# TP4: introduction à Gradle et JUnit

## Introduction

Les objectifs de ce TP sont de se familiariser avec un "build automation tool" (qu'on pourrait traduire par quelque chose comme "outil de gestion automatique de projets" en français) et d'intégrer des tests unitaires sur la base du projet de RPG que nous avons commencé dans les TPs précédents. Il existe différents ' 'build automation tools" pour Java tels que Maven ou Ant. Le concept est similaire au "makefile" (commande *Make*) utilisés pour automatiser la compilation et l'édition des liens, etc. pour le langage C par exemple. Dans le cadre de ce TP, nous allons utiliser **Gradle** ( https://gradle.org/ ).

Nous allons ensuite mettre en place des *tests unitaires* (unit tests) pour notre projet. De manière générale, il est bien connu que tester régulièrement les programmes qu'on écrit est une bonne pratique. Les tests dits "unitaires" ont pour but de tester les plus petites "unités" possibles de notre programme (e.g., une fonction) de manière à réduire le risque de bugs par la suite. Pour notre projet Java, nous allons utiliser **JUnit 5** ( https://junit.org/junit5/ ) qui est un framework open-source pour écrire et exécuter de tels tests.

## Gradle

### Installation

Sur les plateformes Unix, Gradle peut être installé facilement via la commande `sdk`.

0. Ouvrir un terminal.
1. Installer SDKMAN: `curl -s "https://get.sdkman.io" | bash`.
2. Dans le même shell, tapez: `source "/home/od/.sdkman/bin/sdkman-init.sh"`.
3. Enfin, installer Gradle avec: `sdk install gradle 6.8.3`.

## Conventions

Gradle privilégie la notion de convention par rapport à des systèmes tels que Ant, par exemple, où il faut tout déclarer explicitement. De plus, Gradle utilise des "domain specific language" (DSL) tel que Groovy ou Kotlin, par opposition à Maven ou Ant qui utilisent des fichiers XML. Cette dernière approche (XML), peut rendre la lecture des fichiers de configuration très difficile et assez désagréable. La syntaxe proposée par Gradle via les DSL est plus "légère".

Cependant, il faut prendre garde à respecter des conventions prédéfinies si on veut avoir des scripts de configurations les plus légers possibles. Il est toutefois possible de changer les conventions.

Exemple de conventions que nous allons utiliser pour ce TP:

- le code source (.java) va être placé dans le répertoire: `../main/src/java/`.
- les code des tests unitaires dans: `../test/java/`.

En respectant cela, nous n'aurons pas besoin de spécifier où se trouvent ces fichiers et Gradle pourra les trouver automatiquement grâce à ses conventions.

Gradle est basé sur le concept de *tasks* (tâches). Ce sont des blocs de code (un peu comme des fonctions) que vous définissez dans votre fichier .gradle. Vous trouverez un exemple d'une task `helloWorld` dans le fichier `build.gradle` fourni avec le projet-squelette sur Moodle pour le TP4. Cependant, nous vous demandons de ne pas définir vos propres *tasks* pour ce TP et nous vous demandons d'utiliser l'approche basée sur les conventions et le fichier `build.gradle`.

Nous aurons aussi besoin de la notion de *plugin*. Ces derniers sont une manière d'étendre les possibilités de Gradle. Ils permettent de mettre à disposition plus de *tasks* pour ce dernier. Notamment, nous utiliserons:

```
1 apply plugin: 'java'
2 apply plugin: 'application'
3 apply plugin: 'org.junit.platform.gradle.plugin'
```

Remarquez aussi que nous utiliseront le *repository* `mavenCentral()` qui est "pré-défini" (built-in) dans l'installation Gradle (i.e., les URLs sont définies pour nous et nous n'avons pas besoin de nous en préoccupper).

Notez aussi le mot-clef `dependencies` dans le fichier build.gradle. Dans ce bloc, nous définissons les librairies dont nous avons besoin au moment de la compilation et de l'éxécution comme par exemple: `'org.junit.jupiter:junit-jupiter-api:5.0.1'`

## Build

Dans cette section, nous allons construire (*build*) notre projet avec Gradle. Nous supposons que vous avez téléchargé le projet fourni sur Moodle ou que vous avez copié le fichier `build.gradle` à la racine de votre projet et y avez apporté les modifications appropriées, à savoir le nom de votre main class (variable mainClassName) et avez respecté les conventions mentionnées à la section précédente.

Pour ce TP vous pouvez commener à utiliser des IDE. Cependant, lorsque vous aurez des **TP notés** ils devront **obligatoirement** pourvoir être *builder* depuis le fichier `build.gradle`. Par conséquent, il est donc nécessaire de savoir générer votre projet via la ligne de commande:

- placez vous à la racine du projet (là ou se trouve build.gradle)
- tapez la commande `gradle -i build`
- observez l'output affiché dans la console et essayez de comprendre à quoi correspondent les différents affichages, en particuler ceux liés aux *tests*

Vous pouvez ensuite utiliser un IDE et comparer.

**Remarque:** l'option `-i` en ligne de commande, permet d'avoir plus de "verbosité", i.e., d'afficher les détails de ce qui se passe lors du build avec Gradle. Vous pouvez aussi lancer la même commande sans le `-i` et comparer.

# JUnit

Cette section donne un bref apperçu de l'implémentation de tests unitaires avec le framework *JUnit 5*. Comme mentionner en introduction, JUnit permet de réaliser des tests unitaires en Java pour vérifier le bon fonctionnement d'une petite partie de notre projet. Pour se simplifier la vie, nous allons respecter les conventions de Gradle et créer une classe de test dans `../src/test` (si vous avez téléchargé le squelette, ceci a déjà été fait pour vous). La classe s'appelle `RPGTest` et intègre déjà plusieurs méthodes de test. Vous pouvez remarquer les annotations `@` devant les différentes méthodes.

- les méthodes dans lesquelles nous écrirons effectivement nos tests sont annotées `@Test`.
- les annotations `@BeforeEach` et `@AfterEach` servent à indiquer qu'on veut que ces méthodes soient exécutées avant ou après chaque test (comme leur noms l'indiquent).

- les annotations `@BeforeAll` et `@AfterAll` servent à indiquer des méthodes à exécuter après ou avant tout (nous n'en avons pas forcément besoin).
- la ligne (commentée): `@TestInstance(TestInstance.Lifecycle.PER_METHOD);` sert à indiquer le mode de test "par méthode" ou "par class" (`PER_CLASS`).

En effet, par **défaut**, le comportement est `PER_METHOD`, ce qui signifie que JUnit crée une nouvelle instance de chaque classe de test avant d'éxécuter chaque méthode de test de manière à les éxécuter "de manière isolée" en quelque sorte et ainsi d'éviter des effets de bord possible lié à la modification d'état d'une instance par une autre par exemple. On peut modifier ce comportement et demander à JUnit d'exécuter toutes les méthodes de test sur les mêmes instances avec le mot-clef `PER_CLASS`.

# Exercice 1

## Entrées/sorties (I/O)

Dans cet exercice, nous vous demandons de reprendre votre projet RPG du TP3 (ou si vous ne l'avez pas fait, d'utiliser la solution fournie sur Moodle: dans la section TP4).

Nous vous demandons de créer un/des fichier(s) texte contenant les informations suivantes:

- nom du personnage
- max HP
- types d'armures pour les layers above et below
- somme d'or initiale

c'est à dire les informations permettant l'instanciation d'un personnage. Vous êtes libres de gérer la manière dont vous allez instancier les classes ProtectionStack etc. basé sur le texte utilisé pour les champs correspondants à l'armure, i.e., vous devez définir des conventions. Par exemple "cuir" ou "mailles" pour simplifier et la valeur de la protection.

Par exemple:

```
name: Lancelot
maxHP: 100
belowType: leather
belowPhyProt: 20
belowMagicalProt: 0
```

```
belowFireProt: 0
belowElecProt: 0
aboveType: chainMail
abovePhyProt: 55
aboveMagicalProt: 0
...
```

Vous devrez ensuite lire ce fichier text grâce aux librairies `java.nio` ou `java.io` et le "parser" afin d'utiliser les valeurs pour instancier correctement un personnage à partir de là.

**Remarque:** ceci est un exemple et vous être libres d'utiliser vos propres conventions à partir du moment où vous arrivez à parser votre fichier texte après l'avoir lu, et instancer un personnage à partir de là.

# Exercice 2

Implémenter une classe "Bag" (sac à dos) qui permette au joueur de stocker des objects. Vous aurez besoin d'inventer aussi quelque classes (au moins 2) qui représentent des objets comme des potions ou des armes.

Vous être libre d'implémenter le sac à dos comme bon vous semble, mais vous devez utiliser les *collections* (`java.util.Collection`). Par exemple, vous pouvez considérer le sac comme une pile LIFO (Last In First Out). (On serait alors obliger de dépiler jusqu'à trouver l'objet recherché et repiler le reste).

# Exercice 3

Implémeter des tests unitaires pour les classes/méthodes que vous jugez les plus importantes. Il s'agit ici d'explorer un peu les tests. Vous êtes libre de définir des tests qui vous semblent intéressants.

Vous aurez besoin d'utiliser les *assertions*. L'idée générale est de suivre l'approche suivante, "modèle" ou "pattern" AAA[1]: "Arrange, Act, Assert":

- phase "Arrange" (fixture): instanciez la/les classe(s) que vous vouler tester.
- phase "Act": appeler la méthode à tester ou les méthodes utiles pour vous mettre dans la configuration/l'état de test souhaité
- phase "Assertion": faites une assertion (vous pouvez commencer par vous baser sur l'exemple fourni dans `../src/test/`)

[1]Ref: https://java-design-patterns.com/patterns/arrange-act-assert/

La documentation de JUnit 5 vous sera utile: https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/package-summary.html.

# Rendu

Ce TP n'est pas noté. Néanmoins, nous vous demandons de le déposer sur Moodle via le widget prévu à cet effet. Archivez votre projet tout entier dans **un seul** fichier .zip avec la convention de nommage suivante: **prenomNomTP4.zip**.