCarryNoBoxWhenCreated()`).
- la méthode statique `suite()` dont le code fait référence à la classe de test (`RobotTest.class`).

Pour chaque classe de test que vous écrirez vous devrez respecter cette structure.

[3] Une importation est en commentaire pour le moment.

Q 6.1. Compilez `RobotTest.java` puis exécutez le(s) test(s) qu'il contient pour vérifier qu'il est passé avec succès.

Dans une méthodes de test il faut :

1. mettre en place le contexte du test : créer les objets (robot, caisses, tapis roulant) nécessaires au test,
2. mettre en place et vérifier la situation initiale du test (*préconditions*),
3. exécuter la méthode testée,
4. vérifier la situation atteinte (*postconditions*).

Q 6.2. On cherche à tester le bon comportement de la méthode `take()` de `factory.Robot`.
Deux situations sont possibles, selon que le robot porte déjà ou non une caisse avant l'appel de la méthode. Il faut donc définir deux méthodes de test, une par situation :

1. Créez, dans `RobotTest.java`, une première méthode de test, `robotCanTakeBoxIfNotCarrying()`, pour tester que la partie de la documentation « *this robot takes a box if it was not already carrying one* » est respectée. Vous pouvez vous inspirer de la méthode `robotCanTakeLightBox()` du cours.
2. Créez une seconde méthode de test, `robotCannotTakeBoxIfAlreadyCarrying()`, pour tester que la partie de la documentation « *else nothing happens* » est respectée. Vous pouvez vous inspirer de la méthode `robotCanTakeOnlyOneBox()` du cours.

Q 6.3. Compilez (une erreur de compilation ? lisez bien le message et réfléchissez à ce qu'il signifie) et exécutez ces tests qui doivent se dérouler sans erreur.

Q 7 . Exécution du programme.

Q 7.1. Nous allons utiliser comme « classe principale », la classe `RobotMain` fournie. Vérifiez qu'elle contient bien une méthode statique `main`, que vous pouvez lire.
Faites le nécessaire pour que la classe `RobotMain` appartienne au paquetage `factory` : déplacement du fichier et modification de l'entête et ajoutez les importations nécessaires.
Compilez ensuite ce fichier, comme précédemment depuis le dossier `tp5/` :

```
···/tp5> javac -sourcepath src -d classes src/factory/RobotMain.java
```

Q 7.2. On rappelle qu'exécuter un programme Java consiste à exécuter la méthode `main` définit dans une classe.
Exécutez la méthode `main` de `factory.RobotMain` :

```
···/tp5> java -classpath classes factory.RobotMain
```

L'option « `-classpath classes` » permet d'indiquer le dossier qui contient les « classes compilées » permettant l'exécution.

Remarques :

- on peut noter les différences entre la commande de compilation et celle d'exécution.
  La compilation s'applique sur un fichier (source) fournie à la commande `javac`. On note donc le séparateur des dossiers «`/`» et l'extension du fichier `.java`.
  L'exécution quant à elle, concerne une classe (déjà compilée), dont le nom est transmis à la commande `java`. Dans un nom de classe on trouve le "`.`" séparateur des noms de paquetages et aucune extension.
- lors de l'exécution de la méthode de `main`, les classes `factory.util.Box` et `factory.util.ConveyerBelt` sont également utilisées. Leurs définitions sont donc chargées par la machine virtuelle java (JVM). Ce chargement est possible parce que les fichiers de définition des classes utilisées sont disponibles depuis les chemins de dossier indiquées par l'information fournie dans «`-classpath`» .

Q 8 . 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. Une **archive java** permet de regrouper les classes d'un programme pour une utilisation et une diffusion plus faciles. L'utilitaire qui permet de créer ces archives s'appelle `jar` (pour *Java ARchive*). Il est notamment possible de produire des «***`jar` exécutables***».

Q 8.1. Exécutez la commande «`jar --help`» pour voir apparaître un rapide descriptif de la commande et de ses options. Nous n'en présenterons qu'une petite partie.

Q 8.2. **Création d'archive exécutable**. Une archive exécutable est une archive dans laquelle on va préciser la classe dont on veut exécuter la méthode `main` lors de la création de l'archive.
Comme ceci :[4] :

```
···/tp5>jar cvfe factory.jar factory.RobotMain -C classes factory
```

[4]Commande équivalente : `jar --create --verbose --file factory.jar --main-class factory.RobotMain -C classes factory`

Les options de cette commande se comprennent ainsi :

- `c` est pour la **c**réation,
- `v` est pour le mode bavard (***v**erbose*), qui provoque l'affichage de la trace détaillée lors de la création de l'archive,
- `f` signifie que vous précisez ensuite le nom du **f**ichier créé (ici `factory.jar`),
- `e` permet de préciser la classe dont il faudra **e**xécuter la méthode `main` (ici `factory.RobotMain`),
- `-C classes` indique qu'il faut **C**hanger de dossier (ici vers `classes/`) avant d'ajouter à l'archive le contenu du dossier `factory/`.

Q 8.3. Il est possible de consulter le contenu de cette archive en exécutant[5] :

```
···/tp5> jar tvf factory.jar
```

Vous devez retrouver tous les fichiers du dossier `factory/` ajoutés.

Vous pouvez constater en plus l'ajout d'un fichier `META-INF/MANIFEST.MF` présent dans cette archive. Ce fichier s'appelle le *manifeste* de l'archive. Si vous le consultez, vous constaterez qu'il contient, entre autres, l'information précisant la `Main-Class` de l'archive.

Q 8.4. Exécution de l'archive.

Le caractère « exécutable » de cette archive se vérifie lorsque l'on exécute la commandeé:

```
···/tp5> java -jar factory.jar
```

Vous pouvez constater que la méthode `main` de la classe `factory.RobotMain` a été exécutée, comme précisé dans le manifeste de l'archive.

Q 8.5. Une autre manière de créer une archive exécutable consiste à définir explicitement son propre fichier « *manifeste* ».

Dans le dossier `tp5` vous trouvez le fichier `manifest-factory` fourni. C'est un exemple de fichier de définition d'un *manifeste*. En le consultant vous constatez qu'il définit la classe `factory.RobotMain` comme classe principale (`Main-Class`) d'une archive.

L'archive exécutable se construit alors grâce à la commande :

```
···/tp5> jar cvfm factory.jar manifest-factory -C classes factory
```

En plus des options déjà décrites, l'option `m` provoque l'ajout d'un fichier **m**anifeste dont le nom est précisé (ici `manifest-factory`).

L'effet de l'ajout du *manifeste* est visible dans le fichier `META-INF/MANIFEST.MF` présent dans cette archive.

Q 9 . Exercices : autres méthodes de test. Vous allez compléter les tests de la classe `factory.Robot`.

Q 9.1. La documentation de la méthode `putOn()` de `Robot` nous indique son cahier des charges.

Il faut définir une méthode de test pour chacune des différentes situation suivantes pour tester que :

1. `putOn()` renvoie `true` si le robot porte une caisse et s'il a été possible de la poser sur le tapis roulant (car elle est plus légère que le poids maximal supporté par le tapis roulant, fourni à sa construction). Dans ce cas pour le contexte du test, vous devez donc créer un robot, un tapis roulant et une caisse dont le poids est inférieur au poids maximal accepté par le tapis.
2. `putOn()` renvoie `false` si le robot ne porte pas de caisse,
3. `putOn()` renvoie `false` si le robot porte une caisse mais que celle-ci est trop lourde pour la poser sur le tapis roulant,
4. `putOn()` renvoie `false` si le robot porte une caisse mais le tapis roulant est plein.
   Il est ici nécessaire, dans la situation initiale, de « remplir » le tapis roulant (comme on peut le voir dans le code, les tapis roulants ont une capacité de deux caisses).

Dans ces méthodes de test, en plus du résultat de l'invocation de `putOn()`, il faut tester que dans le premier cas le robot ne porte plus de caisse, et, dans les autres cas, qu'après l'exécution de la méthode `putOn()`, une caisse est toujours portée et que c'est la même caisse qu'avant.

Q 9.2. Compilez et exécutez vos tests, qui doivent être passés avec succès.

A rendre

Vous devez respecter **rigoureusement** les consignes de rendu communiquées.

- Rendez via le GitLab ce TP en créant dans votre dépôt un dossier `tp5`.
- Déposez-y les fichiers correspondant aux codes et aux tests créés et utilisés dans les questions précédentes.
- Rédigez un fichier `readme.md` complet conforme aux consignes. Vous utiliserez la syntaxe *markdown* pour sa mise en forme et vérifierez le résultat en consultant votre dépôt dans un navigateur.

[5] Un fichier `jar` est une archive compressée au format *zip*, vous pouvez donc la consulter à l'aide d'un gestionnaire d'archives.