Programmation orientée objet (B2)

Le cours de programmation orientée objet de deuxième année

- Introduction
- Exceptions
- Moteur de production (gradle)
- Dernière fonctionalités utiles du JDK 17
- Doublures de test
- Logging

Introduction

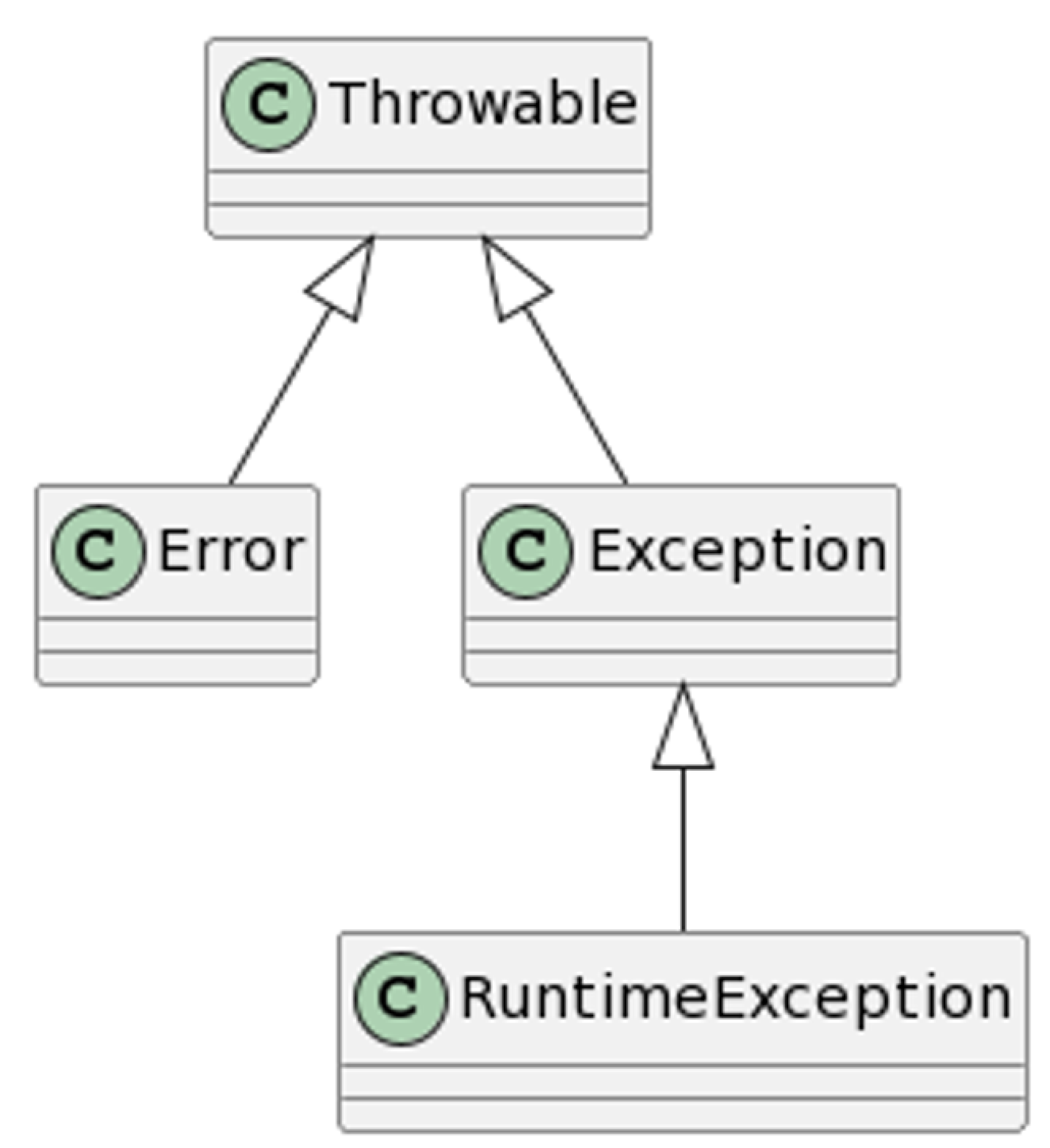
Logiciels

- Gradle (obligatoire)
- Un éditeur de code (au choix), par exemple Eclipse, IntelliJ ou VS Code
- Java JDK 17

Contenu du cours

- Exceptions
- Tests
- Nouveautés Java
- Évènements
- Types génériques
- Réflexion?

Exceptions



On va surtout utiliser Exception et Runtime Exception, mais pas directement (on va plus tot utiliser des exceptions dérivées de ceux-là, tel que NullPointerException ou IllegalArgumentException).

Par défault, le comportement d'une exception est de print l'erreur dans la console, cependant on ne va pas tellement l'utiliser en cours car on va utiliser un système de logging plus élaboré.

Try - catch

```
try {
    // On essaye d'exécuter un certain code ici...
} catch (NullPointerException e) {
    // Le code ici s'exécutera si il y a une NullPointerException et l'erreur sera mise dans
la variable e.
} catch (IllegalArgumentException | IndexOutOfBoundException e) {
    // Le code ici s'exécutera dans le cas d'un IllegalArgumentException ou d'une
IndexOutOfBoundException et sera mise dans la variable e.
}
// Si aucun problème n'est arrivé ou qu'elle a été bien catchée, le code ici s'exécutera...
```

Les try-catch permettent d'exécuer son propre code dans le cas d'une erreur.

Réexécution de fonction

Si on veut réexécuter une fonction il vaut mieux éviter de rappeller simplement la fonction récursivement dans le `catch` car si une erreur persiste, cela augmente grandement le stack des fonctions appellée ce qui peut mener à faire crash le programme.

Il vaut mieux utiliser un do-while

```
// On initialise les variables en dehors de la boucle
boolean locked = false;
int entier = 0;
do {
    System.out.print("Entrer un entier : ");
    try {
        entier = lireEntier();
        // Si le code a fonctionné, on remet le locked à false pour sortir de la boucle
        // On doit le remettre à false car si elle a raté la première fois, le locked aura été
mis à true par le catch
        locked = false;
    } catch {
        // Si un soucis survient, on met la variable locked à true, pour que la boucle
réexécute le code
        // Il n'y a ainsi aucune récursion, donc pas de risque de stackoverflow
        locked = true;
```

```
    }
} while (locked);
```

Throw - throws

```
public class PersonalException extends Exception {
    // Ici on met le code de l'exception, par exemple on peut y mettre des messages d'erreurs,
des fonctions spéciales, etc.
}

// La méthode suivante retourne une exception
public static void method() {
    // Si quelque chose ne fonctionne pas on peut retourner notre exception custom
    throw new PersonalException();
}
```

On peut créer nos propre exceptions pour des cas particulier de nos programmes en étendant la classe Exception puis en utilisant `throw new` pour l'appeller.

Finally

```
try {
    // On essaye d'exécuter un certain code
} catch (NullPointerException e) {
    // On exécute le code ici si le code dans le try ne fonctionne pas
} finally {
    // Quoi qu'il arrive, le code ici sera exécuté, même si le catch retourne une exception.
}
// Si une erreur arrive dans catch ou finally, le code ici ne sera pas exécuté
```

Lorsque l'on veut qu'un code s'exécute quoi qu'il arrive, on peut utiliser le bloc `finally`, ainsi même si l'un des `catch` retourne une exception, on exécutera quand même le bloc finally.

Tester les exceptions

On peut également tester des exceptions avec `assertThrows`

```
assertThrows(RunTimeException.class, ()=>maMethode());
```

On passe directement la méthode à `assertThrows`. La syntaxe étrange s'appelle une lambda, c'est une fonction anonyme temporaire qui exécute la méthode à tester. Cette *lambda* est nécessaire, sinon on passe le résultat de `maMethode` à la place de passer la méthode elle même. La fonction assertThrows va ensuite tester pour voir si une exeception est émise par la méthode, et si oui, elle va tester que l'exception est bien de la classe `RunTimeException`.

Moteur de production (gradle)

L'objectif est d'automatiser les actions pour produire un logiciel, gérer les dépendances, les détecter et adapter la production du logiciel à la platforme. Gradle est un système qui permet d'automatiser tout ça.

Cela permet donc de rendre un projet indépendant de l'environement de développement de la personne qui écrit le code, car Gradle va automatiquement gérer toutes les dépendences nécessaires.

Historique des moteurs de production

Année	Programme	Langage	Avantage
1977	make	C et autres	
2000	Apache Ant	Java	
2004	Maven	Java	Plus complet que Apache Ant
2008	Gradle	Java (et autres comme Kotlin)	Plus complet que Maven

Que font Gradle et Maven

- La compilation
- Le packaging (jar, war, etc)
- Gestion des dépendences
- Génération de la documentation
- Gestionnaire de sources
- Accès au depot des gestinnaires des dépendences
- Le déploiement en différents environements (test, développement, production, etc)

Qu’est ce que Gradle

Gradle est un moteur de production tournant sur la JVM (Java Virtual Machine). Un moteur de production sert à automatiser les étapes de construction d’un projet. Un moteur de production est nécessaire car le faire à la main serait source d’erreur, très lent et complexe. Gradle au fur et à mesure du temps est devenu très important dans l’écosystème Java.

Pour aller plus loin que les informations du cours, on peut aller consulter la documentation officielle de Gradle.

Gestion simplifiée des dépendences

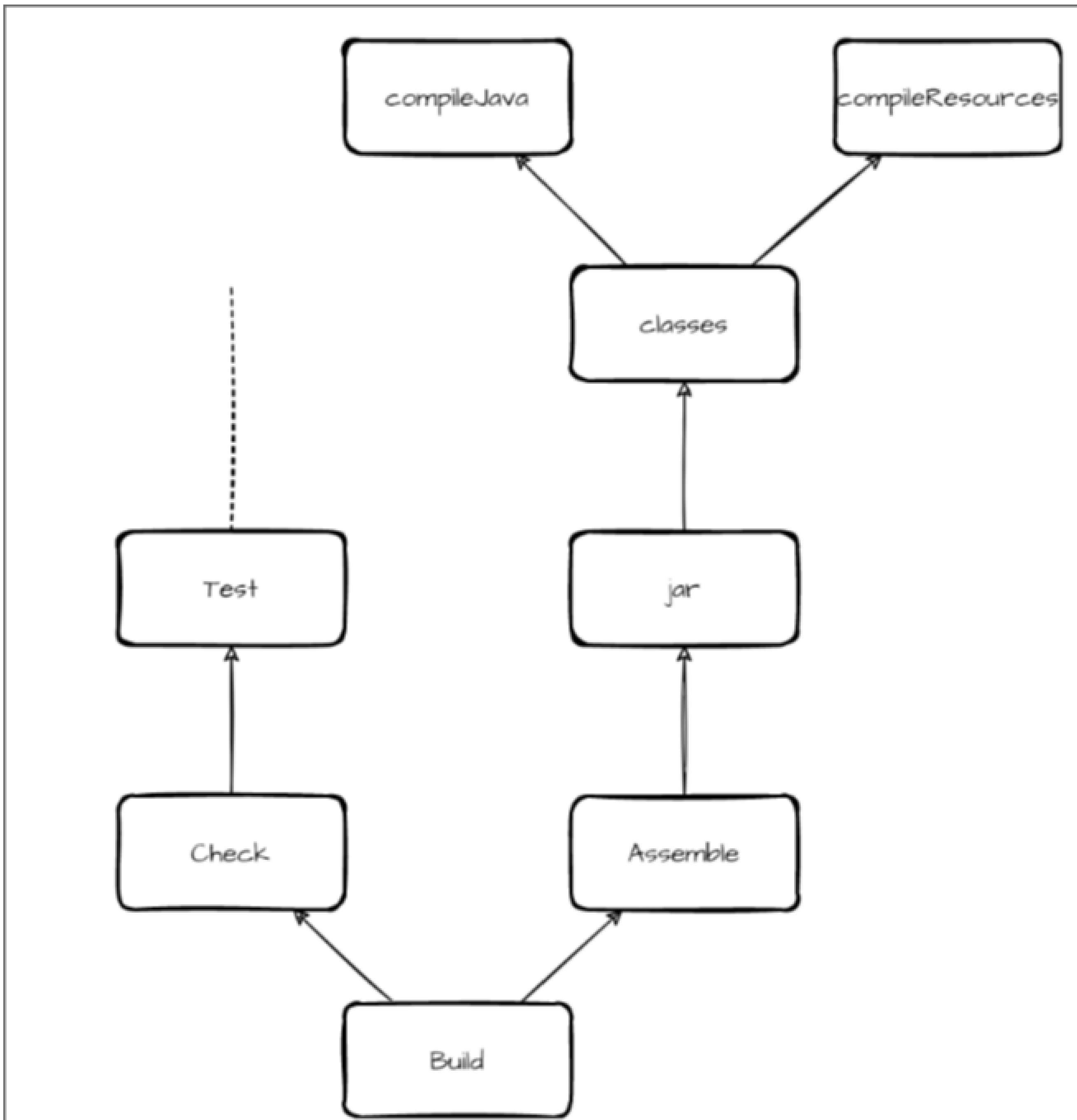
Gradle permet par exemple une gestion simplifiée des dépendances qui permet de simplement installer beaucoup de dépendences depuis les répertoires Maven (un autre moteur de production).

On peut en savoir plus en consultant le site MVN repository.

Les tâches Gradle

Gradle fonctionne sur base de tâches. Chaque tâche permet d'obtenir des sorites (fichiers ou répertoires mis à jour) à partir d'entrées (fichiers, répertoires, configuration, etc).

Les tâches peuvent en inclure d'autres, par exemple la tâche `build` d'un projet Java standard inclus la tache `check` qui va effectuer les tests, et la tâche `assemble` qui va générer l'exécutable.



Système de plugin

De base Gradle est très simple et peu utilisable en pratique. Pour rendre Gradle vraiment utile il faut y intégrer des plugins, comme le plugin `java` par exemple qui permet de compiler des applications Java. Ce système rends donc Gradle très flexible car il peut être beaucoup étendu et n'importe qui peut en créer des plugins

Processus d'une constructoin de projet Gradle

1. L'**initiation** met en place l'environement et détermine les projets qui le compose
2. La **configuration** construit et configure le DAG des tâches (l'arbre qui définit les liens entre les tâches) et définit les tâches à exécuter pour accomplir la tache initiale

3. L’**exécution** exécute les tâches identifiées

La seule phase dans laquelle on intervient est la phase de configuration.

Avantage de Gradle

- Plus rapide
- Plus flexible (pas limité à Java, contrairement à Maven)
- Fichier de config et plus simple
- Gestion des dépendances plus complète
- Indépendant
- Multilangage

Configuration de Gradle

- `settings.gradle` qui est à la racine du projet et contient le nom du projet ainsi que l'ensemble des sous-projets
- `build.gradle` est dans chaque (sous-)projet et contient l'ensemble des éléments utile pour compiler le projet (version de Java, dépendance, , tests, PMD, etc)

Liste des librairies

On peut rechercher les libraries sur le site *MVN Repository*.

Qu'est ce qu'une librairie

On ne s'amuse pas à réinventer la roue dès que l'on veut faire un programme. Du coup on va réutiliser des composants qui ont déjà été fait par d'autres personnes. Une librarie c'est exactement ça, c'est une collection d'outils qui permettent de programmer plus facilement et plus rapidement sans devoir réinveter les même choses en boucle.

Identification d'une dépendence/librarie

Il y a 3 éléments qui identifie une dépendence :

- Le *group id* qui défini le groupe des librairies
- Le *artifact id* qui défini la dépendence dans le groupe
- La *version* de la dépendence

Il est important de vérifier la version des dépendences qui ne sont pas toujours compatibles entre elles et il faut éviter de prendre des versions trop vieilles.

Installation de Gradle

Suivez les instructions sur le site officiel de Gradle.

Création d'un projet et autopsie

Pour créer un projet vous pouvez simplement aller créer un dossier vide, puis entrer dedans avec votre terminal et lancer la commande : `gradle init`

Après quoi Gradle va vous poser certaines questions, pour l'exemple voici ce que vous devez répondre :

- Project type ? → 2 (application)
- Implementation language ? → 3 (java)
- Build script DSL ? 2 (groovy)
- Test framework ? 4 (JUnit Jupiter)

Pour toutes les autres questions, faites juste ENTER pour choisir l'option par défault

Autopsie des tâches

Une fois cela fait nous pouvons ensuite avoir une liste des tâches possibles. Pour cela il suffit d'exécuter `./gradlew tasks`.

Ainsi on voit que si on fait `./gradlew run` ça va lancer le projet (par défault ça va afficher Hello World) ou encore si on fait `./gradlew check` ça va effectuer tous les tests.

Autopsie de la structure des fichiers

```
.
├── app
│   ├── build.gradle
│   └── src
│       ├── main
```

```
|       |   ├── java
|       |   |   └── hello
|       |   |       └── App.java
|       |   └── resources
|       └── test
|           ├── java
|           |   └── hello
|           |       └── AppTest.java
|           └── resources
├── .gitattributes
├── .gitignore
├── .gradle
|   └── file-system.probe
├── gradle
|   └── wrapper
|       ├── gradle-wrapper.jar
|       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
└── settings.gradle

13 directories, 11 files
```

- `app/` est le dossier qui contient tout ce qui est relatif à votre projet (en particulier le code source)
  - `app/build.gradle` contient les informations sur le build du projet
  - `app/src/` contient le code source de l'application
    - `app/src/main/` et `app/src/test/` contiennent les deux partie du projet (main pour le code principal et test pour les test unitaires), ces dossiers sont ensuite divisé par langage de programmation (ici Java uniquement), puis par package. Par exemple le App.java se situe dans le package Java test.gradle.
- `.gradle/` est le cache du projet
- `gradle/` contient le *wrapper* de Gradle, c'est grâce à cela que va être automatiquement téléchargée la version Gradle appropriée si on viendrait à essayer de lancer le projet sans avoir Gradle installé ou sans avoir la bonne version de Gradle installée pour le projet.
- `gradlew` et `gradlew.bat` sont les scripts (un pour Linux/macOS et un pour Windows) qui vont exécuter le wrapper gradle pour télécharger la bonne version automatiquement. Il est recommander de les utiliser à la place de la commande `gradle` même si les deux fonctionnent en vérité.
- `settings.gradle` défini quels sont les projets qui font partie du dossier ainsi que d'éventuels autres plugins.
- `.gitignore` défini les fichiers à ignorer dans git

- `.gitattributes` défini les préférences du projet pour git

Autopsie des fichiers de configuration

Vous pouvez maintenant aller ouvrir les fichiers `app/build.gradle` et `settings.gradle` pour découvrir ce qu'il se passe à l'intérieur.

Build.gradle

Le fichier `build.gradle` indique :

- La liste des plugins (ici, uniquement "application")
- La sources des dépendances
- Liste des dépendances
- La définition de la version de java pour paramétrer le plugin "application"
- La définition de la classe principale de l'application (ici test.gradle.App)
- La configuration de la tache de test

Settings.gradle

Le fichier `settings.gradle` indique :

- Une liste de plugins, ici il y a uniquement un plugin servant à télécharger automatiquement des versions du JDK
- Le nom du projet global
- La liste des sous projets (ici il n'y en a qu'un c'est `app`)

Intégration avec Eclipse

Importer un projet Gradle

On peut par exemple importer notre projet Gradle créé plus tot dans Eclipse. Pour cela on peut aller dans Eclipse > File > Import > Gradle > Existing Gradle Project. Ensuite on peut sélectioner le dossier de notre projet Gradle. Enfin dans les "Import Options", on peut cocher la case "Override workspace settings" et définir le Java home à la localisation du JDK aproprié. Une fois cela fait on peut cliquer sur "Finish" pour importer le projet. Pour être sur que tout a bien charger on peut faire un clic droit sur le projet puis aller dans Gradle puis dans "Refresh Gradle project"

Lancer une tâche Gradle

Pour lancer une tache ou voir la liste des tâches on peut aller dans l'onglet “Gradle tasks”, si l'onglet n'est pas affiché on peut l'afficher en faisant Window > Show view > Other > Gradle Tasks.

Ensuite il suffit de cliquer sur les tâches que l'on veut exécuter, on peut ensuite voir son résultat dans l'onglet Console.

A savoir qu'Eclipse génère tout un tas de fichiers qui ne sont pas intéressants à ajouter dans Git, il vaut donc mieux les ajouter au `.gitignore`

```
# Backupfiles from mergetools #
*.orig

# Java Class Files #
*.class

# Package Files #
*.jar
*.war
*.ear
*.zip

# Eclipse #
.project
.settings
.classpath
bin
```

Ajouts de plugins et dépendences supplémentaires

Il y a certain plugins Gradle que l'on est obligé d'avoir pour l'activité intégrative. Il faut en installer 2 :

- PMD qui fournit des rapports sur les potentielles faiblesses de notre code
- JaCoCo qui fournit un rapport sur le code coverage (la couverture de code couverte par les tests)

PMD (dans Gradle)

Pour installer le plugin PMD dans Gradle, on va tout d'abord l'ajouter dans la liste des plugins du `build.gradle` du dossier `app`.

```
plugins {
    id 'application'
    id 'pmd' // ← Ajout de cette ligne
}
```

Ensuite on peut ajouter le fichier ruleset de pmd présent sur l'espace de cours dans le projet. Une fois cela fait, on peut modifier de nouveau le build.gradle pour y ajouter la configuration de PMD :

```
pmd {
    ruleSets = []
    ruleSetFiles = files("pmd-ruleset.xml") // ← mettre le chemin de fichier relatif vers le
fichier PMD ici
    maxFailures = 15 // Défini la limite acceptable d'erreurs PMD avant de stopper le build
}
```

Une fois cela fait PMD va afficher des erreurs dans la console lors du build si l'une de ses règles n'est pas respectée. Et au delà 5 erreurs, PMD va faire stopper le build.

PMD génère aussi un rapport dans le dossier `app/build/reports/pmd/`.

PMD dans Eclipse

Pour avoir les erreurs affichées directement dans l'IDE quand on écrit le code on peut activer le plugin Eclipse comme en B1.

Installation

Pour cela on peut aller dans Help > Eclipse Marketplace > eclipse-pmd > Install.

Ensuite il faut accepter la license et autoriser toutes les sources du plugin quand demandé.

Une fois le plugin installé, il va vous demander de redémarrer Eclipse.

Configuration

Ensuite pour l'activer sur notre projet, on peut aller dans les propriétés du projet (clic droit sur le projet > Properties) puis aller dans l'onglet “PMD” et choisir l'option “Enable PMD for this project”.

Une fois cela fait on peut cliquer sur Add > Project > Browse puis sélectionner le fichier ruleset et cliquer sur Finish.

Une fois cela fait on a maintenant les alertes directement dans le code.

Jacoco

Pareil que PMD on doit d'abord ajouter Jacoco dans la liste des plugins :

```
plugins {
    id 'application'
    id 'pmd'
    id 'jacoco' // ← Ajout de cette ligne
}
```

Ensuite on peut le configurer, ici on va le configurer comme dit dans sa documentation en le faisant exécuter à chaque test (un report Jacoco sera ainsi généré à chaque fois que les tests seront effectués) :

```
test {
    finalizedBy jacocoTestReport // report is always generated after tests run
}

jacocoTestReport {
    dependsOn test // tests are required to run before generating the report
}
```

Une fois cela terminé, on peut maintenant lancer la tâche `check` ou `test` et un report jacoco devrait être généré dans le dossier `app/build/reports/jacoco`

Gestion des dépendences

Chaque dépendence dans Gradle a une certaine portée, ainsi certaines dépendencent ont une portée uniquement sur les tests unitaires et d'autres sont nécessaires pour le code principal.

Par exemple si on veut installer la dépdance `Apache Commons Text`. On peut aller rechercher le nom de la dépendance sur le site MVN repository. Ensuite on peut cliquer sur la version qui nous intéresse (ici 1.10.0) puis cliquer sur l'onget "Gradle (short)" pour savoir ce que nous devons ajouter dans la section `dependencies`

Par défault le site MVN repository va proposer de l'installer pour `implementation`, cependant on pourrait très bien choisir `api` ou `testImplementation` :

- `implementation` est la portée requise pour compiler la source de production du projet qui ne font pas partie de l'API exposée par le projet.
- `api` est la portée requise pour compiler la source de production du projet qui font partie de l'API exposée par le projet
- `testImplementation` est la portée requise pour compiler et exécuter la source de test du projet. Par exemple, le projet a décidé d'écrire le code de test avec le cadre de test JUnit.

Vous pouvez trouver plus d'information sur ce sujet sur la page consacrée à la gestion de dépendences Java de Gradle.

Projets modulaires avec Gradle

Gradle permet de décomposer le code en différents modules ce qui permet de maintenir le code plus facilement, ainsi que de le rendre plus robuste et portable.

- Pour l'activer on peut créer un projet avec `gradle init` et activer l'option pour y créer des sous-projets.
- Modifier le projet pour y ajouter des sous-projets

Lorsque l'on crée un projet modulaire avec gradle init, la librairie de test automatiquement choisie est JUnit Jupyter

Structure des projets modulaires

Les projets modulaires ont une structure plus complexe que les projets non-modulaire car ils ont des sous-projets supplémentaires que `app`. Par défault quand on init un projet Gradle avec des sous-projets, on va avoir les dossiers `app`, `buildSrc`, `list` et `utilities` qui vont être créés.

- Le dossier `app` par convention sert toujours pour le code principal.
- Le dossier `buildSrc` sert à créer des plugins ou tâches Gradle spécifiques
- Les dossiers `list` et `utilities` sont juste là à titres d'exemples. Vous pouvez par exemple avoir un sous-projet pour une librarie utilisée par d'autres sous projets, ou encore un sous-projet pour une application CLI et une autre pour une application Android.

Dernière fonctionalités utiles du JDK 17

> Je n’ai ici gardé que les plus importants changements à utiliser.

Depuis JDK 8

Lambdas et interfaces fonctionnelles

Les lambdas qui sont des fonctions anonymes stoquées dans des variables. Et les interfaces fonctionnelles sont des interfaces n’ayant qu’une seule méthode pour une lambda.

Grâce à cela, on peut avoir des types de fonctions plus précis et rendre le code plus sûr.

Voici un exemple d'interface fonctionnelle :

```
// Définition d'une interface foncitonnelle "MyFunction" ayant une fonction prenant en argument 2 int et en retournant 1
@FunctionalInterface
interface MyFunction {
    int apply(int x, int y);
}

// On peut ensuite créer une lambda suivant cette interface
// Le type est donc MyFunction, les arguments sont x et y et le corps de la méthode est après la flèche
MyFunction add = (x, y) -> x + y;

// On peut ensuite utiliser notre fonction
System.out.println(add.apply(1,1)); // Ceci va afficher "2"
```

Default methods in interfaces

Maintenant on peut écrire un code par défault dans les interfaces (rendant donc l'implémentation de ces méthodes par défault optionnelle).

```
public interface Vehicle {
    // Ceci sont des méthodes d'interface normales, il est donc obligatoire de les écrire pour implémenter l'interface
    String getBrand();
    String speedUp();
    String slowDown();

    // Ceci sont des méthodes ayant un code par défault, il n'est donc pas obligatoire de les écrire pour implémenter l'interface
    default String turnAlarmOn() {
        return "Turning the vehicle alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }
}
```

Date and time API

Avant Java 8 il n'y avais pas d'API par défault pour gérer le temps et les dates. Maintenant il y en a une. Vous pouvez avoir tous les détails dans [la Javadoc de java.time].

A partir de Java 8 on peut simplement récupérer la date et l'heure actuelle en faisant `LocalDateTime.now()`.

Stream API

Pour expliquer plus en détails les avantages de la stream API, voici quelques exemples de code avec et sans stream.

Les streams API sont assez important et sont attendu à l'AI.

Vous pouvez apprendre à les utiliser en regardant [la Javadoc de Stream].

Exemple 1

Sans stream api :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
for (int number: numbers) {
    if (number % 2 == 0) {
        System.out.println(number);
    }
}
```

Avec stream api :

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

Exemple 2

Sans stream api :

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
for (String name: names) {
    System.out.println(name.toUpperCase());
}
```

Avec stream api :

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream() // Génère le Stream
    .map(String::toUpperCase) // Passe tout en uppercase
    .forEach(System.out::println); // Print chaque élément dans la console
```

Exemple 3

Imaginons que l'on a une liste de valeurs en nombre et en String et que l'on veut faire la moyenne tout en passant le premier élément :

Sans la stream API :

```
public static double getAverage(List<String> values) {
    double sum = 0.0;                                             // Commence à 0
    final int counter = values.size();                            // Compte le nombre d'items
dans la liste
```

```
    if (list.isEmpty()) return 0;                              // Retourne 0 si la liste est
vide pour empécher une division par 0
    for (int i = 1; i < values.size(); i++) {                  // Fait une boucle passant le
premier item
        String stringValue = value.get(i);                     // Récupère la valeur en
String
        double numericValue = Double.parseDouble(stringValue);  // La converti en nombre
        sum += numericValue;                                    // L'ajoute à la somme
    }
    return sum / counter;                                       // Fait le calcul de la somme
}
```

Avec la stream API :

```
public static double getAverage(List<String> values) {
    return values.stream()                                  // Converti la liste en Stream
        .skip(1)                                            // Passer le premier élément
        .mapToDouble(value -> Double.parseDouble(value)) // Tout convertir en double
        .average()                                          // Caluler la moyenne de tout
        .orElse(0.0);                                       // Si la liste est vide, va retourner
0
}
```

Depuis JDK 9

Listes immuables

On peut maintenant créer des listes (ou d’autres collections) immuables en Java avec `of` ou `copyOf` :

```
List<String> list = List.of("Hello", "World");
```

Depuis JDK 10-11

Inférence de type

Depuis Java 10 ou 11 on peut maintenant utiliser l'inffférence de type, on est donc plus obligé de préciser le type d'une variable locale. On peut simplement utiliser le mot clé `var`

```
// Avant Java 10
String message = "Hello, World!";

// Possible après Java 10
var message = "Hello, World!";
```

JDK 17

Sealed classes

Les *sealed classes* permettent de limiter les classes qui peuvent hériter de la classe actuelle. Il n'est pas recommandé d'utiliser ceci pour la même raison qu'il n'est pas recommandé d'utiliser l'héritage.

```
// Seule les classes Cricle et Square pourront hériter de Shape
public sealed class Shape permits Circle, Square {
}
```

Switch expressions

Les switch peuvent maintenant fonctionner avec n'importe quel type (et pas uniquement int comme avant), de plus on peut en écrire plusieurs sur la même ligne :

```
int month = 2;
int daysInMonth = switch (month) {
    case 1, 3, 5, 7, 8, 10, 12 -> 31;
    case 4, 6, 9, 11 -> 30;
    case 2 -> 28;
    default -> throw new IllegalArgumentException("Invalid month: " + month);
};
```

Records

Les records permettent de créer très simplement des data class pour représenter certains concepts et les stoquer.

Voici comment représenter une Personne avec un nom et un age avant les Records :

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

Et voici comment représenter à peu près la même chose avec un Record

```
record Person(String name, int age) {}
```

Text blocks

Depuis JDK 17 on peut aussi simplement faire des Strings de plusieurs lignes avec `"""`.

Voici comment faire un String de 3 lignes avant JDK 17 :

```
String message = "Welcome\n How are you?\n I hope you have a nice day";
```

Et voici comment faire la même chose depuis JDK 17 :

```
String message = """
                 Welcome
                 How are you?
                 I hope you have a nice day
                 """;
```

Doublures de test

Introduction

Les doublures de tests permettent d'isoler les classes à tester et de briser les intéraction entre elles. Les doublures de tests ne remplace pas JUnit et permet de tester les appels que la classe va faire aux autres classes.

Par exemple, imaginons que l'on a une classe `Messagerie` qui peut lire les messages d'un·e utilisateur·ice sur base de son identifiant et mot de passe. Pour vérifier l'utliisateur·ice, la Messagerie va faire appel à la classe `Identification`. Cependant ici on ne veut tester uniquement Messagerie mais pas la classe d'Identification.

Pour avoir plus d'explications vous pouvez aller voir [cet article] (le code en exemple n'y est pas forcément de la meilleure des qualités mais c'est pratique pour comprendre le principe des doublures).

Pourquoi utiliser une doublure ?

Il y a plusieurs raisons pour laquelle on voudrait mettre en place une doublure :

- La classe testée fait appel à un composant dificile ou couteux à mettre en place (par exemple une base de donnée, c'est nottament le cas pour notre classe `Identification` dans l'exemple)
- Le test vise à vérifier le comportement d'une classe dans une situation exceptionnelle (imaginons, une déconnexion réseau, on ne vas pas réellement déconnecter la machine juste pour faire un test)
- La classe testée fait appel à un composant qui n'existe pas encore ou qui n'est pas suffisament stable (cela permet par exemple de tester notre classe `Messagerie` alors que la classe `Identification` dont elle dépends n'existe pas encore)
- Le test fait appel à du code lent (par exemple, si notre Identification prends un certain temps, cela ralentirait grandement les tests pour rien)
- Le test fait appel à du code non déterministe (par exemple à l'heure ou à l'aléatoire. Par exemple si une classe ferait appel à une classe générant des nombres entre 1 et 6, nos tests serait faux 5 fois sur 6)
- Séparer le code de test du code de l'application (par exemple si on crée une méthode `fakeGenerateNumber()` dans une classe aléatoire, cela complique les choses pour rien)

Types de doublures

Les stub objects

Un stub est simplement une classe écrite à la main spécialement pour le contexte du test. On sait quelle valeurs sont attendues d'avance et on les hardcode dans la classe.

Par exemple, dans le cas de l'idenficiation de et de la messagerie on peut avoir une interface `Identification` et créer une classe `IdentificationStub` implémentant cette interface de façon à hardcoder les valeurs attendues pour le test (par exemple) :

```
public class IdentificationStub implements Identification {
    boolean identify(String username, String password) {
        // Renvois true si l'identifiant est 'toto' et le mot de passe 'mdp'
        return "toto".equals(username) && "mdp".equals(password)
    }
}
```

Pour le test il suffit alors simplement d'injecter la classe `IdentificationStub` dans le constructeur de la classe `Messagerie`.

```
public class MessagerieTest {
    @Test
    void testLireMessages() {
        // On injecte le stub dans la classe à tester
        Messagerie messagerie = new Messagerie(new IdentificationStub());

        // On fait les tests comme on le souhaite dessus...
    }
}
```

Les fake objects

Un fake est une doublure écrite à la main qui implémente le comportement attendu mais de façon plus simple que la classe réelle. Contrairement au stub qui est écrit spécialement pour un test précis, le fake a vocation à être suffisament générique pour être utilisé dans plusieurs tests. Il est donc plus complexe que le stub mais plus réutilisable.

Pour reprendre l'exemple précédent on peut imaginer une classe `IdentificationFake` qui implémente `Identification` mais qui a une méthode supplémentaire `addAccount(String username, String password)` permettant de personaliser le test.

```
public class IdentificationFake implements Identification {
    Map<String, String> comptes = new HashMap<String, String>();

        @Override
        public boolean identify(String username, String password) {
            // Vérifie que l'identifiant et le mot de passe soit dans la liste des comptes
            return comptes.containsKey(identifiant) &&
comptes.get(identifiant).equals(password);
        }

        public void addAccount(String username, String password) {
            // Ajoute un nouvel identifiant-mot de passe dans la fausse liste des comptes
            comptes.put(username, password);
        }
}
```

Comme pour le stub on peut donc aller l'injecter dans le constructeur lors du test, à la différence qu'ici on peut l'utiliser pour plusieurs tests différents et le configurer différemment à chaque fois.

```
public class MessagerieTest {
    private Identification identification = new IdentificationFake();

    @Test
    void testLireMessages() {
        // On configure notre fake object
        identification.addAccount("toto", "mdp");

        // On peut ensuite l'injecter dans notre classe à tester
        Messagerie messagerie = new Messagerie(identification);

        // Enfin on peut faire nos tests comme on le souhaite...
    }

    // on peut ensuite faire d'autres tests sur le même principe sans avoir à créer plusieurs
classes pour chaque cas
}
```

Les dummy objects

Les dummy sont le type de doublure le plus simple, ce sont simplement des classes implémentant l'interface attendue mais ne faisant absolument rien car ils ne sont jamais vraiment utilisés.

Par exemple si on teste un cas précis où l'identification n'est jamais utilisée on peut créer une classe dummy implémentant `Identification` et qui renverrai toujours la même valeur (car quelque soit la valeur on s'en fout puis ce qu'elle ne sera pas utilisée) :

```
public class IdentificationDummy implements Identification {
    @Override
    public boolean identify(String username, String password) {
        return true;
    }
}
```

Ici le code du test se fait exactement comme pour le stub

```
public class MessagerieTest {
    @Test
    void testLireMessages() {
        // On injecte le stub dans la classe à tester
        Messagerie messagerie = new Messagerie(new IdentificationDummy());

        // On fait les tests comme on le souhaite dessus...
    }
}
```

Les mock objects

Les mocks objects sont plus complexes mais plus flexibles que les autres et c'est ceux là que l'on va privilèger pour l'activité intégrative en utilisant la librarie Mockito.

Contrairement aux autres, les mocks sont générés par une librarie, on a donc pas besoin de créer la classe nous même, il suffit juste de dire dans notre test que l'on souhaite créer un Mock et quelle valeur on veut que certaines méthodes retournent.

Ainsi les Mocks ont la simplicité des Fake objects mais sans avoir à créer la moindre classe soi-même.

Pour l'exemple précédent à la place de créer tout une classe on a simplement à définir ceci dans le test :

```java
public class MessagerieTest {
    // On demande à Mockito de créer un mock pour nous
    @Mock private Identification identification;

    @Test
    void testLireMessages() {
        // On configure le mock pour lui dire les paramètres et réponses attendues
        when(identification.identify("toto", "mdp")).thenReturn(true);

        // On l'injecte dans le constructeur de la messagerie
        Messagerie messagerie = new Messagerie(identification);

        // Enfin on peut faire les tests sur la messagerie comme on le souhaite...
    }

    // On peut ensuite réutiliser notre mock de la même façon pour d'autres tests
}
```

Libraries

Il existe 2 librairies principales pour faire du *mocking* en Java, mais ici c'est Mockito qui a été privilégié.

Mockito

- facile d'utilisation
- configuration via annotation simple
- très grande communauté
- Choisi pour le cours

La documentation de Mockito est assez affreuse mais au moins elle est là, vous pouvez retrouver quelques liens intéressants sur leur site, ainsi que leur documentation officielle.

EasyMock

- Facile d'utilisation
- Configuration simple mais plus chiant que l'autre
- Moins utilisé que mockito

Spy objects

Les spy objects permettent de vérifier qu'une méthode à été appellée, de savoir combien de fois et avec quels arguments. Pour reprendre l'exemple précédent, si on imagine que la méthode `identify` est void, et ne retourne donc rien; on pourra tout de même la tester en vérifiant qu'elle a

bien été appellée avec les bons arguments.

Cela peut être fait dans un Fake object mais est beaucoup plus compliqué à mettre en place, cela est en revanche trivial à faire avec Mockito :

```
public class MessagerieTest {
    @Mock private Identification identification;
    @Test
    void testLireMessages() {
        // On injecte la méthode dans le constructeur de la messagerie
        Messagerie messagerie = new Messagerie(identification);

        // On fait nos tests...

        // On peut ensuite par exemple aller vérifier que la méthode ~identify~ a été appellée
exactement une fois avec les paramètres "toto" et "mdp":
        verify(identification, times(1)).identify("toto", "mdp");
    }

}
```

De plus Mockito permet également d'espioner de vrais objets.

```
public class MessagerieTest {
    @Spy private Identification identification = new RealIdentification();

    @Test
    void testLireMessages() {
        // On injecte la classe espion dans la messagerie
        Messagerie messagerie = new Messagerie(identification);

        // On fait nos tests

        // On peut vérifier que l'identification a bien été appellée :
        verify(identification, times(1)).identify("toto", "mdp");

        // Note, si on le souhaite on pourrait même stub les méthodes de la vrai classe en
faisant when().thenReturn() par exemple
    }
}
```

Logging

Le logging permet de débugger plus simplement les application avec plus de finesse qu'avec `System.out.println`, cela permet nottament de filtrer les logs selon le type (DEBUG, INFO, etc) ainsi que rediriger le flux des logs dans des fichiers.

JUL

JUL est la classe de log par défault dans Java, elle peut être importée depuis `java.util.logging`.

Log4j2

Log4J 2 est le successeur de Logback qui lui même est le succèsseur de log4j. Plus personne n'utilise (ou n'est sensé utiliser) log4j à cause de très très gros soucis de sécurité.

Cette librairie n'est pas inclue de base dans Java mais est disponible dans la dépendence `org.apache.logging.log4j:log4j-code:2.20.0`.

Log4j2 est également configuré avec de l'XML ou avec un fichier properties.