Programmation Orientée Objet
TD 3 : Intégration et rassemblement
Compte-rendu de l'équipe *Lorem Ipsum* (G4E1)

Coordinateur : Sébastien DELPEUCH, Tandem 1 : Rémi DEHENNE, Aymeric FERRON, Tandem 2 : Tom MOËNNE-LOCCOZ, Aurélien MOINEL

25 septembre 2020

Résumé

Dans le cadre de cette troisième séance de TD des modules PG202–PG203 à l'ENSEIRB-MATMECA, nous abordons les concepts d'arborescence de fichiers compilés par l'utilisation de paquetages, ainsi que la redéfinition de fonctions et le partage d'instances dans une classe.

1 Compilation et arborescence de fichiers

Lors des semaines précédentes, nous avions déjà séparé les fichiers de sources, les sources des tests et le *bytecode* Java dans des répertoires différents, à l'aide de règles de compilation et d'options en ligne de commande. Nous souhaitons désormais organiser les fichiers compilés à l'aide de paquetages.

1.1 Compilation des sources

À la compilation, on souhaite que l'ensemble des fichiers produits soient générés dans un répertoire distinct, et ne viennent pas « polluer » les répertoires du code source ou la racine du projet. Pourtant, la plupart des compilateurs créent par défaut les fichiers dans le répertoire courant. Pour modifier ce comportement, le compilateur Java propose une option `-d <répertoire>` afin de spécifier le répertoire racine des fichiers compilés.

Cependant, redéfinir le répertoire de compilation nécessite d'indiquer ce même répertoire à l'exécution, et lors de la compilation de fichiers qui utilisent des fichiers déjà compilés. En effet, les fichiers de *bytecode* Java sont recherchés par défaut dans le répertoire courant. Les options `-cp` ou `-classpath` sont alors utilisées afin d'indiquer dans quels répertoires chercher les fichiers `.class` utilisés. Sous un système d'exploitation UNIX-like, les répertoires de `classpath` sont séparés par le caractère `:` et sous Windows par `;`.

Pourtant, générer le *bytecode* Java dans un unique répertoire peut vite apparaître chaotique : il serait préférable de pouvoir organiser les fichiers compilés pour faciliter leur utilisation par d'autres utilisateurs.

1.2 Utilisation de paquetages

Même si le code source peut être organisé en sous-répertoires, les fichiers compilés sont par défaut placés dans un seul répertoire. Ainsi, deux classes de même nom placées dans des répertoires de sources différents seraient compilés dans le même fichier `.class` et s'écraseraient. De plus, outre les portées `public` et `private`, il peut être nécessaire d'interdire le code client à accéder à certaines classes ou membres d'une classe, tout en autorisant certaines classes internes à y accéder.

Pour cela, le langage Java propose le mécanisme de *paquetages*. Chaque fichier peut être affilié à un paquetage par une déclaration `package <nom.du.paquetage>;`, qui l'autorise à accéder aux attributs, méthodes et classes du même paquetage dont la visibilité est *paquetage*[1]. Le nom du paquetage définit également l'arborescence des fichiers dans le répertoire de compilation : ainsi, en compilant un fichier `Jauge` appartenant au paquetage `fr.enseirb-matmeca.tec`, le répertoire de compilation sera organisé sous la forme :

```
.
|-- fr
     |-- enseirb-matmeca
          |-- tec
               |-- Jauge.class
```

L'organisation des fichiers `.class` dans le répertoire de compilation ne dépend donc pas de l'organisation des fichiers sources. Ainsi, dans ce TD, les fichiers de sources et de tests sont stockés dans des répertoires différents mais appartiennent au même paquetage `tec`. Les fichiers compilés sont alors stockés dans le même répertoire `tec`. Cette indépendance entre l'arborescence des sources et des fichiers compilés permet d'organiser le code source de manière *logique*, tant pour les mainteneurs que pour les utilisateurs. En effet, un mainteneur organise son code selon des choix d'implémentation, qui n'ont pas nécessairement de sens pour un utilisateur. Les paquetages peuvent donc rendre une bibliothèque plus claire et plus facile à prendre en main pour cet utilisateur, qui n'a pas à se soucier de l'architecture réelle des sources[2].

1. Une classe ou un membre de classe ont pour portée *paquetage* lorsqu'aucun mot-clé de portée n'est indiqué. Seules les autres classes du paquetage peuvent alors y accéder.

2. En langage Java, l'indépendance entre organisation des sources et des fichiers compilés n'est cependant pas totale : un fichier `.java` est ainsi traduit par un fichier `.class` de même nom.

L'utilisation de paquetages implique de modifier certains arguments passés à la machine virtuelle Java. À la compilation, rien ne change : on compile le code source avec la commande `java -d build src/*.java`, puis éventuellement le code source des tests avec `javac -d build -cp build tst/*.java`, en indiquant au compilateur de chercher les fichiers de *bytecode* Java depuis la racine `build/`. À l'exécution, cependant, la classe à exécuter doit être préfixée par son nom de paquetage. Ainsi, la commande `java -ea -cp build tec.Test*` permet d'exécuter tous les tests du répertoire `tec`.

2 Amélioration de l'existant : affichage des instances et partage d'objets

On désire désormais améliorer le code des séances précédentes.

2.1 Redéfinition de la méthode `toString()`

Pour faciliter le débogage, il est utile de disposer d'une méthode permettant de représenter l'état des instances sous forme d'une chaîne de caractères. En langage Java, toute classe hérite de `java.lang.Object`, qui définit une méthode `toString()` par défaut. Néanmoins, cette implémentation initiale ne permet de connaître que le nom et l'adresse de l'objet. Pour améliorer ce comportement pour une classe donnée, la méthode peut être surchargée : on réécrit une méthode de même nom, éventuellement annotée par `@Override`, et qui renvoie une chaîne de caractères représentant l'état de l'objet de manière plus explicite. Le Code 1 propose une redéfinition de `toString()` pour la classe `Jauge`.

Code 1 : Exemple de redéfinition de la méthode toString() pour la classe Jauge

```
@Override
  public String toString() {
    return "<" + valeur + " [0," + MAX + "[>";
  }
```

2.2 Partage d'instances de `Position`

De plus, à des fins d'optimisation, on souhaiterait éviter de dupliquer certains objets similaires au sein du code. L'implémentation actuelle de la classe `Position` peut être coûteuse en mémoire, en particulier si le nombre de passagers est important. En effet, chaque instance de `Position` est constante :

de ce fait, chaque changement d'état d'un passager crée une nouvelle instance de classe. Ainsi, pour un nombre n de passagers réalisant x changement(s) d'état[3], la complexité spatiale est de $O(n + \sum_{i=1}^{n} x_i)$, avec x_i le nombre de changement(s) d'état du passager i[4].

Une solution pourrait consister à retirer le caractère constant des différents objets. Ainsi, pour chaque passager, une unique instance muable de `Position` pourrait être modifiée par effet de bord à chaque changement d'état. On limiterait donc le nombre d'objets `Position` à un par passager. La complexité spatiale alors obtenue vaudrait $O(n)$.

Cependant, cette complexité pourrait être améliorée davantage. En effet, nous remarquons que deux passagers peuvent occuper une même position. On préfère donc réaliser une unique instance de la classe `Position` par position possible. L'ensemble des passagers partage alors quelques instances constantes de la classe `Position`. Il en résulte donc un nombre d'instances de positions variant en fonction du nombre possible de positions, indépendamment du nombre de passagers. Dans notre cas, il existe seulement trois positions (`DEHORS`, `ASSIS`, `DEBOUT`) : la complexité spatiale résultante est donc constante ($\Theta(1)$).

Afin de mettre en place ce partage, trois instances de `Position` sont stockées en tant qu'attributs de la classe `Position`, grâce au mot-clé **`static`** et initialisées au moment du chargement en mémoire de la classe `Position`. Le mot-clé **`final`** de ces attributs assure quant à lui qu'ils référenceront toujours les mêmes instances, et la portée **`public`** rend ces attributs utilisables par d'autres classes. On s'assure finalement que seule la classe puisse appeler le constructeur `Position`, en le déclarant privé.[5]

Cependant, ce code de `Position` n'est pas indépendant de la classe `TestPosition` et nécessite des corrections. En effet, dans le code précédent, on crée une nouvelle instance de `Position()`, sur laquelle on appelle une méthode de changement d'état. Par conséquent, l'instanciation `Position pos = new Position().debout()` devient simplement `Position pos = Position.DEBOUT`

2.3 Partage d'instances de chaînes de caractères

La bibliothèque standard Java recourt également à des mécanismes similaires. Ainsi, nous avons détaillé dans le deuxième rapport le partage d'instances pour les chaînes de caractères littérales.

Ce mécanisme peut également être utilisé avec des chaînes de caractères non litérales. En effet, la méthode `intern()` de la classe `String` utilise la même *pool* que celle des chaînes littérales, qui ré-

3. La valeur de m varie pour chaque passager.

4. On ne considère pas ici la libération de mémoire des instances qui ne sont plus utilisées, puisque cette libération est réalisée de manière non déterministe par le ramasse-miettes.

5. Il s'agit d'une implémentation du patron de conception *singleton*, où exactement trois singletons de `Position` sont créés.

pertorie les différentes chaînes de caractères déjà instanciées par `intern()`. Pour chaque instance de `String` sur laquelle elle est appelée, elle vérifie si la chaîne de caractères est déjà répertoriée : le cas échéant, elle renvoie une référence vers une instance de `String` identique dans la *pool*, sinon elle ajoute la nouvelle instance à la *pool*. Même s'il est nécessaire d'instancier une chaîne de caractères pour appeler `intern()` (`String maChaine=new String("Kamoulox").intern()`), cela permet à terme de limiter le nombre de chaînes identiques dans le tas : les chaînes instanciées pour appeler `intern()` seront en effet libérées par le ramasse-miettes si elles sont inutilisées.

Code 2 : Code généré par le compilateur lors de la création de chaînes littérales

```
// Comportement désiré (deux instances créées, mais une seule conservée)
String s1 = "Kalki";
String s2 = "Kalki";
System.out.println(s1 == s2); // => true

// Code théorique (trois instances créées et conservées)
char valeur[] = {'K', 'a', 'l', 'k', 'i'};
String s1 = new String(valeur);
String s2 = new String(valeur);
System.out.println(s1 == s2); // => false

// Code généré par le compilateur (trois instances crées, mais une seule
   conservée)
char valeur[]={'K','a','l','k','i'};
String s1 = new String(valeur).intern(); // L'ajout d'intern() évite la
   duplication d'instances.
String s2 = new String(valeur).intern();
System.out.println(s1 == s2); // => true
```

Ce partage d'instances est permis par le caractère immuable des instances. Même si la complexité spatiale générale n'est pas diminuée puisqu'autant d'instances sont allouées, l'impact mémoire est *à terme* amoindri, une fois les instances identiques libérées par le ramasse-miettes. Pour un nombre n de chaînes de caractères de longueur m, le coût mémoire serait après libération de la mémoire de $O(n+m)$ (n références pointant vers la même chaîne de longueur m), au lieu de $O(n \times m)$ lorsque les instances sont copiées (n références pointant vers n chaînes de longueur m).

Par ailleurs, deux chaînes x et y de la *pool* peuvent être comparées en temps constant $\Theta(1)$, par comparaison d'adresses avec l'opérateur == : il n'est donc pas nécessaire de comparer ces chaînes caractère par caractère avec `equals()`, cette opération étant linéaire en le nombre de caractères ($O(min(|x|,|y|))$).

Par conséquent, le partage d'instances permet de limiter la consommation de mémoire et peut permettre d'accélérer les opérations de comparaison. Cependant, ces instances doivent être immuables afin d'éviter des comportements imprévisibles causés par des effets de bord.

Conclusion

Finalement, ce troisième TD nous a permis d'organiser le *bytecode* Java généré, par l'utilisation de paquetages et de nous familiariser avec la notion de *partage d'instances*, afin d'améliorer l'efficacité du code.

Commentaires

Sébastien (Coordinateur) : Suite au passage en co-modal je me retrouve coordinateur seul en présentiel avec mes 4 camarades en distanciel. Pour fonctionner nous avons mis un Discord en place, mes 4 camarades sont sur le Discord et je navigue entre les salons pour rester au courant de l'avancement, recueillir les questions et faire le lien avec l'enseignant. Mélanger le présentiel et le distanciel en même temps est très complexe à gérer. Une partie du travail a été fait à la maison car le co-modal m'a rendu beaucoup moins productif.

Rémi (Tandem 1) : Cette semaine, j'ai essentiellement « bataillé » avec Aymeric pour trouver l'origine des erreurs de compilation dues à l'utilisation de paquetages. De manière surprenante, compiler les sources et les fichiers de tests un à un sans se soucier de l'ordre de compilation ne posait pas de problème lors de l'utilisation d'un paquetage anonyme. Avec un paquetage nommé, pourtant, les fichiers doivent être compilés dans l'ordre des dépendances. Au final, la solution la plus simple consiste à compiler tous les fichiers d'un répertoire en une seule commande, afin que les dépendances soient gérées par le compilateur et non par le Makefile. En réalité, en Java, on préférerait sans doute utiliser un outil de compilation automatique plus adapté, tel que Maven ou Gradle.

Aymeric (Tandem 1) : Rémi et moi avons travaillé sur la première partie du TD. Dans un premier temps, nous avons modifié l'arboresence et le Makefile, ce qui n'a pas posé de diffculté. Dans un second temps, nous avons créé les paquetages. Cela a été l'occasion pour moi d'affiner ma compréhension des mots clefs Java `private` et `public` : `private` permet de rendre la méthode ou la variable accessible uniquement depuis la classe à laquelle elle appartient alors que `public` permet d'y avoir accès depuis l'intégralité du code. Ne rien spécifier permet ainsi d'accéder à l'élément depuis tout le paquetage.

Au moment de créer les paquetages, nous avons été confrontés à de nombreuses erreurs de compilations. En effet, notre Makefile compilait les fichiers un par un dans le désordre sans se soucier des dépendances entre les fichiers. Cette situation ne présentait pas de problème avant l'insertion des paquetages, mais suite à cela, nous avons dû modifier le Makefile. Ainsi, il compile désormais toutes les sources en une seule commande grâce à laquelle le compilateur gère les dépendances entre elles, puis les tests sont alors compilés selon le même procédé afin que le compilateur gère les dépendances entre les tests ainsi que les dépendances des tests aux sources.

Aurélien (Tandem 2) : Cette séance a été l'occasion de redécouvrir le partage d'objets entre classes, mais plus particulièrement de mieux comprendre le comportement, et le code généré du compilateur Java lors de l'utilisation d'une énumération. Je m'interroge néanmoins sur la convention utilisée pour l'arborescence liée aux paquetages. En effet, nous avons les répertoires `src` et `tst` séparés, mais tous deux possèdant le paquetage `tec`. N'aurait-il pas mieux valu dans ce cas déplacer le répertoire `tst` dans `src` ?

Tom (Tandem 2) : Pendant cette séance, j'ai principalement travaillé sur la partie 4 du TD concernant le partage des instances de la classe `Position`. Cela m'a permis de remettre en doute et de solidifier mes connaissances sur les mots-clés pour les états (**`static`** et **`final`**) ainsi que les modificateurs (**`private`** et **`public`**). Sur ces points, Aurélien et Rémi m'ont grandement aidé.