

Compte rendu TD 3

Programmation Orientée Objet



Si la classe *Position* ne construisait pas d'objets constants, combien faudrait-il d'instances pour représenter la position d'un passager dans la classe *Passager Standard* ?

Comparez avec le nombre d'instances dans le code actuel.

Supposons que la classe *Position* ne construit pas d'objets constants, donc il y a une ou plusieurs méthode qui pourrait modifier l'état de l'objet (la valeur de ses variables d'instance), donc logiquement à chaque fois qu'on veut modifier l'état du passager on utilise une méthode d'instance spécifique. Donc une seule instance dans *Passager Standard* suffit. Conclusion: Si *Position* ne construit pas d'objets constants, il faut autant d'instances de *Position* que d'instances de *Passager Standard*.

Si on compare avec le nombre d'instances dans le code précédent, on retrouve le même résultat à chaque instantiation de *Passager Standard*, l'attribut *Position* est instancié suivant sa position debout assis ou dehors avec le constructeur adapté dans la classe *Position*.

Rq: Si on modifie l'état du passager, on utilise la méthode de classe publique *estAssis* par exemple pour rendre le passager assis et là, on retourne un *new Position(ASSIS)* donc à chaque modification on instancie à nouveau la classe pour modifier l'état courant du passager contrairement à l'instanciation de *Passager Standard* initial où il faudrait choisir parmi les 3 soit une instantiation de *Position*.

Expliquer vos modifications ?

A quel moment la classe *Position* est-elle instanciée ?

Pour permettre le partage d'instances, il suffit de créer trois instances correspondantes aux positions debout, assis et dehors dans la classe *Position*. Ils seront déclarés dès le début de la classe avec le mot clé: *private final static* ; c'est à dire ça serait indépendamment de l'instanciation de la classe tout entière *Position*. Prenons un exemple dans le fichier test de *Position*, on a mis dans une fonction test: *Position p=Position.assis()* ; c'est ça à quoi ça sert le mot clé *static* : à utiliser les attributs d'une classe sans pouvoir les instancier avant.

C'est un attribut de *Position* donc il doit être déclaré en *private* pas en *public* (il y a des getters pour accéder à la valeur de ce dernier depuis un fichier test par exemple de l'extérieur). Ainsi reste le mot final qui veut signifier le fait que cet élément ne peut être changé à la suite du programme, si on veut modifier l'état du passager (on parlera après de cela) on va appliquer: *this=Position.Dehors* pour faire sortir le passager par exemple, mais aucune méthode de *Position* ne pourrait changer l'attribut de *Position* *assis=new Position(ASSIS)* qui sera instancié à l'intérieur de la classe *Position* elle-même.

Ainsi, nos modifications consistent à construire trois attributs *assis*, *dehors* et *debout* dans la classe *Position* en instanciant les trois avec le constructeur qui prend en paramètre *DEHORS*, *ASSIS* ou *DEBOUT*.

On va supprimer le constructeur public qui ne sert plus rien à présent, car dans `Passager Standard`, si on veut une position, on y accède directement avec le mot clé `static` et en plus les 3 instanciations (attributs) faites avant sont construites avec le constructeur privé: `private Position(courant e)`. Les trois attributs `DEHORS`, `ASSIS` et `DEBOUT` sont déclarés en `static` aussi car les instanciations `static` ont besoin d'arguments `static` aussi pour que cela marche. Mais l'attribut `int courant`, on le laisse tel quel.

On ajoute 4 autres modifications, les vérificateurs de l'état du passager. Là, au lieu de faire un test booléen du genre: `courant==DEHORS` par exemple, on testera directement l'objet instancié avec les 3 instances possibles, cela donnerait par exemple pour l'état `assis`: `this==Position.Assis`.

Reste les 3 fonctions qui retournent une nouvelle instance `debout`, `assis` ou `dehors` ; là au lieu de retourner une nouvelle instanciación avec le mot clé `new` (ces fonctions servent avant à modifier l'état du passager au niveau de l'autobus à chaque arrêt) on retourne tout simplement `Position.dehors` pour `estDehors()` par exemple. Les 3 fonctions seront déclarées en `static` pour la raison expliquée avant. Pour la méthode `String`, rien à signaler. Pour être plus clair, les vérificateurs dont la modification a été signalée dans les paragraphes précédents n'ont pas besoin d'un mot clé `static` car ils agissent pas sur les 3 instances déclarées en `static` alors que les nouvelles fonctions retournent une instanciación qui est censé être `static` et utilisé ultérieurement donc ça doit être déclaré en `static` par contre une simple comparaison au sein de la classe n'a pas besoin de l'être.

Après on passe les tests de `testPosition` après avoir fait les modifications nécessaires dans ce fichier et tout est OK jusqu'à là.

Passons au fichier `Passager Standard`, comme expliqué précédemment, le constructeur va faire: `this.position=Position.Dehors` sans instancier la classe `Position` pour y faire un attribut objet car c'est `static`. Les vérificateurs ne changent pas, ils utilisent les mêmes vérificateurs de `Position` avec `maPosition.estInterieur()` par exemple pour tester si le passager est à l'intérieur du bus ou non que les modificateurs de `Position` qu'on a expliqués avant. Reste les changements `changerEnAssis` et de même pour changer l'état en `dehors` ou en `debout`. Il faudrait changer la syntaxe en: `maPosition=Position.Assis` et non pas `return maPosition.assis` car on crée pas d'instances à chaque modification d'état comme on a dit au début, on partage les instances soit 3 instances suffisent pour implémenter `Passager Standard` et l'utiliser après.

Avec le fichier faussaire `Faux Véhicule.java`, le `Passager Standard.java`, les 2 classes de départ: la jauge et la position ou bien que `Position` pour l'instant (test faussaire) ; les tests passent en vert. Tout est ok pour la classe `Passager Standard`.

RQ: C'est écrit dans le sujet que la classe construit des objets constants et que ce partage doit être mis en œuvre par la classe, d'où le mot final dans les 3 instances de départ qui définissent les 3 états possibles d'un passager standard.

2ème partie de la question:

Supposons qu'on implémente une classe sans constructeur, le compilateur javac va mettre un constructeur par défaut ou bien remplir un tableau dont la taille est déjà connue par des null si il est pas rempli, ce qui veut dire que le compilateur a besoin d'assez d'informations sur les attributs de la classe avant même de produire le .class après.

Donc, en s'appuyant sur cet argument, c'est au moment même de la compilation, au moment de produire les fichiers .class correspondants que le compilateur instancie ces 3 classes pour voir à quels attributs il en à faire.

3ème question: (réécriture du code de Position.java)

Il faudrait un peu plus optimiser le code de Position.java. Pour cela, les 3 premiers attributs DEHORS, ASSIS, DEBOUT qui servent à l'instanciation de nos trois instances de Position pour permettre le partage vont être mis en commentaire, reste à régler le problème des paramètres à l'instanciation. Pour cela on supprimera les arguments à l'instanciation et on mettra en commentaire le constructeur privé. Ainsi l'instance à l'intérieur de la classe correspondante à l'état ASSIS va s'écrire:

```
private final static ASSIS=new Position();
```

Avec le mot ASSIS, on saura l'état du passager sans avoir à ajouter des attributs et un constructeur alors que le nom de la variable signifie l'état en question. Pour les autres fonctions, commençant par les getters est Debout par exemple ca va retourner : this==Position.estDebout et les fonctions qui modifient l'état indirectement en retournant la bonne instance et qui sont bien sûr déclarés en static, il prennent aucun argument et retourne pour estAssis: this==Position.Assis.

Cela a été déjà fait partiellement, on a plus besoin de courant comme attribut que des 3 instances ; même le constructeur vu le nom de la variable peut ne rien faire car l'instanciation appelle le constructeur sans arguments. On peut le virer mais c'est mieux qu'il persiste dans le code car le compilateur va introduire un constructeur par défaut qui ne fait rien dans le cas contraire.

RQ: dans la méthode String du fichier Passager Standard.java, on retourne "test" tout simplement pour simplifier le code car le but est de tester Passager Standard le vrai code en s'appuyant sur Position avec un fichier faussaire: fauxpassager.java qui a été mis en place pour tester le code qui dépend d'un autre sans que l'autre tandem ait finit sa partie.

Utilisez cette méthode pour montrer (à travers un exemple) le code que doit générer le compilateur pour respecter la spécification du langage Java.

Le 1er exemple retourne true pour les deux. Il n'y a pas d'instanciation en utilisant la classe String. En revanche pour le deuxième, ça doit retourner false car on instancie les classes ici, les emplacements mémoire sont différents, un equals(anObject Object) à la place d'un double égal retournerait true dans le 1er cas du 2ème cas de figure. On précise dans ce dernier cas que "Kalki" est une instance implicite.

Retournons à notre question. La classe String du kit JDK où les méthodes sont en static dispose d'une méthode particulière appelée: intern(). Cette méthode permet de déclarer un String avec par exemple String s=s1.intern(); sachant que s1 est instancié et de retourner vrai même s' il y eu instantiation. Cela permet de lever le problème de non-partage des instances. Par exemple, explorons le code suivant:

1. public class InternExample{
2. public static void main(String args[]){
3. String s1=new String("hello");
4. String s2="hello";
5. String s3=s1.intern();//maintenant, ca sera la même chose que s1
6. System.out.println(s1==s2);//faux car les références de variables pointent vers différentes instances
7. System.out.println(s2==s3);//vrai car les références de variables pointent vers les mêmes instances}}

La méthode intern tirée de la documentation de la classe String de JAVA permet d'instancier sans pouvoir que les références des variables instanciées pointent vers différentes instances et avoir une égalité au niveau des instantiations après le test avec les double égal(==).

Après, on a réorganisé notre dépôt en créant 3 répertoires: le src qui contient les vraies code sources comme Position/Jauge/Autobus/PassagerStandard.. et le tst qui contient les fichiers qui servent aux tests faussaires ainsi que les tests du début testPosition/testJauge et enfin le fichier build qui contiendrait tous les fichiers .class issus de la compilation des .java.(On verra la commande par la suite qui permet de faire ceci)

Expliquer l'option -d.

On compile les fichiers sources de cette manière: javac -d build src/*.java . L'option -d permet de spécifier où mettre les fichiers compilés (.class de src/.java) après compilation. ça vient du mot anglais directory qui signifie folder ou répertoire de travail. Ça permet aussi de créer un répertoire correspondant au paquet ici tec dans le répertoire build.

Expliquer l'option -cp.

Pour compiler les fichiers tests avec javac avant compilation, il faudrait gérer les dépendances. En gros, il faudrait indiquer où trouver les fichiers dont un autre fichier en a besoin. C'est le but de l'option de compilation -cp. Avec javac -d build -cp build tst/*.java, on spécifie au compilateur que les fichiers tests après compilation les mettre dans le répertoire build et que n'importe quel fichier dans tst qui dépend d'une autre classe va se retrouver dans le répertoire build. D'où l'utilité de l'option -cp à la compilation sur le terminal.

Donner la commande qui lance l'exécution de la classe **TestJauge à partir de la racine de l'arborescence.**

La commande comme expliquée avant est la suivante:(-cp fait référence à CLASSPATH le chemin vers les classes dont dépendent les fichiers à compiler sur place)

`javac -cp build -ea tst/TestJauge.java` sans que le `.class` résultant ne soit dans `build`.
TestJauge dépend de Jauge.class qui a été compilé et mis dans `build` avec:

`javac -d build src/*.java`. Cela sera lancé depuis la racine de l'arborescence avec le fichier dans le répertoire `test`.

Puis on lance l'exécution avec `java -ea tst/TestJauge` pour activer les assertions. (Cela se fait à l'exécution par à la compilation)

On a produit un Makefile adapté qui en plus de compiler dispose d'un nombre de target qui appelle les lignes de compilations puis exécute la cible en question. Tous les tests passent en vert, tout est OK.

Quelle instruction faut-il ajouter au début de chaque fichier source ?

Pour que toutes les classes appartiennent au même paquetage pour des raisons d'utilisation du paquetage et de ses classes particulières après éventuellement, il faut ajouter la commande suivante au début de chaque fichier avant le `public class`:

```
package tec;
```

Le package après le lancement du Makefile s'appellera `tec`, et une classe `x` sera appelé par `tec.x` ou bien `import tec.*` pour utiliser la classe `x` sans préciser `tec.x` mais seulement `x`.

(Voir les TDs suivants)

Comment déclarer une classe avec une portée interne au paquetage ?

Rappel:

Les modificateurs de visibilité d'une classe par rapport à une autre ou un package (possible) sont indiqués devant l'en-tête d'une classe (ou d'une méthode ou devant le type d'un attribut dans un autre contexte de portée de données). Lorsqu'il n'y a pas de modificateur, on dit que la visibilité est la visibilité par défaut.

Une classe ne peut que :

- avoir la visibilité par défaut (sans aucun mot clé): elle n'est visible alors que de son propre paquetage.
- se voir attribuer le modificateur **public** : elle est alors visible de partout.

Pour déclarer une classe avec une portée interne au paquetage, il faut pas de mot clé avant le `public class` au début du fichier (visibilité par défaut) ; une déclaration avec `public` rendrait la classe accessible par tous les fichiers à l'extérieur (même pas appartenant au paquet du tout et au répertoire par des systèmes de dépendance au niveau de la ligne de compilation).

C'est comme les champs d'une classe.

Si un champ d'une classe `A` :

- est **private**, il est accessible uniquement depuis sa propre classe ;

- a la visibilité paquetage (par défaut), il est accessible de partout dans le paquetage de A mais de nulle part ailleurs ;
- est **protected**, il est accessible de partout dans le paquetage de A et, si A est publique, grosso modo dans les classes héritant de A dans d'autres paquetages ;
- est **public**, il est accessible de partout dans le paquetage de A et, si A est publique, de partout ailleurs.

Résumé: pour rendre une classe interne au paquet, il faut pas 'utiliser de mot clé comme public,etc.. juste déclarer la classe normalement (class Position{...} dans tec par exemple pour la rendre interne à tec). La classe Autobus par exemple doit être déclarée comme accessible depuis l'extérieure du paquet soit un public class Autobus au début de Autobus.java. (comme indiqué dans la feuille de td [3])

Quelle contrainte y-a-t-il sur l'organisation des fichiers des classes compilées ?

Le principale problème qui se retrouve en réorganisant les fichiers classes compilées c'est de pouvoir gérer les dépendances et lancer les tests pour chaque classe ou bien les tests de recette tout à la fin en ajoutant les bonnes options à la compilation et en précisant où les classes (fichiers compilés) doivent se retrouver et en prendre compte avec l'option -cp pour exécuter un test sinon cela ne marcherait pas. Il faudrait compiler dans les bons répertoires et de préciser le package tec dans chaque fichier pour éviter certaines erreurs de compilation. C'est pour cela qu'un Makefile sera opérationnel, dynamique et utile dans ce cas. Quand on a une classe publique compilée on pourrait l'utiliser ailleurs pour d'autres manipulations et donc le package auquel elle appartient perd son vrai sens en quoi tous ses fichiers construisent un projet unique, ça pourrait être une problématique, de même disposer des .class seulement comme dans le td1 ne permet pas une exploitation approfondie du code et du débogage donc une réutilisabilité très limitée de la part d'un client externe.

Donner le nom complet d'une des classes de test ?

Après un make source puis un make test on compile les tests à l'aide des .class des sources compilées dès le début. Le nom complet d'une des classes de test est par exemple:

build/TestPosition.class depuis le répertoire principal.

OU:

build/TestPassagerStandard depuis le répertoire principal.

etc..

1. **Donner la commande pour compiler les fichiers du répertoire `src` et vérifier l'emplacement des fichiers compilés.**

```
javac -d build src/*.java
```

```
ls build
```

=>on retrouve tous les fichiers en extension `.class`.

2. **Donner la commande pour compiler les fichiers du répertoire `tst` et vérifier l'emplacement des fichiers compilés.**

```
javac -d build -cp build tst/*.java
```

```
ls build
```

=>on retrouve tous les fichiers tests en extension `.class`, les dépendances ont été bien prises en compte.

3. **Donner la commande qui lance l'exécution de la classe de test à partir du répertoire du projet⁴.**

On commence tout d'abord par compiler les sources : `make source`. Tous les fichiers sources sont en `.class` dans le répertoire `build`.

Après, on exécute la commande suivante:

```
javac -d build -cp build Simple.java (Simple.java est dans le répertoire principale)
```

Puis, vient l'exécution du test globale via:

```
java -cp build Simple (pour déterminer le $classpath value)
```

DERNIÈRE QUESTION:(remarque plutôt)

Pour avoir un bon affichage après l'exécution des tests de recette (`make source`, `javac -d build -cp build Simple.java` puis `java -cp build Simple`) ; il faudrait ajuster les `toString` d'`Autobus` et des deux classes `PassagerStandard` et `Jauge`. Pour cela on adopte le même format d'affichage qu'en C pour avoir le même format d'affichage sur le terminal.

->Pour `autobus`, la méthode d'instance publique `toString` est écrite de cette façon ou plus précisément elle retourne:

```
"[arret " + arretCourant + "]" + "assis" + jaugeAssis + " debout" + jaugeDebout;
```

->Pour `jauge`, de la même manière, le retour de la fonction est:

"<" + valeur + ">" ;

->Pour passer standard, ça sera:

nom + " " + maPosition;

Rq: Tous les toString des 3 classes sont sans arguments fonctionnels bien sûr.

Commentaires

- Martial :
- Nathan :
- Maxime :
- Guillaume:
- Ismail : Ce td est plus instructif que les 2 derniers. Cela nous a permis d'approfondir nos connaissances sur les paquets déjà abordés en cours ainsi que de pouvoir bien compiler tout en gérant les dépendances et les emplacements de fichier. Cela nous a aussi conduit à mettre en œuvre un Makefile pour automatiser les compilations/exécutions avec les bonnes dépendances récursives dedans et savoir comment les ordonner (faire un make test précède faire un make source par exemple). Le partage des instances est une notion nouvelle, très intuitif et qui nous a pris peu de temps pour pouvoir passer les tests avec mais cela a été très utile pour nos progressions en langage JAVA et le paradigme objet qui en fait sujet.