POO : API & Outils
TD et TP

Thibaut Smith
thibaut.smith@soprasteria.com
thibaut.smith-fisseau@univ-lemans.fr

18 décembre 2019

Les objectifs du cours (5 TD + 6 TP) :

- Prendre connaissance des nombreuses API Java
- Appréhender des concepts objets
- Utiliser un environnement de développement Java
- Savoir faire des tests unitaires
- Savoir utiliser les outils du développeur (maven, git, easymock)
- Un TP final noté devoir-maison
- Un DS
- Un TP noté (à confirmer)

Table des matières

1 TD

1.1 Utilisation d'Eclipse

Étape 1
Présentation du logiciel

Étape 2
Création d'un projet Java

Étape 3
Paramétrage des options

(a) Configuration éditeur

(b) Paramétrage plugins

(c) Paramétrage JDK

(d) Paramétrage Maven

Étape 4
Création d'un projet Maven

(a) Choisir "sans archetype"

(b) Ajouter une classe dans le package "src/main/java" contenant une méthode *public static void main(String[] args)* faisant un *System.out.println("Hello")*

(c) Vérifiez que votre projet se lance avec la commande "mvn clean compile".

Étape 5
Configurez la compilation du projet via votre IDE (Eclipse/IntelliJ, etc) et vérifier que vous arrivez à voir le message dans la console.

Étape 6
Développer une classe et l'instancier dans votre fonction *main*.

Étape 7
Configuration d'un debugger

Étape 8
Debug :

(a) Ajouts de points d'arrêts ("breakpoints") pour indiquer au debugger où s'arrêter.

(b) Vérifiez que le debugger s'arrête bien là où vous les avez posés.

Étape 9
Développement de classes et de tests unitaires

(a) Ajoutez un package *communication*

(b) Ajoutez-y une classe *Menu* permettant de demander quelques questions à l'utilisateur via *System.out*

(c) Récupérez les réponses via *System.in*

Étape 10
Vérifiez que le plugin Eclemma est déjà installé : il permet d'analyser la couverture de code.

Étape 11
Lancer la couverture de code.

1.2 Refactoring, utilisation Maven, et système de contrôle de versions

Étape 1

Principe du refactoring

Étape 2

Quelles parties du code pouvons-nous factoriser sur cet exemple ?

```
ArrayList<Integer> list1 = new ArrayList<Integer>();

list1.add(1);
list1.add(2);
list1.add(3);
list1.add(4);

ArrayList<Integer> list2 = new ArrayList<Integer>();

list2.add(5);
list2.add(6);
list2.add(7);
list2.add(8);

ArrayList<Integer> list3 = new ArrayList<Integer>();

list3.add(9);
list3.add(10);
list3.add(11);
list3.add(12);

int index = 0;
for(Integer value : list1) {
    System.out.println("list1.get(" + index++ + ") = "  +
       value);
}

index = 0;
for(Integer value : list2) {
    System.out.println("list2.get(" + index++ + ") = "  +
       value);
}

index = 0;
for(Integer value : list3) {
    System.out.println("list3.get(" + index++ + ") = "  +
       value);
}
```

https://e-gitlab.univ-lemans.fr/snippets/10

Étape 3

Tests unitaires et refactoring : TDD

Étape 4

Utilisation de Maven

(a) Cycle de vie des dépendances

(b) Commandes clean, compile, test, install, package, deploy

(c) Trouver les dépendances

Étape 5

Rappel des outils de versionnement de code source (SCM)

Étape 6

Stockez votre projet sur Gitlab ou GitHub

1.3 Modélisation UML

On souhaite développer une application permettant à deux personnes de jouer une partie d'échec en même temps sur le même ordinateur. Nous allons modéliser un programme permettant de jouer aux jeux d'échecs, utilisant ces règles-ci :

- Le jeu d'échecs se joue à deux joueurs qui font évoluer seize pièces chacun, respectivement blanches et noires, sur un échiquier de 64 cases en alternance blanches et noires (8 colonnes et 8 rangées).
- Jouer un coup consiste à effectuer un déplacement de l'une de ses pièces, accompagné éventuellement de la capture d'une pièce adverse se trouvant sur la case d'arrivée de la pièce jouée.
- Blancs et Noirs jouent à tour de rôle, par convention les Blancs jouent le premier coup de la partie.
- On ne peut pas passer son tour.
- Les bords de l'échiquier sont infranchissables par les pièces.
- Aucune pièce ne peut venir occuper une case déjà occupée par une pièce de son propre camp.
- Une pièce amie ou une pièce adverse qui se trouve sur la même colonne, rangée ou diagonale qu'une Dame, une Tour, ou un Fou constitue un rempart au-delà duquel ces pièces à longue portée ne peuvent sauter.
- Chaque joueur possède initialement : un roi, une dame, deux fous, deux cavaliers, deux tours et huit pions.

Étape 1

Faites le diagramme de classe de l'application.

(a) Quelles classes vont être nécessaires ?

(b) Chaque pièce possède un type de mouvement différent. Comment modéliser les particularités de chaque pièce ?

(c) Modéliser toutes les pièces :

1. Le roi
2. La reine
3. Les fous
4. Les tours
5. Les cavaliers
6. Les pions

(d) Modéliser la couleur de chaque joueur.

(e) Mises à part les classes liées aux règles du jeu, nous avons aussi besoin de représenter les deux joueurs, les pièces qui ont été capturées, et l'état de l'échiquier.

Étape 2

Comment faire pour vérifier la validité d'un mouvement fait par le joueur ?

Étape 3

Faites le diagramme de séquence du mouvement d'une pièce par un joueur. Note : une pièce peut être prise au cours du mouvement.

Étape 4

Nous souhaitons ajouter une fonctionnalité de sauvegarde de la partie pour pouvoir reprendre ultérieurement. Ajouter au modèle actuel les méthodes qui s'occuperont de la sérialisation/dé-sérialisation des objets.

1.4 Questions d'examens des années précédentes

Étape 1

Examen 2018-2019 : écrire la surcharge du constructeur de la classe Voiture pour initialiser l'immatriculation en même temps que la voiture.

```
public class Voiture {
  private String immatriculation;
  public String getImmatriculation(){
    return this.immatriculation;
  }
  public void setImmatriculation(String immatriculation){
    this.immatriculation = immatriculation;
  }
  private String couleur;
  public String getCouleur(){
    return this.couleur;
  }
  public void setCouleur(String couleur){
    this.couleur = couleur;
  }
  public Voiture(String couleur) {
    this.setCouleur(couleur);
  }
}
```

Étape 2

Examen 2013-2014 : écrire une interface Comparable qui permet de comparer n'importe quelle classe ?

Étape 3

Examen 2013-2014 : À quoi servent les annotations en Java (donner plusieurs cas d'utilisations) ?

Étape 4

Ecrire une classe d'exception personnalisé NullArgumentException avec une méthode toString qui précise le nom de l'argument qui est null de la manière suivante : « L'argument X est null » avec le nom de l'argument X initialisé lors de l'instanciation de la classe.

Étape 5

Examen 2013-2014 : On désire modéliser des comptes en banque. Chaque compte est caractérisé par un numéro qui doit être unique et par le solde (la quantité d'argent sur le compte).

1. Définir une classe **Account** avec au moins deux constructeurs. Ajouter des méthodes *put* et *get* pour ajouter et retirer de l'argent. La méthode *get* doit faire en sorte que le solde ne devienne jamais négatif et retourne la somme qui a été effectivement retirée. Ajouter une méthode balance qui retourne le solde du compte.
2. Définir une nouvelle classe **Premium** pour un compte autorisant un découvert (solde négatif). Le découvert ne doit pas dépasser une valeur propre au compte qui peut être modifié par une méthode *decouvertAutorise*.
3. Définir une classe Bank qui peut contenir des comptes. Implémenter des méthodes *createAccount* et *createPremium* permettant respectivement de créer un compte et de créer un compte premium. Ces méthodes retournent l'identifiant du compte.

4. Écrire des méthodes *put* et *get* prenant en paramètre un numéro de compte et un montant et permettant d'ajouter et de retirer de l'argent. Comment faire pour que la méthode get prenne en compte le caractère premium du compte ?
5. Écrire une méthode *balance* qui donne le solde de la banque, c'est-à-dire, la somme des soldes de ses comptes.

Étape 6

Examen 2012-2013 : À quoi sert la méthode de Mocking (Mock) ?

1.5 Lecture et rédaction d'API

Dans le TP final, nous allons créer une application Java permettant les fonctionnalités suivantes :

- Utilisation de Spring Framework pour ajouter les fonctionnalités de base de données, utilisation d'API REST en ligne, génération de pages web ;
- Utilisation de H2 pour créer une base de données *in-memory* ;
- Utilisation de Thymeleaf pour faire des pages web ;
- **Récupération d'une adresse via un formulaire pour ensuite appeler une API donnant les coordonnées GPS** ;
- **Utilisation d'une API web pour récupérer la météo aux coordonnées GPS précises**.

Nous allons nous concentrer sur les API web dans ce TD (en gras dans la liste ci-dessus).

Étape 1

Retour sur les basiques du web : HTTP

- HTTP GET
- HTTP POST
- Entêtes
- Exemple d'une API : SpaceX API, PokeAPI, RandomUser.me

Étape 2

Pour obtenir des coordonnées GPS à partir d'une adresse, nous allons utiliser l'**API Adresse** de Etalab, un service gratuit fournit par le gouvernement : comment cet API fonctionne-t-elle ? Lien : `https://geo.api.gouv.fr/adresse`

Étape 3

Pour obtenir la météo à un lieu indiqué par des coordonnées GPS, nous allons utiliser l'API DarkSky : comment cet API fonctionne-t-elle ? Lien : `https://darksky.net/dev/docs`

Étape 4

Notre application Java Spring va consommer ces deux API pour nous permettre de traduire une adresse textuelle en données météos. Écrire l'API correspondant au service que vous avez prévu de remplir : adresse vers données météos

Étape 5

Ajoutez-y les données HTTP à l'API pour qu'un programmeur sache comment l'utiliser. (Rappelez-vous de l'objectif du TP : on demande l'adresse via un formulaire = méthode HTTP POST)

Étape 6

Codez la partie HTML de votre formulaire pour faire un appel HTTP POST. Le rendu doit être un formulaire contenant un champ adresse, son label, et un bouton pour valider le formulaire.

2 TP

2.1 Outillage de l'intégration continue (GitHub, Travis CI, CodeClimate, ReadTheDocs, Gitlab)

Nous allons utiliser la plate-forme d'hébergement de codes sources GitHub possédant un support impressionnant d'outillages pour tester/analyser/améliorer le code. Nous allons voir comment utiliser GitHub pour développer une plate-forme d'intégration continue.

Voici l'objectif du TP :

- Créer un projet Maven générique. Cela va créer les fichiers de configuration nécessaires pour un projet Java. Nous pourrons ensuite initialiser un projet git pour commencer à suivre les modifications du projet. (1h)
- Une fois le projet créé et le projet Git initialisé, on va commencer à développer le programme Java en ajoutant des classes et des tests unitaires. (1h)
- Nous allons finir le TP avec la configuration de plusieurs outils d'analyses de code sources. (1h)

Vous obtiendrez à la fin plusieurs métriques :

- des indicateurs de qualité de code avec *CodeClimate*,
- des résultats des tests unitaires avec *Travis CI*,
- une génération de la documentation de votre projet avec *Read The Docs*.

Étape 1

Vous allez commencer par créer un premier projet Java avec Maven. Utilisez Eclipse pour initialiser votre projet Maven :

- File, New, Maven Project,
- Choisissez "Create a simple project (skip archertype selection)",
- Remplissez la partie GAV du projet Maven (Group, Artifact, Version),
- Et cliquez sur "Finish" pour le créer.

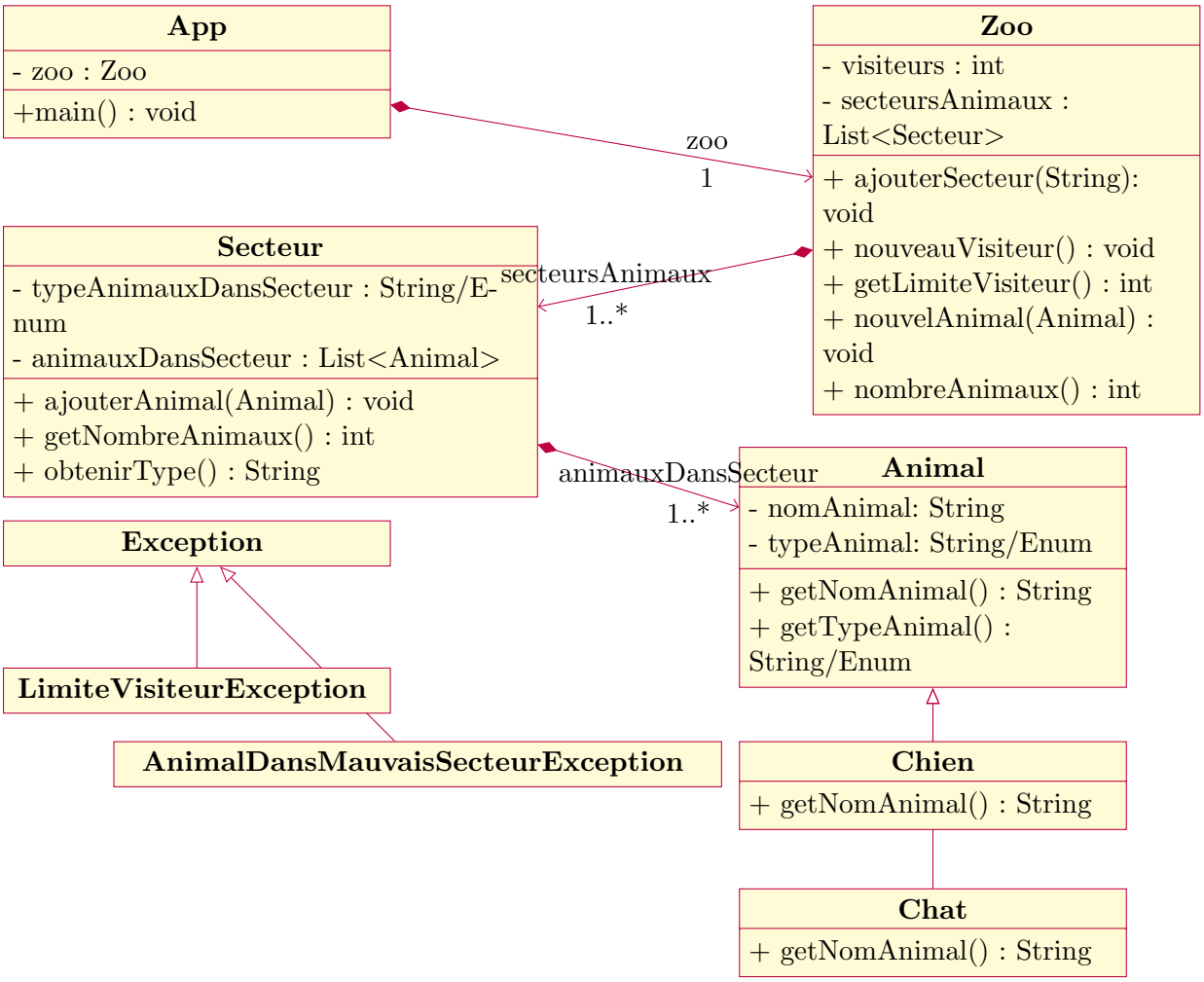
Étape 2

Maintenant que le projet est créé, vous pouvez développer un petit programme Java.

Nous allons programmer une modélisation basique d'un zoo, voici ce que nous allons modéliser :

- Notre zoo contient des secteurs d'animaux, et quelques espèces d'animaux : chien, chat (par exemple).
- Ajouter des animaux au Zoo et les ranger dans leur secteur correspondant.
- Compter un nouveau visiteur dans le Zoo.
- Compter le nombre de visiteurs dans le Zoo.
- Avoir une exception si on dépasse le nombre de visiteurs dans le zoo : 15 visiteurs par secteurs maximum au sein du Zoo.

Voici la modélisation à développer :



Étape 3

Créer le test unitaire suivant : si on dépasse le nombre maximum de visiteurs dans le zoo, une exception doit être lancée.

Étape 4

Créer le test unitaire suivant : si on ajoute un chien dans le zoo, il doit être stocké dans le bon secteur.

Étape 5

Dans le fichier pom.xml de votre projet, utiliser la dépendance suivante pour les tests unitaires :

```
<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Étape 6

Lancez vos tests via Eclipse et via Maven avec la commande “mvn test”.

Étape 7

Créez ou utilisez votre compte GitHub pour créer un nouveau repository.

Étape 8

Créez un nouveau repository GitHub, et configurer **par exemple** GitKraken/Sublime-Merge/Git en ligne de commande ajoutez-y votre projet source.

Un proxy pour utiliser git en ligne de commande est parfois nécessaire (chaque encadré est une seule commande, ne pas sauter à la ligne après le "http.proxy") :

```
$ git config --global http.proxy
   http://proxy.univ-lemans.fr:3128
```

```
$ git config --global https.proxy
   https://proxy.univ-lemans.fr:3128
```

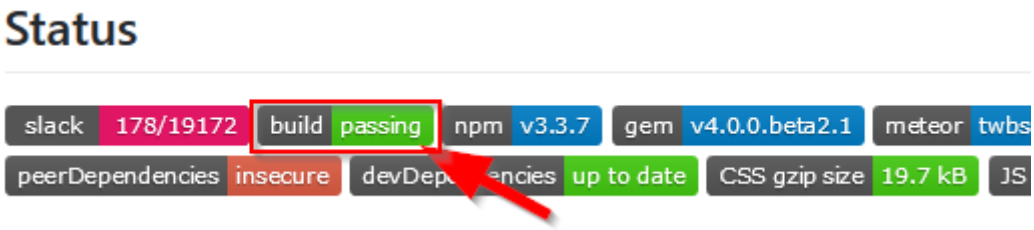
Étape 9

Cherchez à quoi sert **Travis CI** et configurez-le pour votre projet. Il a besoin d'un fichier de configuration *.travis.yml* pour analyser votre code.

Étape 10

La page d'accueil d'un repository GitHub montre par défaut le fichier *README.md*. Créez-en un s'il n'a pas déjà été rajouté par défaut : beaucoup d'exemples sont disponibles sur les autres projets opensource sur GitHub.

Regardez comment utiliser le badge du *build status* généré par **Travis CI** et incorporez-le à votre *README.md* pour afficher un statut en temps réel de la validation de votre projet. (Exemple : `https://github.com/twbs/bootstrap`, toutes les icônes affichées dans le *README.md* sont des icônes du même type que ceux fournis par **Travis CI**.)



Étape 11

Cherchez à quoi sert **CodeClimate** et configurez-le pour fonctionner avec votre projet GitHub.

Étape 12

Créer le test unitaire suivant : si on ajoute un chien dans le zoo, le nombre d'animaux doit être correct. Y'a t'il un changement sur CodeClimate ?

Étape 13

Cherchez à quoi sert **ReadTheDocs** et configurez-le pour fonctionner avec votre projet.

Étape 14

Ajoutez un nouveau test unitaire et validez son exécution sur **Travis CI**.

Étape 15

Observez les indicateurs de **CodeClimate** et enlevez quelques *code smells* (= du code de mauvaise qualité entraînant une maintenance plus longue).

Étape 16

Allez voir la doc générée par **ReadTheDocs**, améliorez-la sur votre projet et observez la mise-à-jour sur **ReadTheDocs**.

Étape 17

Dans le fichier *README.md* de votre projet, ajoutez un lien pour pouvoir accéder au badge et pour accéder aux projets sur **Travis CI**, **CodeClimate**, et **ReadTheDocs**.

Étape 18

Une fois tout configuré, envoyez-moi l'URL (thibaut.smith@soprasteria.com) de votre projet GitHub par mail.

Étape 19

(Info : GitHub et d'autres partenaires proposent plein d'outils professionnels exceptionnellement gratuit pour les étudiants : `https://education.github.com/pack`.)

Étape 20

Prenez en compte les remarques de **CodeClimate** pour améliorer votre projet.

Étape 21

Bonus : Hébergez votre projet sous GitLab et trouvez un équivalent à Travis CI sur Gitlab, et essayez de le faire fonctionner.

— Créer un nouveau projet Gitlab
— Ajouter un nouveau 'remote' à votre projet Git
— Poussez vos modifications sur le projet Gitlab

2.2 TP DM noté 1/2 : Réalisation d'une application web avec Spring Boot

Ce TP est noté !

- Vous pouvez le terminer chez vous ;
- A rendre via : UMTICE (cours : POO API et outillages, clé : PAAP4) ;
- Date limite : 31 décembre.

Vous allez réaliser une application à l'aide de Java et des technologies suivantes :

- Spring `https://spring.io/`
- Spring Boot `https://spring.io/projects/spring-boot`
- JPA `https://www.tutorialspoint.com/jpa/index.htm`
- Hibernate `http://hibernate.org/`
- H2 `http://www.h2database.com/html/main.html`
- Spring Data JPA `https://spring.io/projects/spring-data-jpa`
- Thymeleaf `https://www.thymeleaf.org/`

Nous allons utiliser beaucoup d'API, n'hésitez pas à chercher sur internet des indications et de la documentation. Suivez les étapes tranquillement en vérifiant que l'application fonctionne toujours.

Étape 1

Commencez par générer votre application sur le site de Spring, avec Spring Initializr : `https://start.spring.io/`

Étape 2

Vous allez générer un projet Maven, avec Java, et Spring Boot 2.2.1. Dans le groupe, ajoutez le nom du groupe Maven du projet que vous souhaitez créer. Pour l'artifact, choisissez "tp5".

Étape 3

Ajoutez les dépendances suivantes :

- Web
- JPA
- Hibernate
- H2
- DevTools
- Thymeleaf

Étape 4

Générez votre projet.

Étape 5

Cherchez une description succincte de chaque dépendance ajoutée pour trouver à quoi sert-elle, et ajoutez-y ces informations dans un fichier "README.md" à la racine de votre projet.

Étape 6

Lancez votre IDE, configurez le proxy si nécessaire, et importez le nouveau projet (Vérifiez les aides plus bas dans le document si besoin).

Étape 7

Pour lancer une application Spring Boot, vous pouvez utiliser la commande maven suivante :

```
mvn spring-boot:run
```

Créez une nouvelle "Run Configuration" de type Maven et dans le goal, utilisez la commande ci-dessus. Lancez la nouvelle configuration et regardez ce qui est inscrit dans le terminal pour voir si tout est OK.

Étape 8

Vous devriez avoir le message suivant dans le terminal :

```
Tomcat started on port(s): 8080 (http) with context path ''
```

Ouvrez la page `http://localhost:8080`.

Si vous avez un message d'erreur, cela signifie peut-être que le port 8080 est déjà utilisé. Vous pouvez ajouter cette configuration dans le fichier *application.properties* pour démarrer sur le port 9090 :

```
server.port=9090
```

Étape 9

Nous allons maintenant créer notre première page. Commencez par créer les packages "model", et "controller" (Spring respecte le modèle MVC). **Attention** : le contrôleur a besoin de la vue pour fonctionner (pensez-y lors de vos tests).

Étape 10

Dans le package "controller", créez la classe HelloWorldController.java avec le contenu suivant :

```
@Controller
public class HelloWorldController {

    @GetMapping("/greeting")
    public String greeting(@RequestParam(name="nameGET",
       required=false, defaultValue="World") String
       nameGET, Model model) {
        model.addAttribute("nomTemplate", nameGET);
        return "greeting";
    }
}
```

`https://e-gitlab.univ-lemans.fr/snippets/2`

Étape 11

Dans le dossier "resources", créez un dossier "templates" et dedans, le fichier "greeting.html" :

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Blog</title>
    <meta http-equiv="Content-Type" content="text/html;
       charset=UTF-8" />
</head>
<body>
    <p th:text="'Bonjour ' + ${nomTemplate} + ' !'" />
</body>
</html>
```

`https://e-gitlab.univ-lemans.fr/snippets/3`

Étape 12

Relancez votre application avec le launcher et allez sur la page `http://localhost:`

`8080/greeting`. Tentez ensuite l'URL suivante : `http://localhost:8080/greeting?name=ENSIM`

Étape 13

Relisez le code du contrôleur, aidez-vous de documentation sur internet, et répondez aux questions suivantes :

1. Avec quelle partie du code avons-nous paramétré l'url d'appel /greeting ?
2. Avec quelle partie du code avons-nous choisi le fichier HTML à afficher ?
3. Comment envoyons-nous le nom à qui nous disons bonjour avec le second lien ?

Ajoutez les réponses dans le README.

Étape 14

Nous allons maintenant activer la base de données H2. Dans le fichier "application.properties", ajoutez les lignes suivantes :

```
# Enabling H2 Console
spring.h2.console.enabled=true
```

Relancez l'application et allez sur l'URL : `http://localhost:8080/h2-console`, et appuyez sur "Connect". Si vous avez une erreur, essayer l'url suivante :

```
jdbc:h2:mem:testdb
```

Étape 15

Vous êtes sur l'interface de votre base de données **in-memory** !

Étape 16

Nous allons rajouter notre première classe du modèle MVC pour notre site : la classe *Address*. Dans le package "model" créé tout à l'heure, vous pouvez ajouter la classe Address suivante :

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;
    private Date creation;
    private String content;
}
```

`https://e-gitlab.univ-lemans.fr/snippets/4`

Rajoutez les méthodes *get* et *set* de toutes les propriétés via le bon raccourci de votre IDE.

Étape 17

Relancez-votre application, retournez sur la console de H2 : `http://localhost:8080/h2-console`. Avez-vous remarqué une différence ? Ajoutez la réponse dans le README.

Étape 18

Expliquez l'apparition de la nouvelle table en vous aidant de vos cours sur Hibernate, et de la dépendance Hibernate de Spring. Ajoutez la réponse dans le README.

Étape 19

Nous allons utiliser Spring pour ajouter des éléments dans la base de données. Pour ce faire, créer un fichier “data.sql” à côté du fichier “application.properties” et ajoutez-y le contenu suivant :

```
INSERT INTO addresse (id, creation, content) VALUES (1,
   CURRENT_TIMESTAMP(), '57 boulevard demorieux');
INSERT INTO addresse (id, creation, content) VALUES (2,
   CURRENT_TIMESTAMP(), '51 allee du gamay, 34080
   montpellier');
```

`https://e-gitlab.univ-lemans.fr/snippets/5`

Étape 20

Relancez l'application, retournez sur la console H2 : `http://localhost:8080/h2-console`. Faites une requête de type SELECT sur la table Article. Voyez-vous tout le contenu de data.sql ? Ajoutez la réponse dans le README.

Étape 21

Nous allons maintenant créer une interface de type *Repository* pour accéder aux adresses. Une classe *Repository* permet d'accéder aux données de la base de données. Á côté de Adresse, créez une interface nommée “AdresseRepository” avec le code suivant :

```
@Repository
public interface AddressRepository extends
   CrudRepository<Address, Long> {

}
```

`https://e-gitlab.univ-lemans.fr/snippets/6`

Étape 22

Nous allons créer un autre contrôleur pour donner un accès aux adresses via l'url `http://localhost:8080/adresses`. Créez la classe **AddressController.java** dans le package des contrôleurs.

```
@Controller
public class AddressController {

    @Autowired
    AddressRepository addressRepository;

    @GetMapping("/adresses")
    public String showAddresses(Model model) {
        model.addAttribute("allAdresses",
           addressRepository.findAll());
        return "addresses";
    }
}
```

`https://e-gitlab.univ-lemans.fr/snippets/7`

Étape 23

Pouvez-vous trouver à quoi sert l'annotation **@Autowired** du code précédent sur internet ? Ajoutez la réponse dans le README.

Étape 24

Créez de plus le fichier “addresses.html” dans le dossier “templates” de tout à l’heure :

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Adresses</title>
    <meta http-equiv="Content-Type" content="text/html;
       charset=UTF-8" />
</head>
<body>

<h1>Les differentes adresses</h1>

<ul th:each="address: ${allAddresses}">
    <li th:text="${address.content}" />
</ul>

</body>
</html>
```

Étape 25

Vous êtes maintenant prêt et autonome pour la suite !

Étape 26

Et si vous rajoutiez le nom de la personne qui a fait la recherche d’une adresse ?

1. Ajouter l’auteur dans l’entité “Address.java”,
2. Dans le fichier data.sql, ajoutez la donnée,
3. Côté vue, utilisez la nouvelle donnée et affichez-la.

Étape 27

Pouvez-vous rajouter une navbar pour naviguer entre chaque pages ?

Étape 28

Utilisez les *Thymeleaf fragments* pour pouvoir déplacer la déclaration de la navbar à un seul endroit, et l’inclure dans chaque vues.

Étape 29

Regardez sur internet comment ajouter Bootstrap à votre projet.

Étape 30

Expliquez la méthode que vous avez utilisé pour ajouter Bootstrap dans le README.

2.3 Utilisez des API Java (log/listes/comparator)

Étape 1

Sur le TP du Zoo, mettez en place l'API *Comparator*. Elle vous permet de comparer vos propres classes entre elles. Nous allons l'utiliser pour savoir quelle secteur possède le plus d'animaux.

(a) Faites un Comparator permettant de savoir quel secteur a le plus d'animaux.

(b) Testez votre comparateur avec des tests unitaires.

Étape 2

Ajoutez la possibilité d'écrire des logs avec Log4j2. Vous devez ajouter une nouvelle dépendance pour Maven, un fichier de configuration, et utiliser des méthodes pour écrire des logs.

(a) Ajoutez les deux dépendances suivantes dans le pom.xml de votre projet Maven :
- `https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core`
- `https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-api`

(cf. `https://logging.apache.org/log4j/2.x/maven-artifacts.html`).

(b) Ajoutez une configuration pour écrire les messages de log dans la console et dans un fichier. Vous pouvez par exemple utiliser une configuration via fichier XML log4j2.xml. (exemple : chercher "log4j2 configuration" sur Google, mais attention à la version de la documentation). Il faut mettre le fichier de configuration Log4j2 dans le build path de votre programme. Par exemple dans : src/main/resources/.

(c) Utilisez le logger dans le code source comme ceci :
Avec les imports Java suivant :

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
```

Dans les variables de votre classe :

```
private static final Logger logger =
   LogManager.getLogger(VotreClasse.class);
```

Et où vous écrivez une log :

```
logger.info("Nouvel animal : " +
   ani.getClass().toString());
```

(d) Utilisez les différents niveaux de logs : trace, debug, info, warn, error, fatal. Regardez si vous voyez toutes les logs dans la console.

(e) Écrivez les logs dans un fichier de log (en plus de la console).

(f) Écrivez les logs de niveau fatal, error et warn dans la console, toutes les autres devant aller dans un fichier (configuré précédemment).

(g) Avez-vous une idée de pourquoi utiliser une log, plutôt que les Streams ? (Exemple : *System.out*, *System.in*)

Étape 3

Vérifiez la couverture de code via votre IDE.

Étape 4

Faites un nouveau commit Git de votre projet et poussez-le sur GitHub. Regardez l'outil Code Climate pour voir s'il n'y a pas d'anomalies.

Étape 5

Si Code Climate remonte des anomalies, corrigez-les.

2.4 TP DM noté 2/2 : Utilisation d'API web avec Spring Boot

Ce TP est noté !

- Vous pouvez le terminer chez vous ;
- A rendre via : UMTICE (cours : POO API et outillages, clé : PAAP4) ;
- Date limite : 31 décembre.

Ce TP est dans la continuité du précédent, on va ajouter les fonctionnalités prévues dans le TD 5 :

- Utilisation de Spring Framework pour ajouter les fonctionnalités de base de données, utilisation d'API REST en ligne, génération de pages web ;
- Utilisation de H2 pour créer une base de données *in-memory* ;
- Utilisation de Thymeleaf pour faire des pages web ;
- **Récupération d'une adresse via un formulaire pour ensuite appeler une API donnant les coordonnées GPS** ;
- **Utilisation d'une API web pour récupérer la météo aux coordonnées GPS précises**.

Étape 1

Nous allons incorporer une nouvelle page (à l'url "adresse") contenant le formulaire du TD 5, permettant de demander une adresse.

(a) Créez un contrôleur, et ajoutez un formulaire permettant d'insérer l'adresse dans son template.

(b) Mettez à jour la navbar pour qu'un lien aille sur cette nouvelle page "adresse".

Étape 2

Nous allons maintenant travailler sur les interactions entre pages.

(a) Créez un nouveau contrôleur, et son template (vide pour l'instant) pour l'url "meteo".

(b) Cette page "meteo" doit **être la cible** du formulaire de la page "adresse" que vous venez de créer, et sur validation du formulaire, doit envoyer le contenu vers le controller météo.

Modifier les attributs du formulaire précédent pour rediriger vers "meteo".

(c) Dans le contrôleur de la météo, ajoutez une méthode permettant de recevoir des appels POST (indice : annotation *PostMapping*).

(d) Récupérez la donnée du formulaire entrée par l'utilisateur dans le template. Dans la méthode du contrôleur de la météo, trouvez le bon paramètre permettant d'indiquer à Spring de valoriser l'adresse insérée dans le formulaire (aidez-vous de internet et de la documentation Spring).

(e) Une fois que le code compile, vérifiez qu'il fonctionne en mettant un point d'arrêt et en lançant le programme en mode debug.

Étape 3

À partir de l'adresse récupérée de l'utilisateur, faites un appel HTTP GET vers l'API Adresse de Etalab.

(a) Lisez la documentation Etalab Adresse : `https://geo.api.gouv.fr/adresse`

(b) Faites des tests d'URL via le navigateur (l'API fonctionne en GET, donc testable facilement via le navigateur).

(c) Regardez la structure de la réponse donnée par Etalab Adresse, et concevez l'objet Java de réponse (cf. documentation Spring : `https://spring.io/guides/gs/consuming-rest/`).

N'oubliez pas le proxy à éventuellement configurer si vous êtes sur le réseau de l'université : dans la partie "Astuces" dans ce document, il y a une aide sur la configuration du proxy pour Java.

Étape 4

Pour vérifier que vous obtenez bien les informations dans votre application, essayez de l'afficher dans le template Thymeleaf du contrôleur météo, ou via point d'arrêt en mode debug.

Étape 5

Une fois que vous arrivez à obtenir les coordonnées GPS, vous pouvez maintenant appeler l'API de DarkSky. Lisez la documentation pour comprendre comment spécifier la clé API, comment indiquer les coordonnées GPS, et comment est structuré la réponse. N'hésitez pas à faire des tests via votre navigateur.

(a) Lisez la documentation Dark Sky : `https://darksky.net/dev/docs`

(b) Créez un compte pour avoir une clé API.

(c) Faites des tests d'URL via le navigateur (l'API fonctionne en GET, donc testable facilement via le navigateur).

(d) Regardez la structure de la réponse donnée par Dark Sky, et concevez l'objet Java de réponse (cf. documentation Spring)

Étape 6

Mettez la réponse à chacune de ces questions dans le README de votre projet :

— Faut-il une clé API pour appeler DarkSky ?
— Quelle URL appeler ?
— Quelle méthode HTTP utiliser ?
— Comment passer les paramètres d'appels ?
— Où est l'information dont j'ai besoin dans la réponse :
  — Pour afficher la température du lieu visé par les coordonnées GPS
  — Pour afficher la prévision de météo du lieu visé par les coordonnées GPS

Étape 7

Pour rendre le devoir-maison :

(a) Dans le README.md du projet, ajoutez un lien vers votre projet GitHub.

(b) Poussez votre code sur GitHub.

(c) Faites un zip de votre code source (lancez un "mvn clean" pour enlever les fichiers temporaires).

(d) Uploadez le zip sur UMTICE, dans la partie devoir-maison.

2.5 Utilisez une API sur internet (Maven, API)

Nous allons apprendre à utiliser des API internet pour pouvoir interagir avec des services web.

Étape 1

Créez un programme vous permettant de savoir quand sont les prochains jours féries à l'aide d'une API sur la manipulations de date (Joda-Time).

Nous allons de plus utiliser un fichier ICS "référentiel" fourni par Google Calendar et contenant déjà tous les jours fériés français (les fichiers ICS sont des formats de données adaptés aux échanges d'événements de calendriers). Vous aurez besoin de l'API iCal4j pour manipuler ce fichier.

(a) Créer un nouveau projet Maven et ajoutez la dépendance suivante pour manipuler les fichiers ICS (documentation `http://ical4j.github.io/docs/` et des exemples `https://github.com/ical4j/ical4j/wiki/Examples`) :

```
<dependency>
   <groupId>org.mnode.ical4j</groupId>
   <artifactId>ical4j</artifactId>
   <version>1.0.2</version>
   <!-- ou plus grand en fonction de la doc/exemples a
      votre disposition -->
</dependency>
```

(b) Utilisez le fichier ICS à l'adresse suivante : `https://www.google.com/calendar/ical/fr.french%23holiday%40group.v.calendar.google.com/public/basic.ics`.

(c) Stockez les jours fériés dans une liste et retrouver le prochain jour férié le plus proche de la date du jour.

(d) Le fichier des calendriers étant souvent mis à jour, faites en sorte de le télécharger par l'application. Si vous avez besoin de configurer un proxy pour faire un téléchargement, voici comment faire :

```
java.net.Proxy proxyDL = new
   java.net.Proxy(java.net.Proxy.Type.HTTP,new
   InetSocketAddress("proxy.univ-lemans.fr",
   3128));
```

(e) Bonus : trouvez le mois possédant le plus de jours fériés (hors week-end !) en 2019 :-).

Étape 2

Nous allons développer une application Java pour jouer au "Wiki Game" : `https://fr.wikipedia.org/wiki/Wikip%C3%A9dia:Exercices/Course_wikip%C3%A9dienne`.

Le but du jeu est d'arriver à une page "cible" préchoisie en naviguant via les pages Wikipedia.

L'application devra vous permettre de choisir la page de départ et la page d'arrivée. A chaque page, l'application doit fournir la liste des liens présents dans la page à l'utilisateur, parmi lesquels, l'utilisateur pourra en choisir un pour tenter d'arriver à la page d'arrivée.

L'application donnera le nombre d'articles qui a été nécessaire pour atteindre votre page d'arrivée.

(a) Regardez comment utiliser Wikipédia sous forme d'API : `https://www.mediawiki.org/wiki/RESTBase`, `https://en.wikipedia.org/api/rest_v1/`

(b) Regardez comment utiliser une API Rest avec Java. Vous avez plusieurs possibilités : OkHttp `http://square.github.io/okhttp/`, Jersey `https://jersey.github.io/`, RESTEasy, Restlet, URLConnectionn, etc.

(c) Voici le fonctionnement attendu du programme :

1. Enregistrer dans l'application la page de départ et la page d'arrivée.
2. Ouverture de la page de départ.
3. Analyse de l'article récupéré pour obtenir tous les liens vers d'autres pages Wikipédia.
4. Présentation à l'utilisateur des liens de la page et demander lequel utiliser.
5. Boucler à partir de l'étape 3, jusqu'à trouver la page d'arrivée.

2.6 TP Noté : Réalisation d'un programme

Vous avez 3h pour développer le programme correspondant à la modélisation UML donnée plus bas, et ajouter les fonctionnalités supplémentaires indiquées dans les questions suivantes. L'utilisation des outils testés en TP n'est pas obligatoire mais rapporte des points. Voici la liste :

1. Un système de versionnement (Git, SVN, etc) ;
2. Maven ;
3. GitHub/Gitlab, Travis CI, Code Climate ;

Pour rendre votre travail :

— Vous devrez envoyer votre code source via UMTICE à l'adresse suivante (clé PAAP4) : `http://umtice.univ-lemans.fr/course/view.php?id=4869#section-5`
— Faites un "mvn clean", puis faites un zip entier du dossier contenant les sources.
— Si votre projet est de plus hébergé avec GitHub/Gitlab, n'oubliez pas d'indiquer l'URL vers le projet.

En cas de problème technique avec les technologies choisies (Maven, etc), n'hésitez pas à demander de l'aide. Utilisation de internet recommandée.

Lisez tout le sujet.

Bon courage !

Étape 1

Si vous utilisez un ordinateur de l'université, configurez Eclipse avec le proxy ainsi que Maven. Référez-vous aux parties "Astuces" ci-dessous si besoin.

Étape 2

Créez un nouveau projet Java avec Maven, sans archetype.

Étape 3

Ajoutez la dépendance de JUnit.

Étape 4

Réalisez le diagramme UML ci-dessous en Java.

Toutes les méthodes présentes dans le diagramme doivent être codées et doivent fonctionner.

Étape 5

Voici le sujet de l'application :

(a) Nous souhaitons développer un programme Java reflétant les autoroutes de France (via une modélisation basique).

(b) Développez des classes pour chacun de ces aspects liés aux autoroutes :
- Compagnie
- Route
- Porte
- Porte d'entrée
- Porte de sortie
- ~~Porte technique~~
- Tarifs
- Un voyage

(c) Une compagnie d'autoroute possède des routes, qui possèdent différent types de portes.

(d) ~~Les techniciens de l'autoroute peuvent patrouiller, et emprunter les portes techniques.~~

(e) Une compagnie possède aussi une liste de tarifs en fonction de chaque paire de portes. Chaque tarif et valable pour une période précise. Exemple : "Entree 1 avec Sortie 2 coûte 3.40 €".

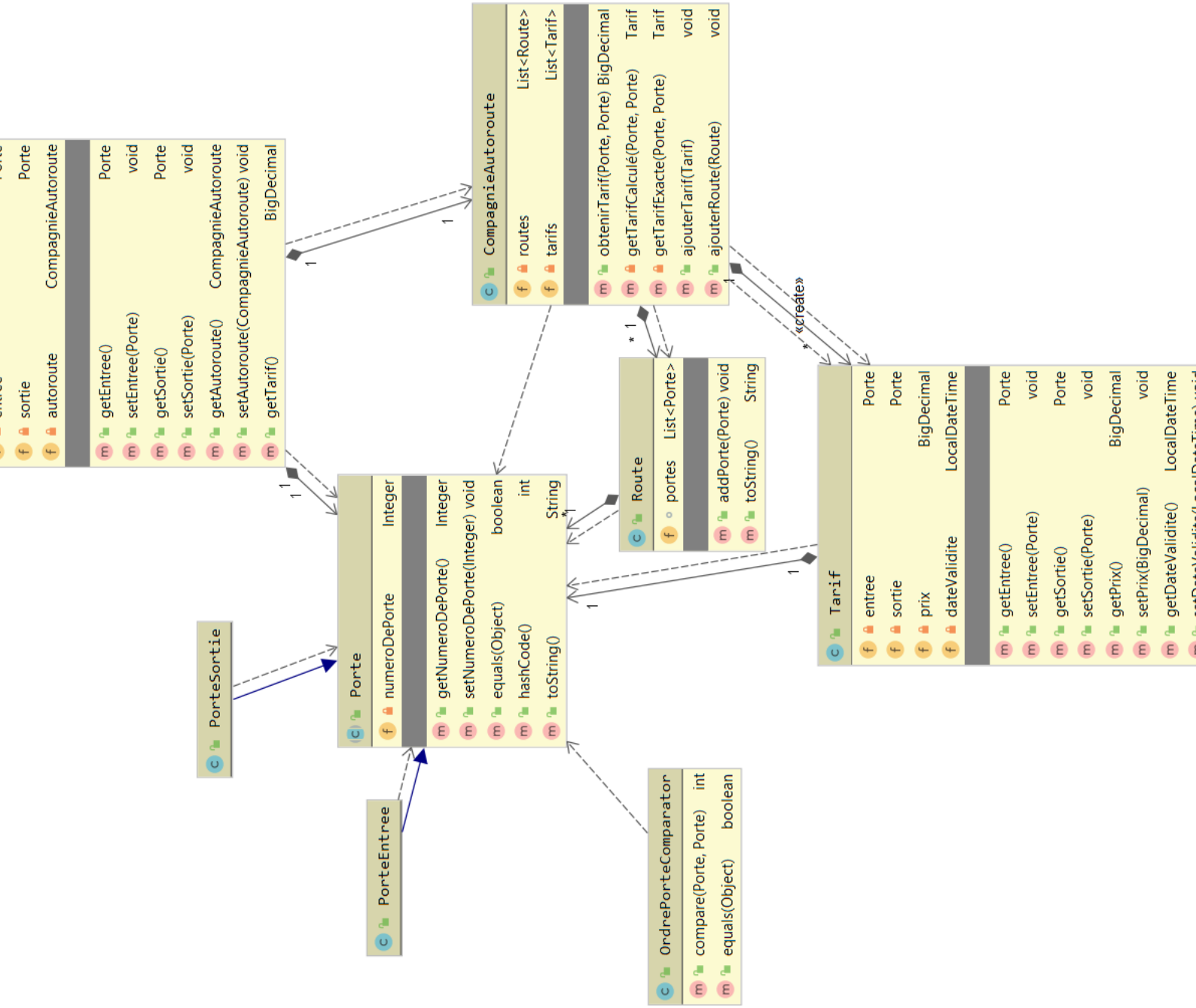
(f) Un Voyage contient une porte d'entrée, une porte de sortie, et un tarif

(g) On souhaite pouvoir ajouter des autoroutes au système, avec les portes, et utiliser un *Comparator* (classe *OrdrePorteComparator*) pour trier correctement la liste des portes de l'autoroute.

(h) On souhaite pouvoir planifier un voyage, indiquer les portes, et récupérer le prix de l'autoroute.

(i) Un tarif se calcule de la manière suivante (méthode public *CompagnieAutoroute.getTarif*) :

1. Si on a un tarif existant déjà configuré pour le couple entrée/sortie, on prend ce tarif (dans la modélisation, la méthode privée *CompagnieAutoroute.getTarifExacte*) ;
2. Si nous n'avons pas de tarif existant pour le couple Porte d'entrée/sortie, on utilise les tarifs de chaque porte pour calculer le prix global (dans la modélisation, la méthode privée *CompagnieAutoroute.getTarifCalculé*).
   Algorithme : Si l'objet Tarif analysé est entre les portes d'entrée et de sortie du voyage, on l'additionne au prix trouvé.


CompagnieAutoroute
routes List<Route>
tarifs List<Tarif>
obtenirTarif(Porte, Porte) BigDecimal
getTarifCalculé(Porte, Porte) Tarif
getTarifExacte(Porte, Porte) Tarif
ajouterTarif(Tarif) void
ajouterRoute(Route) void
Route
portes List<Porte>
addPorte(Porte) void
toString() String
Tarif
entree Porte
sortie Porte
prix BigDecimal
dateValidite LocalDateTime
getEntree() Porte
setEntree(Porte) void
getSortie() Porte
setSortie(Porte) void
getPrix() BigDecimal
setPrix(BigDecimal) void
getDateValidite() LocalDateTime
Porte
numeroDePorte Integer
getNumeroDePorte() Integer
setNumeroDePorte(Integer) void
equals(Object) boolean
hashCode() int
toString() String
PorteSortie
PorteEntree
OrdrePorteComparator
compare(Porte, Porte) int
equals(Object) boolean
Porte
sortie Porte
autoroute CompagnieAutoroute
getEntree() Porte
setEntree(Porte) void
getSortie() Porte
setSortie(Porte) void
getAutoroute() CompagnieAutoroute
setAutoroute(CompagnieAutoroute) void
getTarif() BigDecimal
«create»
1
*
*1

Étape 6

Une classe de tests unitaires pour *OrdrePorteComparator.java* est disponible à l'adresse suivante : `https://e-gitlab.univ-lemans.fr/snippets/12`. Incorporez-la à votre application, lancez les tests et vérifiez le passage de tous les tests avec Eclipse et Maven (pour voir que la dépendance de JUnit est utilisée).
Info : n'oubliez pas la dépendance Maven pour JUnit 4. Testez avec "mvn test".

Étape 7

Une classe de tests unitaires pour *CompagnieAutorouteTest.java* est disponible à l'adresse suivante : `https://e-gitlab.univ-lemans.fr/snippets/13`. Incorporez-la à votre application, lancez les tests et vérifiez le passage de tous les tests avec Eclipse et Maven (pour voir que la dépendance de JUnit est utilisée).

Étape 8

Créez un projet git **public** sur GitHub.
Attention : Si vous préférez Gitlab, utilisez le Gitlab officiel `https://gitlab.com` pour avoir accès à des *Runners* gratuit qui feront la compilation de vos tests (Gitlab CI) et l'analyse du code source (CodeClimate), après avoir configuré le fichier *.gitlab-ci.yml*

Étape 9

Configurez votre projet sur GitHub.

Étape 10

Configurez votre projet avec Code Climate.

Étape 11

Faites un commit dédié à la correction de quelques *code smells* détecté par Code Climate.

Étape 12

Ajoutez une impression écran de Code Climate dans le README.md.

Étape 13

Nous allons maintenant générer un site auto-construit de votre application.

(a) Essayez de lancer la commande suivante :

```
mvn site:site
```

(b) Faites en sorte que la commande se termine en "BUILD SUCCESS".

(c) Vous pouvez utilisez la commande suivante pour le voir en live :

```
mvn site:run
```

(d) (Si vous devez changer le port par défaut, suivez la documentation ici : `https://maven.apache.org/plugins/maven-site-plugin/examples/siterun.html`)

(e) Ajoutez la Javadoc de votre application à votre site (ajoutez des commentaires dans votre application avant).

(f) Regardez comment marche la fonctionnalité "GitHub Pages" de GitHub.

(g) Faites fonctionner *GitHub Pages* avec le site que génère

```
mvn site:site
```

Étape 14

(Bonus) Générez un fichier .jar avec la commande suivante, le .jar doit fonctionner en utilisant la commande "java -jar fichier.jar".

```
mvn clean package
```

— Indice : Le paramétrage du build du jar passe par la configuration du pom.xml.
— Info : si la commande “java ..” ne fonctionne pas car Java est introuvable, configurez la variable d’environnement “Path” pour qu’elle contienne le chemin vers le fichier “java.exe”, puis lancez un nouveau terminal.

3 Astuces

3.1 Problèmes avec les dépendances Maven dans Eclipse

Proxy + Maven + Eclipse = problèmes !
Voici quelques astuces pour vous débloquer (toutes indépendantes) :

1. Vérifier les options d'Eclipse pour s'assurer que le proxy est configuré et fonctionne (en essayant de voir la liste des plugins par exemple).
2. Maven dans Eclipse possède un fichier de configuration, visible ici : "Window >Preferences >Maven >User settings" (le chemin est pré-rempli s'il n'est pas modifié). Ajouter une partie pour configurer le proxy.
3. Configurez Maven en ligne de commande en téléchargeant la dernière version de Maven, ajoutez-le dans votre PATH (pour pouvoir l'utiliser en ligne de commande), et lancer la commande suivante dans le dossier de votre programme :

   ```
   mvn clean compile
   ```

   Le message d'erreur sera plus clair et pourra être plus facilement cherché sur internet.
4. Dans "Eclipse : clique droit sur le projet >Maven >Update project" pour prendre en compte les modifications du POM, et vérifier les dépendances.

3.2 Configuration du proxy dans les applications, Maven, Eclipse, Spring

3.2.1 Maven

Dans votre fichier de configuration Maven, utiliser ce fichier de configuration Maven pour le proxy à l'emplacement suivant "/home/*user*/.m2/settings.xml", ou "D :/Profiles/*user*/.m2/settings.xml" (votre dossier personnel), ou "D :/Users/*login ENSIM*/.m2/settings.xml".

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
    <proxies>
        <proxy>
            <active/>
            <protocol>http</protocol>
            <port>3128</port>
            <host>proxy.univ-lemans.fr</host>
            <id/>
        </proxy>
    </proxies>
</settings>
```

Disponible au lien suivant : `https://e-gitlab.univ-lemans.fr/snippets/1`

3.2.2 Eclipse

"Windows >Preferences >General >Network Connections" avec les options suivantes pour le schéma HTTP, HTTPS et SOCKS (Et avec **Active Provider** à *Manual*)

— Host : proxy.univ-lemans.fr
— Proxy : 3128

3.2.3 Programmation Java

```
java.net.Proxy proxyTest = new
   java.net.Proxy(java.net.Proxy.Type.HTTP,new
   InetSocketAddress("proxy.univ-lemans.fr", 3128));
OkHttpClient.Builder builder = new
   OkHttpClient.Builder().proxy(proxyTest);
Starter.client = builder.build();
```

3.2.4 Spring

```
SimpleClientHttpRequestFactory clientHttpReq = new
   SimpleClientHttpRequestFactory();
        Proxy proxy = new Proxy(Proxy.Type.HTTP, new
           InetSocketAddress("proxy.univ-lemans.fr", 3128));
        clientHttpReq.setProxy(proxy);

        RestTemplate restTemplate = new
           RestTemplate(clientHttpReq);
```

3.3 Problème de JDK/JRE avec Maven

Si Maven en ligne de commande ne fonctionne pas à cause d'un problème de JRE/JDK, allez vérifier la configuration JDK/JRE dans Eclipse dans les préférences : Windows, Preferences, Java >Installed JREs : **il faut un JDK de configuré**.

3.4 Problème de compilation non compatible JDK 5

La configuration par défaut de la version de Maven utilisé à l'ENSIM demande une rétrocompatibilité avec Java 5 sur tous les nouveaux projets via le plugin *maven-compiler-plugin*.
Pour modifier cette option dans vos projets Maven, vous pouvez ajouter ces paramètres dans le *pom.xml* qui va indiquer à la compilation d'être compatible avec le JDK 8 :

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Disponible au lien suivant : `https://e-gitlab.univ-lemans.fr/snippets/11`