TP de Compléments en Programmation Orientée Objet n° 1 : Gradle, objets, classes et encapsulation

Attention : partie III) à rendre sur Moodle

À propos

Gradle : « moteur de production », sorte de GNU make évolué. Gradle est en fait plus comparable à Apache Maven. La configuration de Gradle consiste à déclarer un certain nombre de paramètres (plugins de compilation, chemin du kit de dévelopement, dépendances, ...), puis à laisser l'outil gèrer les différentes étapes pour compiler et/ou exécuter le projet (téléchargements, exécutions du ou des compilateurs et outils, ...). En cela il est assez différent de make, où l'on doit explicitement donner la suite des étapes.

Gradle est très bien supporté par les principaux environnements de développement pour Java (IntellIJ IDEA, Eclipse, NetBeans, Android Studio, ...).

I) Configurez gradle

a) Sur les machines fixes de l'UFR (pas sur les portables !)

Gradle a souvent besoin de télécharger des fichiers sur le web pour fonctionner (mises à jour de gradle, dépendances de vos projets, etc.).

Pour que cela fonctionne en salle de TP, il faut configurer le proxy HTTP(S) :

1. créez un dossier `$HOME/.gradle/` (s'il n'existe pas déjà)
2. créez un fichier `gradle.properties` dans le dossier `$HOME/.gradle/` (s'il n'existe pas déjà)
3. ajoutez-y les lignes suivantes :

```
systemProp.https.proxyHost=cache
systemProp.https.proxyPort=3128
systemProp.http.proxyHost=cache
systemProp.http.proxyPort=3128
```

(si elles n'y sont pas déjà, sinon vérifier que les clés correspondantes contiennent les bonnes valeurs)

b) Sur vos machines

Il faut que Gradle soit installé.

— Si vous avez un système de paquets (comme `apt` sous Debian/Ubuntu/...), installez gradle via le gestionnaire de paquets. Ex :

```
$ sudo apt install gradle
```

— Sinon vous pouvez suivre les instructions sur le site de Gradle (`https://gradle.org/install/`).
Remarque : si vous faites ainsi, vous aurez directement la bonne version de Gradle, ce qui rend inutile l'étape II) a) 2. Si vous sautez cette étape, dans la suite, vous remplacerez la commande `./gradlew` par juste `gradle`.

II) Crééez un projet gradle

a) Configuration

Il s'agit en réalité de 2 étapes : demander à gradle de télécharger une version récente de lui-même et de configurer le script `gradlew` qui appelle cette dernière version ; puis d'initialiser le projet à proprement parler.

Les étapes :

1. Créez un répertoire pour votre projet et allez dedans :

```
$ cd $HOME/chemin/vers/mes/projets
$ mkdir monProjet
$ cd monProjet
```

2. Configurez le *wrapper* `gradlew` pour qu'il utilise la dernière version de Gradle :

```
$ gradle wrapper --gradle-version 5.6.2 --distribution-type BIN
```

3. Lancez le *wizard* d'initialisation de votre projet :

```
$ ./gradlew init
```

(`gradlew` téléchargera la version de gradle requise lors de son premier usage).

4. Répondez aux questions. Les réponses apportées dependront du projet que vous créez. Probablement, vous répondrez :
   - type de projet → « application » (ce sera le cas dans ce TP) ou « library ».
   - « implementation language » (en quel langage programmez vous ?) → « Java »
   - « build script DSL » (quel langage utilisera gradle pour sa configuration ?) → « Groovy » [1]
   - « test framework » (choix du cadriciel de test) → « JUnit 4 » (pas d'importance pour ce TP).
   - Nom du projet → ce que vous voulez (choisissez un nom que vous saurez reconnaître !)
   - Nom du *package* de sources : il s'agit du *package* principal de ce projet. Respectez les conventions de Java (tout en minuscules !).

b) Vérifications et essais

1. Prenez ensuite le temps de regarder les dossiers et fichiers créés : (par exemple, exécutez `tree`)

```
.
├── build.gradle
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle
└── src
    ├── main
    │   ├── java
```

1. Le DSL Groovy est plus ancien et probablement mieux documenté pour l'instant ; mais il semble que gradle préfère maintenant Kotlin ; ce choix n'est pas très important, mais pour ce TP, les exemples seront donnés en Groovy.

```
    │   │   └── monapp
    │   │       └── App.java
    │   └── resources
    └── test
        ├── java
        │   └── monapp
        │       └── AppTest.java
        └── resources
```

Quelques remarques : le code de votre projet se situera dans `src/main/java/monapp` (`monapp` étant le nom que j'ai donné à mon *package* principal). Le code de test se trouvera dans `src/test/java/monapp`.

2. Ouvrez et regardez les fichiers `settings.gradle` (vérifiez que c'est dans ce fichier que le nom de votre projet est configuré) et `build.gradle` (repérez l'endroit où le nom de la classe principale à exécuter est renseigné ; repérez aussi la liste des dépendances et constatez que Gradle a ajouté par défaut une dépendance à Google Guava).
3. Vous pouvez ensuite compiler et exécuter le projet par défaut pour vérifier que tout est bon :

```
$ ./gradlew build
BUILD SUCCESSFUL in 12s
7 actionable tasks: 7 executed
$ ./gradlew run

> Task :run
Hello world.

BUILD SUCCESSFUL in 863ms
2 actionable tasks: 1 executed, 1 up-to-date
```

c) Importation dans environnement de développement intégré (EDI/IDE)

Importez maintenant votre projet gradle dans votre IDE préféré [2]. Constatez que cela est effectivement bien prévu par celui-ci, que les tâche gradle sont bien à portée de clic et que les paquetages appartenant aux dépendances déclarées dans `build.gradle` sont bien accessibles. Par exemple, ouvrez le fichier `App.java` depuis le projet importé dans votre IDE et insérez-y, au début, la ligne `import com.google.common.collect.*;`. Vérifiez que l'IDE ne signale pas d'erreur.

III) Mini-projet : gestionnaire d'archive zip

Pour illustrer Gradle, programmons maintenant un petit logiciel : un gestionnaire d'archive zip. À la différence des commandes `zip` et `unzip` sous Linux, nous voulons que notre utilitaire ne s'exécute qu'au travers d'une seule commande, avec différentes options (un peu comme les commandes `tar` et `unrar`).

Pour les besoins de l'explication, appelons notre commande `eclair`. La syntaxe d'appel serait la suivante : `eclair <commande> <arguments>`, où `<commande>` serait :

— "`-c`" : pour créer une archive zip. `<arguments>` serait alors une liste de fichiers de taille au moins 2, dont le premier élément serait le nom de l'archive zip à laquelle ajouter les fichiers nommés par le éléments restants (archive à créer si elle n'existe pas).
Exemple :

2. Cherchez comment faire !

```
$ eclair -c mon_archive.zip fichier1.txt fichier2.jpg fichier3.wav
```

— "`-x`" : pour extraire le contenu une archive zip. `<arguments>` serait alors une liste de taille exactement 2, dont le premier élément serait le nom de l'archive zip à décompresser et le second serait le dossier de destination.
Exemple :

```
$ eclair -x mon_archive.zip $HOME/Documents
```

Par ailleurs si aucune option n'est donnée ou si les arguments passés en ligne de commande sont invalides, la commande n'écrira aucun fichier et se contentera d'afficher une aide consistant en la liste des commandes avec leur syntaxe et leur description.

Pour réaliser ce petit logiciel, vous allez utiliser deux dépendances :

— Apache commons-cli : cette bibliothèque sert à analyser les arguments de la ligne de commande (le paramètre de la méthode `main`) en fonction d'une liste d'option préconfigurées. Cette bibliothèque sait aussi générer le message d'aide à partir de la liste des options. Documentation ici : `https://commons.apache.org/proper/commons-cli/introduction.html`.

— Zip4j : une bibliothèque pour gérer les fichiers zip. Documentation : `https://github.com/srikanth-lingala/zip4j`.

Étapes :

1. créez un projet comme expliqué précédemment
2. éditez le fichier `build.gradle`
3. allez à la section `dependencies`, retirez[3] la dépendance à Google Guava qui avait été mise par défaut (supprimez la ligne)
4. ajoutez les dépendances aux bibliothèques commons-cli et Zip4j. Pour cela, insérez les lignes suivantes à la section `dependencies` :

```
implementation 'net.lingala.zip4j:zip4j:2.1.3'
implementation 'commons-cli:commons-cli:1.4'
```

5. Constatez, à la prochaine exécution de Gradle (essayez avec `./gradlew build`), que Gradle télécharge bien ces deux bibliothèques.
6. Programmez votre logiciel en utilisant ces bibliothèques, c'est à dire en important des classes (`import`) et des méthodes (`import static`) des paquetages `net.lingala.zip4j` et `org.apache.commons.cli` dans vos fichiers `.java` qui en ont besoin.
7. Constatez que, quand vous compilez (`./gradlew build`) ou exécutez votre projet (`./gradlew run`) via gradle, Java trouve bien ces paquetages, ou bien que l'éditeur de votre IDE ne signale pas d'erreur.
8. Recherchez comment générer un fichier `.jar` pour votre logiciel, à l'aide de Gradle.

Indications :

— Pour passer des arguments à `main` quand vous exécutez votre programme via la tâche `run` de Gradle, vous pouvez utiliser le paramètre `--args`[4] . Exemple :

```
./gradlew run --args="-x uneArchive.zip dest"
```

3. Sinon cherchez à quoi sert Guava. Laissez la dépendance si vous pensez que ça vous servira dans ce projet !
4. Attention : il est possible qu'un bug de `gradlew` empêche actuellement d'échapper des guillemets dans le paramètre de `--args`. Évitez donc de tester votre commande sur des fichiers dont les noms contiennent des espaces.

IV) Objets, classes, encapsulation

Exercice 1 : Encapsulation et sûreté

Voici deux classes avec leur spécification. Pour chaque cas :
— soit la spécification est satisfaite par la classe, dans ce cas, justifiez-le ;
— soit la spécification n'est pas satisfaite, dans ce cas écrivez un programme qui la met en défaut (sans modifier la classe fournie), puis proposez une rectification de la classe.

1.
```java
public class EvenNumbersGenerator {
    static int MAX = 42;
    public int previous = 0;
    public int next() {
        previous += 2; previous %= MAX;
        return previous;
    }
}
```

Spécification : la méthode `next` ne retourne que des entiers pairs.

2.
```java
public class VectAdditioner {
    private Point sum = new Point();

    public void add(Point p) {
        sum.x += p.x; sum.y += p.y;
    }

    public Point getSum() {
        return sum;
    }
}
```

Spécification : la méthode `getSum` retourne la somme de tous les vecteurs qui ont été passés en paramètre par la méthode `add` depuis l'instanciation.

Exercice 2 : Nombres complexes

Pour le vocabulaire, référez vous à `https://fr.wikipedia.org/wiki/Nombre_complexe`.

1. Écrivez une classe `Complexe`, avec :
   — les attributs (`double`) : parties réelle et imaginaire du nombre (`static` ou pas ?) ;
   — le constructeur, prenant comme paramètres les parties réelle et imaginaire du nombre ;
   — la méthode `public String toString()`, permettant de convertir un complexe en chaîne de caractères lisible par l'humain ;
   — les opérations arithmétiques usuelles (somme, soustraction, multiplication, division) ;
   — le test d'égalité (méthode `public boolean equals(Object other)`) ;
   — les fonctions et accesseurs spécifiques aux complexes : partie réelle, partie imaginaire, conjugaison, module, argument...

   Vous pouvez vous aider des fonctionnalités de génération de code de votre IDE.

   Attention, style demandé : attributs non modifiables (si vous savez le faire, faites une vraie classe immuable), les opérations retournent de nouveaux objets.i
2. Ajoutez à votre classe :
   — des attributs (constants : vous pouvez ajouter `final`) pour les valeurs les plus courantes du type `Complexe` : à savoir 0, 1 et i (le nombre i tel que $i^2 = -1$).
   Ces attributs doivent-ils être `static` ou non ?
   — une méthode (fabrique statique)

   ```java
   public static Complexe fromPolarCoordinates(double rho, double theta)
   ```

   qui construit un complexe depuis son module ρ et son argument θ (on rappelle que la partie réelle vaut alors $\rho \cos\theta$ et la partie imaginaire $\rho \sin\theta$).
   Remarquez que cette méthode joue le rôle d'un constructeur. Pourquoi ne pas avoir fait plutôt un autre constructeur alors ? (essayez de compiler avec 2 constructeurs puis expliquez pourquoi ça ne marche pas)

3. Améliorez l'encapsulation de votre classe, afin de permettre des évolutions ultérieures sans « casser » les clients/dépendents de celle-ci : en l'occurrence, les attributs doivent être privés et des accesseurs publics [5] doivent être ajoutés pour que la classe reste utilisable.
4. Testez en écrivant un programme (méthode `main()`, dans une autre classe), qui fait entrer à l'utilisateur une séquence de nombre complexes et calcule leur somme et leur produit. Améliorez le programme pour permettre la saisie des nombres au choix, via leurs parties réelles et imaginaires ou via leurs coordonnées polaires.
5. Écrivez une version « mutable » de cette classe (il faut donc des méthodes `set` pour chacune des propriétés).

   Changez la signature [6] et le comportement des méthodes des opérations arithmétiques afin que le résultat soit enregistré dans l'objet courant (`this`), plutôt que retourné.

5. Remarquez qu'il n'y a pas de raison de favoriser le couple parties réelle/imaginaire par rapport au couple module/argument (les deux définissent de façon unique un nombre complexe) ; et qu'il faut donc considérer ce dernier couple comme un couple de propriétés, pour lequel il faudrait utiliser aussi la notation `get`. Ainsi, cette classe aurait 4 propriétés (peu importe si elles sont redondantes : les attributs ne le sont pas ; cela limite les risques d'incohérences).

6. Si la méthode déjà programmée est `static` rendre non statique et enlever un paramètre ; dans tous les cas retourner `void`.