

1. Présentation générale

Chaque semaine vous allez devoir rendre une partie de votre travail sur Moodle. Ça sera généralement la partie centrale d'un projet. Ces documents seront évalués dans le cadre de la note de TP. Ainsi, il n'y aura pas un TP noté final, mais de multiples travaux à faire dans le cadre d'un contrôle continu. Les TP ne doivent pas être fait à la maison, seulement pendant les séances, en présence d'un enseignant.

Ce système de notation repose sur l'honnêteté de tout le monde. Si des tricheries sont constatées, on sera obligé de revenir à des devoirs sur table et des contrôles de TP sur table également. À vous de voir ce que vous préférez.

Ce premier TP présente la création de projets Java gérés par Maven, ainsi que les tests unitaires JUnit5, avec Eclipse. Vous pourrez vous servir de Visual Studio Code ou IntelliJ uniquement si vous savez déjà vous en servir. Vous n'aurez aucune assistance sur ces outils.

Comme il y a beaucoup de problèmes de mise en route, de configuration des machines, ce TP1 comptera comme un bonus dans la note finale. Ce que vous pourrez faire sera compté dans votre note si ça permet de la monter, mais ça sera ignoré si vous n'avez pas pu avancer dans le TP.

2. Création d'un projet Maven sur Eclipse

On commence par préparer un espace de travail (*workspace*) pour tous les TP :

1. Lancer Eclipse,
2. Choix du *workspace* : en créer un nouveau, réservé au cours *R4.02 Qualité logicielle*.

2.1. Configuration de Eclipse pour travailler avec Maven

Il y a une configuration à faire sur les postes fixes de l'IUT avant de continuer. Ne pas la faire sur les portables prêtés par l'IUT et sur vos ordinateurs, seulement sur les postes fixes des salles de TP.

1. Menu **Window**, item **Preferences**,
2. Cherchez et déployez la catégorie **Maven**, puis **User Settings**, voir la figure 1,
3. Dans la case **Global Settings**, tapez `/etc/maven/settings.xml`. Vous pouvez cliquer sur le lien **open file** pour vérifier que le fichier est présent. Ce fichier contient l'adresse IP et le port du proxy, `129.20.239.11`, port `3128`. Le proxy est le mécanisme qui bloque les accès interdits à l'extérieur de l'IUT et qui met les fichiers en cache pour un accès plus rapide.
4. Fermez le dialogue avec **Apply and Close**.

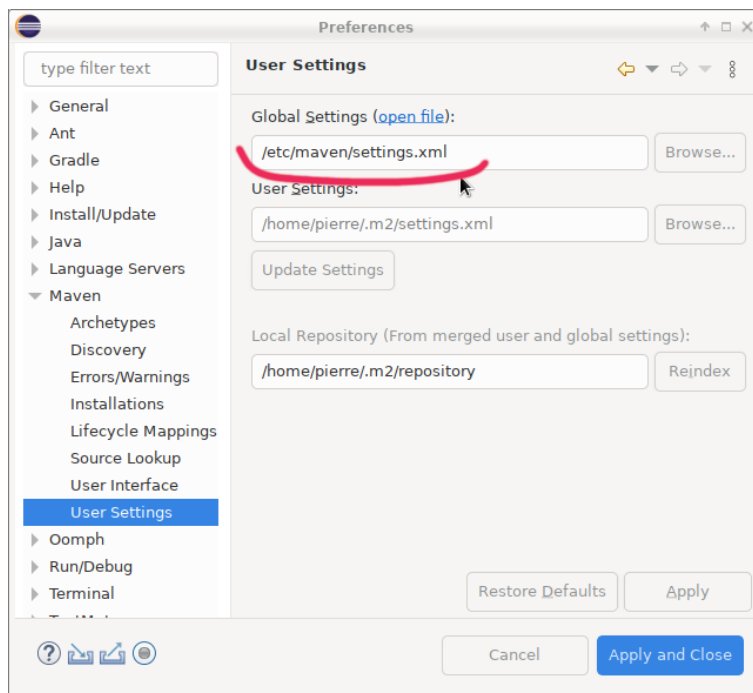


Figure 1: User Settings dans les préférences

Normalement, cette configuration est mémorisée dans le *workspace*, donc vous n'aurez pas à la refaire dans les prochains TPs.

2.2. En cas de `/etc/maven/settings.xml` manquant (pour info)

À ne faire que sur les PC fixes des salles de TP.

La configuration de Eclipse dépend du fichier `/etc/maven/settings.xml`. Parfois il n'est pas accessible, à cause d'une panne temporaire de Puppet, un service de mise à jour de tous les PC de l'IUT. Si ça arrive, Eclipse se retrouve totalement bloqué, parce qu'il se sait plus qu'il faut passer par le proxy pour ses téléchargements. Il faut alors créer un fichier de configuration dans votre compte.

1. Ouvrez un terminal shell,
2. CTRL-cliquez sur le bouton jaune de droite pour ouvrir le source ci-dessous dans un nouvel onglet, copiez toutes ces lignes et collez-les dans le terminal par un SHIFT-CTRL-V : 📄

```
mkdir -p ~/.m2
cat <<data > ~/.m2/settings.xml
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 https://maven.apache.org/x
<proxies>
  <proxy>
    <id>proxy IUT</id>
    <active>true</active>
    <protocol>https</protocol>
```

```
<host>129.20.239.11</host>
<port>3128</port>
</proxy>
</proxies>
</settings>
data
```

NE PAS COPIER CE SOURCE DU PDF, MAIS DE [CE LIEN](#)

Ces commandes créent le dossier `.m2` dans votre compte et le fichier `settings.xml` qui permet de se connecter aux dépôts Maven.

Le dossier `.m2` de votre compte contient les fichiers `jar` qui complètent le SDK Java quand il y a des dépendances dans un projet, des `import` pas installés en standard.

2.3. Création du projet Maven

Apache Maven est un environnement pour construire, tester et distribuer des logiciels Java. Il automatise toutes les phases de compilation, tests, mise en jar et téléchargement sur un dépôt. Maven se sert d'un fichier appelé `pom.xml` à la racine d'un projet et qui décrit ce qu'il y a dans le projet : ses sources et ses dépendances, c'est à dire les bibliothèques utilisées, ainsi que les particularités de construction.

D'autres outils existent, comme `make` (inadapté à Java), `Ant`, `Ivy`, et `Gradle`.

Maven est intégré dans Eclipse, et peut également être lancé en ligne de commande dans le dossier du projet. Il faut dans tous les cas qu'il y ait un fichier appelé `pom.xml` décrivant le projet et ses dépendances (imports et outils). Heureusement, il y a un assistant dans Eclipse qui permet de les écrire sans efforts.

1. Créer un nouveau projet :
 - a. Menu **Fichier/Nouveau...**, choisir l'item **Maven Project**
 - b. Ça peut aussi se trouver dans une boîte de dialogue **Autres...**

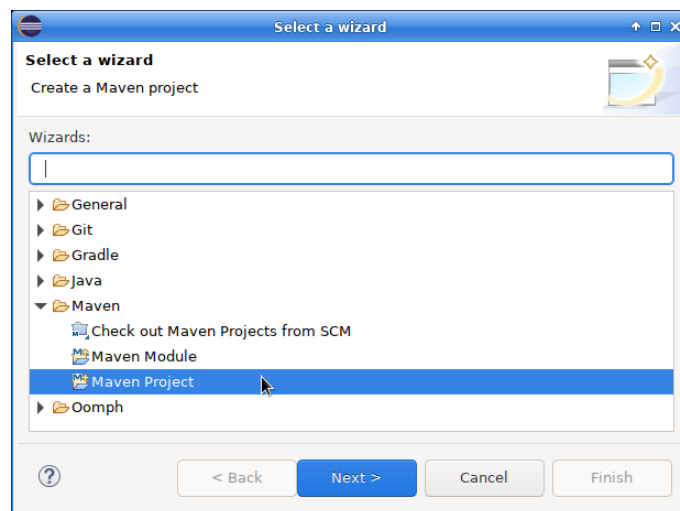


Figure 2: Choix du type de projet : *Maven Project*

2. Dans le dialogue qui apparaît, cocher ☐ Créer un projet simple (pas d'archétype), puis Suivant

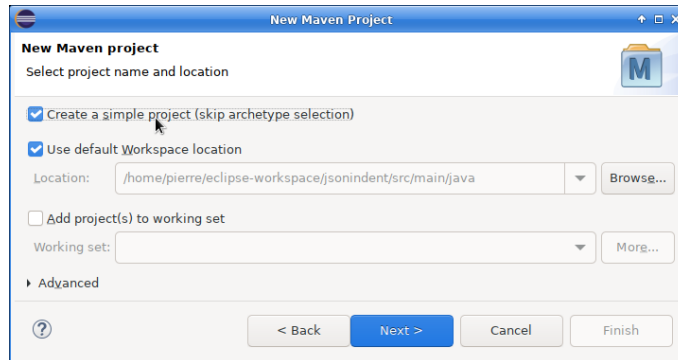


Figure 3: Projet Maven de type simple

3. Ensuite, entrez les informations du projet :
- GroupID : fr.iutlan.q1
 - ArtifactID : tp1
 - Version : 0.0.1 (enlevez snapshot)
 - Name: laissez vide
 - Description : mettez un commentaire comme « projet du TP1 »

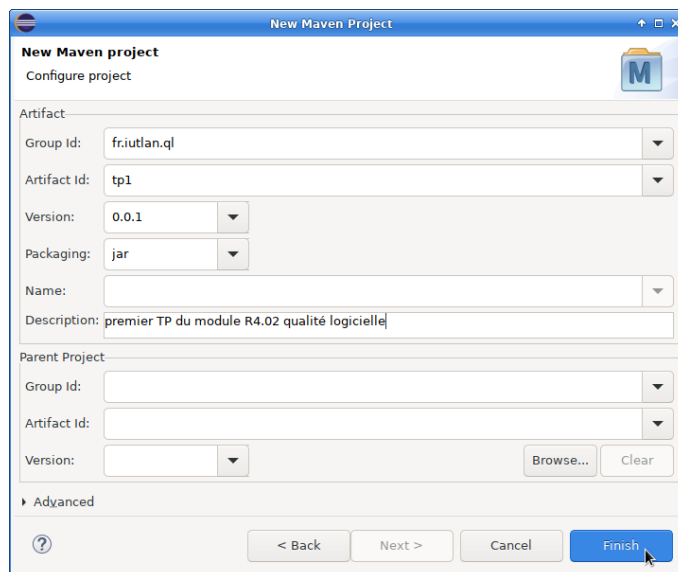


Figure 4: Informations du projet

4. Validez. Eclipse génère un fichier appelé pom.xml ainsi que tous les dossiers nécessaires. Le fichier pom.xml décrit le projet, son nom, package ainsi que ses dépendances. Voici la structure :


```
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>PACKAGE RACINE DU PROJET</groupId>
  <artifactId>NOM DU PROJET</artifactId>
```

```
<version>VERSION DU PROJET</version>  
<description>DESCRIPTION DU PROJET</description>  
</project>
```

C'est un fichier XML. Ce format de données sera étudié en détails dans le TP6. Pour l'instant, considérez que c'est comme un HTML, mais avec des balises différentes. Vous retrouvez les informations que vous avez saisies dans la boîte de dialogue.

Ne confondez pas `<modelVersion>` avec `<version>`. La première, valant toujours 4.0.0, est liée à la version de Maven, tandis que `<version>` donne la version de votre logiciel. Il faudra la faire évoluer à chaque fois que vous publiez une nouvelle version.

NB: Eclipse affiche toujours une erreur sur la première ligne de ce fichier, sur un attribut `xsi:schemaLocation`. N'en tenez pas compte.

5. Un projet Maven créé par l'assistant d'Eclipse n'est pas viable en dehors d'Eclipse. Il faut indiquer la version de Java à employer, car, par défaut, Maven utilise Java 1.5 datant de 2004. Donc il faut ajouter ces lignes au fichier `pom.xml` juste avant la balise `</project>` finale, pour avoir du Java 8 : 

```
<project xmlns="http://maven.apache.org/POM/4.0.0">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>fr.iutlan.ql</groupId>  
  <artifactId>tp1</artifactId>  
  <version>0.0.1</version>  
  <description>premier TP du module R4.02 qualité logicielle</description>  
  
  <properties>  
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
    <maven.compiler.source>8</maven.compiler.source>  
    <maven.compiler.target>${maven.compiler.source}</maven.compiler.target>  
  </properties>  
</project>
```

Vous pouvez mettre 11 à la place de 8 si vous avez Java 11 sur votre machine.

6. Il faut ensuite faire une étape importante : mettre à jour ce projet pour Eclipse. Utilisez le menu contextuel du projet : clic droit sur le nom du projet dans le *package explorer*, puis Maven, puis Update Projet...

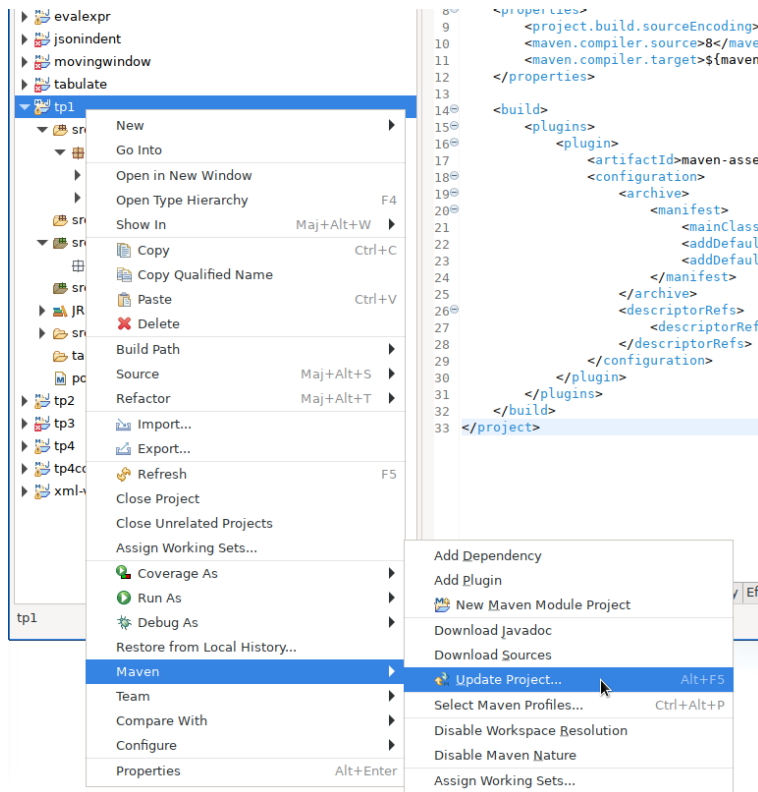


Figure 5: Menu Maven/Update

7. Dans ce dialogue, le projet concerné doit être coché. Valider.

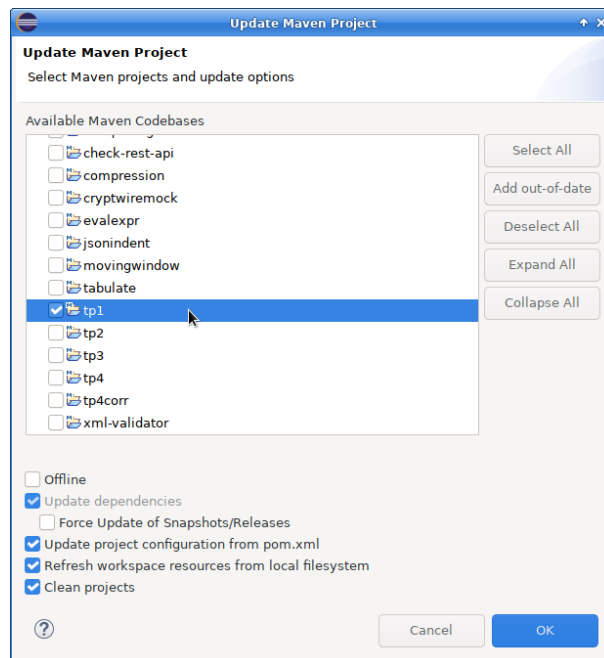


Figure 6: Maven Update, choix des projets

IMPORTANT si la mise à jour reste bloquée, ou affiche un message d'erreur, c'est qu'il y a un gros problème réseau. Voir le § dépannage suivant.

2.4. Dépannage

Il faut savoir que Maven, qu'il soit lancé en ligne de commande `mvn` ou par Eclipse utilise une sorte de cache pour les fichiers, situé dans `~/.m2/repository`. Il est extrêmement fréquent que ce cache soit corrompu :

- erreur réseau lors du téléchargement initial d'un fichier,
- erreur réseau avec le serveur NFS,
- quota disques épuisés.

Dans tous les cas, les fichiers sont créés mais ils sont vides.

Le malheur, c'est que Maven ne fait aucun test d'intégrité (et pourtant il pourrait), et ne relance pas le téléchargement.

La seule solution qui reste, c'est de supprimer le cache. Voici les commandes :

```
cd ~/.m2
rm -fr repository
```

Ensuite, recommencez **Update Maven Project** dans le menu **Project**. Ça doit réussir. Si ça échoue encore, vérifiez :

- vos quotas disque : pensez à nettoyer le cache de Chrome.
- le contenu du fichier `~/.m2/settings.xml` et celui de `/etc/settings/maven.xml`. Le premier est prioritaire sur le second. Sont-ils correctement configurés dans le menu **Window, Preferences, Maven, User Settings** ?

Il faut faire cela au moindre doute, au moindre blocage de Maven ou d'Eclipse – quand ce n'est pas lié à votre programmation.

2.5. Structure des fichiers du projet

Le projet est maintenant opérationnel. Avant de passer au développement, on propose de vérifier la structure des fichiers dans le *workspace*.

8. Ouvrez le dossier du *workspace* en cliquant droit sur le projet, puis menu **Show In puis System explorer**, ou ouvrez directement l'explorateur de fichiers du système.

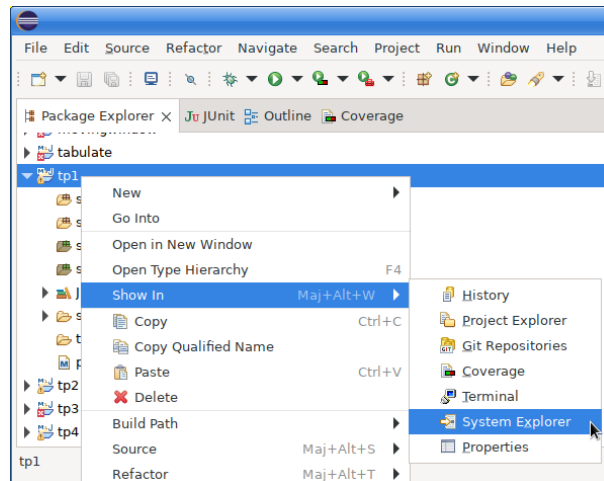


Figure 7: Afficher dans l'explorateur de fichiers

Vous allez voir :

- deux dossiers, **src** et **target** : le premier est pour les sources, le second pour tout ce qui est construit par Maven, **.class** et **.jar**.
- dans **src**, vous verrez **main** et **test**. Le premier est pour les fichiers du projet, le second pour les tests. Ils sont vides pour l'instant.
- dans chacun, vous verrez **java** et **resources**. Le premier est pour les sources Java, le second pour des fichiers de données accompagnant les sources, par exemple les fichiers de traduction, des icônes, etc.

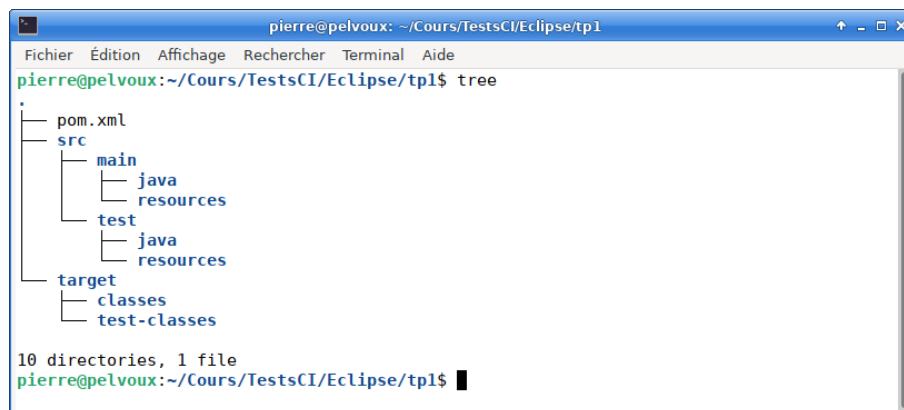


Figure 8: Dossiers du projet

3. Contenu du premier projet

On va programmer une classe et quelques méthodes pour prendre en main tous ces outils. Le but est de construire un fichier *jar* pouvant être lancé comme ceci :

```
java -jar tp1.jar
```

Comment fait-on pour construire un tel fichier *jar* ? Il faut au moins une classe contenant une méthode **main** et ensuite compiler.

3.1. Ajout d'une classe Main

1. Dans `src/main/java` (fenêtre *Package Explorer* de Eclipse), créez un *package* nommé `fr.iutlan.ql.tp1` (interdiction de changer le nom)

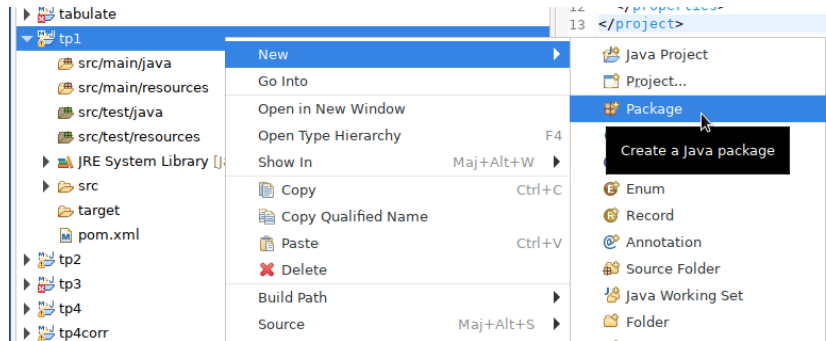


Figure 9: Menu pour créer un *package*

Surtout, ne cochez pas *create package-info.java* ou débarrassez-vous du fichier ensuite.

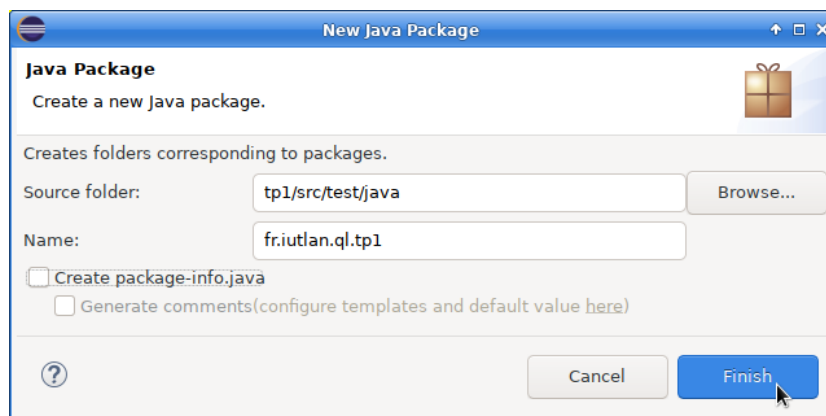


Figure 10: Package du TP1

NB : La copie écran est dans le dossier `test`, mais créez vraiment dans le dossier `main`.

NB : Vous avez l'interdiction absolue de changer le nom du package et de la classe, ainsi qu'en général toutes les méthodes qu'il faudra créer. Cela vous coûtera des points pour toute perte de temps entraînée par vos changements. La raison, c'est que vos sources sont placés dans un *bac à sable* pour y être testés de manière automatique. Si vous changez des noms de classe ou package, vous obligez à les remettre pour pouvoir être testés.

2. Créez une classe nommée `Main` dans ce package. Cochez bien l'option `public static void main()`.

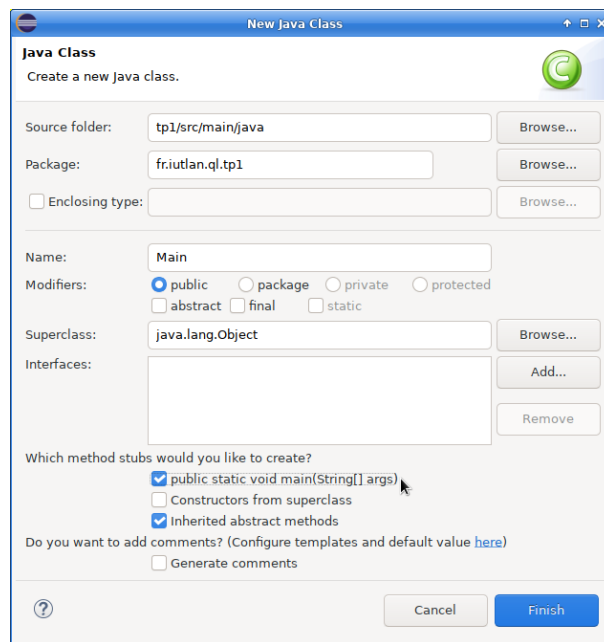


Figure 11: Classe Main

3.2. Construction d'un *jar*

L'extension *.jar* signifie *Java archive*. En fait, c'est un fichier compressé zip contenant les fichiers compilés, les *.class* correspondant aux *.java* et des fichiers complémentaires.

Le problème ici, est de construire un *jar* qui soit exécutable. Il faut qu'il contienne un *manifeste*, c'est à dire un bordereau de livraison, une liste du contenu de l'archive, ainsi que toutes les classes compilées nécessaires. Ce manifeste est créé automatiquement par maven à partir d'un ajout au *pom.xml*.

1. Complétez le fichier *pom.xml* (attention à garder l'existant) :



```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.iutlan.ql</groupId>
  <artifactId>tp1</artifactId>
  <version>0.0.1</version>
  <description>premier TP du module R4.02 qualité logicielle</description>

  <properties>
    ...
  </properties>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <archive>
```

```
<manifest>
  <mainClass>fr.iutlan.ql.tp1.Main</mainClass>
  <addDefaultImplementationEntries>true</addDefaultImplementationEntries>
  <addDefaultSpecificationEntries>true</addDefaultSpecificationEntries>
</manifest>
</archive>
<descriptorRefs>
  <descriptorRef>jar-with-dependencies</descriptorRef>
</descriptorRefs>
</configuration>
</plugin>
</plugins>
</build>
</project>
```

La balise `<mainClass>` donne le nom complet Java de la classe qui contient la méthode `public static void main()`, le point d'entrée du logiciel.

2. Si nécessaire, réindentez correctement le fichier `pom.xml` avec le menu contextuel **Source**, **Format**.

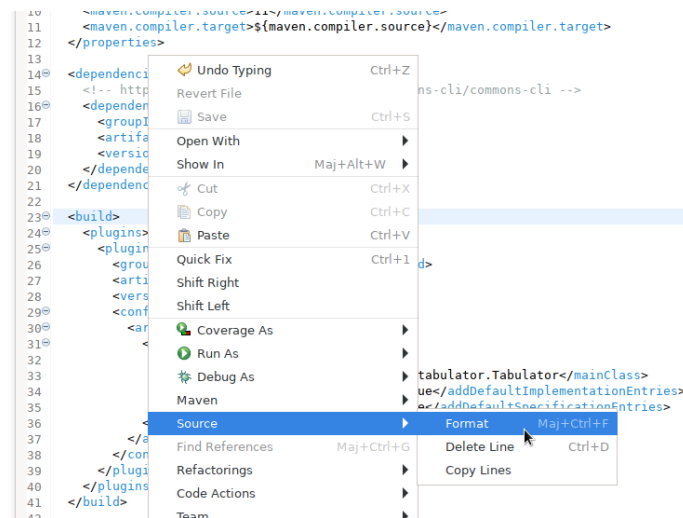


Figure 12: Corriger l'indentation du `pom.xml`

3. Mettez à jour Maven dans Eclipse, voir l'étape 6, page 5.

Maintenant, on peut compiler le logiciel.

1. Ouvrez un shell dans le dossier du projet, dans `tp1`, là où se trouve `pom.xml`.
2. Tapez `mvn package` : compilation des sources
3. Tapez `mvn assembly:single` : regroupement dans un jar exécutable
4. Tapez `cp target/tp1-*-jar-with-dependencies.jar tp1.jar` : copie dans le dossier du TP
5. Tapez `java -jar tp1.jar` : exécution, mais ça n'affiche rien encore.

La commande `mvn` est très générale et a besoin d'au moins un paramètre appelé *cible* (*goal*). Ici, c'est `package` pour compiler les sources, puis `assembly:single` pour créer le *jar* exécutable. Il y

a d'autres cibles, par exemple, `clean` pour supprimer le dossier `target`, c'est à dire tout ce qui est compilé :

```
mvn clean
mvn package
mvn assembly:single
```

On peut mettre plusieurs cibles qui seront exécutées dans l'ordre.

```
mvn clean package assembly:single
```

3.3. Ajout d'une méthode à la classe Main

1. Éditez `Main.java` ainsi :



```
package fr.iutlan.ql.tp1;

public class Main {

    /* retourne la version inscrite dans le fichier pom.xml */
    static String getVersionString() {
        return Main.class.getPackage().getImplementationVersion();
    }

    public static void main(String[] args) {
        System.out.println("Version "+getVersionString());
    }
}
```

2. Reconstituez le fichier `tp1.jar` et lancez-le :



```
mvn clean package assembly:single
cp target/tp1-*-jar-with-dependencies.jar tp1.jar
java -jar tp1.jar
```

3. Incrémentez le numéro de version mineur, `<version>0.0.2</version>` dans le fichier `pom.xml`.
4. Reconstituez le fichier `tp1.jar` et lancez-le (dans le terminal). La version affichée doit avoir changé.

4. Ajout d'une classe

Le but du TP est d'écrire quelques tests unitaires sur une classe. Donc il va y avoir deux classes, la classe à tester et celle qui fait les tests. On commence par la première.

1. Utilisez les menus pour créer une classe appelée `StringUtilsils` dans le *package* `fr.iutlan.ql.tp1`. (ne pas changer les noms)
2. Éditez `StringUtilsils.java` ainsi :



```
package fr.iutlan.ql.tp1;

public class StringUtils
{
    /**
     * calcule la longueur d'une sous-chaîne de str comprise entre beg et end
     * c'est à dire qu'avec str = "XXXXXbegYYYYYYendZZZZZ", il faut retourner
     * la longueur de YYYYYY
     * Si beg ou end sont absents de str, alors lengthBetween retourne -1
     * @param str chaîne dans laquelle on cherche beg puis end
     * @param beg sous-chaîne définissant le début (première occurrence)
     * @param end sous-chaîne définissant la fin (première occurrence après beg)
     * @return nombre de caractères entre beg et end (exclus) dans str
     */
    static int lengthBetween(String str, String beg, String end) {
        // chercher beg dans str
        int begIndex = str.indexOf(beg);
        if (begIndex < 0) return -1;
        // on part de la fin de begIndex
        begIndex += beg.length();
        // chercher end dans str à partir de beg
        int endIndex = str.lastIndexOf(end);
        if (endIndex < 0) return -1;
        // écart entre endIndex et begIndex
        return endIndex - begIndex;
    }
}
```

Si vous remarquez quelque chose, c'est bien mais laissez comme ça pour l'instant. Votre but est de tester cette classe et méthode, donc de faire apparaître les problèmes.

3. Éditez Main.java ainsi :



```
package fr.iutlan.ql.tp1;

public class Main {

    /* retourne la version inscrite dans le fichier pom.xml */
    static String getVersionString() {
        return Main.class.getPackage().getImplementationVersion();
    }

    public static void main(String[] args) {
        System.out.println("Version "+getVersionString());

        int lng = StringUtils.lengthBetween("bonjour tout le monde", "jour", "mon");
        System.out.println("lng = " + lng);
    }
}
```

```
}
```

4. Construisez le jar et lancez-le. Ça devrait afficher `lng = 9`, parce qu'il y a 9 caractères entre la fin de "jour" et le début de "mon". Donc le programme marche. Voilà.

En fait, il y a plusieurs problèmes et le but est de les mettre en évidence par des tests automatisés, qu'on appelle *tests unitaires*.

5. Tests unitaires

Le concept de tests unitaires est simple :

- On dispose d'une classe à tester `Classe1` ayant différentes méthodes.
- On écrit une autre classe `TestsClasse1` dont les méthodes vont essayer celles de `Classe1`, et vérifier qu'elles retournent les bonnes valeurs en fonction des paramètres fournis et qu'elles émettent les bonnes exceptions quand il le faut.
- Si les résultats sont ceux attendus, les tests réussissent. Sinon, `Classe1` est mauvaise.

Le but étant d'éviter de fournir des logiciels incorrects au client. Il est préférable d'éliminer les bugs le plus tôt possible. Donc, il faut prévoir les tests de qualité dès le début de la conception du logiciel.

Ajouter des tests à un projet Maven est très facile. Il suffit de créer une classe dans le dossier `src/test/java`, au lieu de `src/main/java`, mais avec le même *package* que celle qu'on veut tester, de manière à accéder aux méthodes de visibilité réduite au *package* (ni `private`, ni `protected`).

Et dans cette classe de tests, il suffit de programmer des méthodes annotées par `@Test`, et ces méthodes doivent faire différents essais sur la classe à tester.

5.1. Recommandations sur les classes et méthodes à tester

Évidemment, pour bien tester, il faut bien organiser les essais. Normalement, les essais doivent être les plus élémentaires possibles. C'est à dire qu'ils doivent ne vérifier qu'une seule chose à la fois, afin de bien caractériser les défauts s'il y en a. Si une méthode de test vérifie 36 choses en même temps, on ne saura pas exactement pourquoi ça a planté et quoi corriger.

On doit décomposer le plus possible toutes les vérifications et il faut que la classe testée le permette. Si c'est une boîte noire, avec uniquement des méthodes privées, alors il n'y a pas grand chose à faire.

Ces tests élémentaires sont appelés **tests unitaires**. Dans un prochain TP, nous verrons des tests d'intégration (collaboration de plusieurs classes), puis des tests fonctionnels (ex: tests des interfaces utilisateur) qui ne sont essayés que lorsque tous les tests unitaires ont réussi.

Chaque test unitaire est lui-même organisé en trois étapes :

1. *Arrange* ou *Given* : étant donné ceci...
2. *Act* ou *When* : quand je fais cela...
3. *Assert* ou *Then* : je dois obtenir ce résultat.

La première phase, optionnelle, prépare des données. La deuxième phase effectue l'action qui est

surveillée. La troisième phase vérifie que l'action a un résultat conforme. Dans certains cas, le résultat conforme est une exception, c'est à dire qu'on s'attend à ce que l'action plante.

Ces trois étapes sont un **patron de conception des tests** appelé *AAA* ou *Given-When-Then* selon les auteurs.

La complexité des tests réside dans la manière de vérifier les résultats. Ça se fait avec des fonctions de comparaison un peu plus élaborées que `==`, parce que le risque est assez important de rater la comparaison entre ce qui est obtenu et ce qui est attendu. On emploie des méthodes de la librairie *JUnit5* appelées **assertions** (affirmations qui doivent être vraies).

5.2. Présentation rapide de JUnit5

JUnit5 est un ensemble permettant de tester des classes et méthodes Java. Il est composé de trois éléments :

- *JUnit Platform* : outil de lancement des tests, il cherche les classes de test et appelle leurs méthodes. C'est invisible à notre niveau.
- *JUnit Jupiter* : ce sont les méthodes et annotations pour programmer des tests.
- *JUnit Vintage* : c'est un outil pour faire tourner des tests écrits dans les versions précédentes JUnit4 et JUnit3.

Une classe de test ressemble à ceci :

```
package PACKAGE DE LA CLASSE A TESTER;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;

public class CLASSETests { // ou TestsCLASSE

    @Test
    void methodOnNullMustReturnNull() {
        // ARRANGE
        CLASSE obj = new CLASSE();

        // ACT
        String result = obj.method(null);

        // ASSERT
        assertNull(result);
    }
}
```

On peut remarquer tout cela :

- Le *package* est celui de la classe à tester, mais cette classe de test est dans le dossier `src/test/java`, donc elle ne sera pas incluse dans le *jar*.
- `import static` pour pouvoir utiliser les méthodes statiques directement.
- Les importations pourraient être limitées aux seules choses utilisées, au lieu de l'étoile `*`.

- Le nom de la classe de test fait référence à la classe testée. C'est pour séparer des classes de tests dans le même *package*, chacune sa classe testée.
- Le nom de la méthode de test doit être le plus parlant possible, quitte à être très long. En le lisant, on doit voir ce qui est testé.
- On **doit** écrire les trois commentaires // ARRANGE, etc. et laisser une ligne vide entre les trois étapes. C'est une mise en page qui permet à d'autres de lire et comprendre les tests.
- La dernière instruction est une assertion.

5.3. Assertions de JUnit5

Il y en a beaucoup en apparence, voir [la liste](#) , mais seulement les deux suivantes sont à connaître pour l'instant :

- `assertEquals(attendu, résultat)` : le résultat doit être égal à attendu
- `assertNotEquals(attendu, résultat)` : le résultat ne doit pas être égal à attendu


Pour comparer des réels, on ajoute un troisième paramètre, `delta` qui est la tolérance pour une petite différence entre attendu et résultat, à cause de la non-exactitude des représentations. Par exemple $0.3 + 0.6$ n'est jamais égal à 0.9, mais à 0.888...89. Donc `assertEquals(0.9, 0.3+0.6)` échoue, mais `assertEquals(0.9, 0.3+0.6, 0.00001)` réussit.

Deux variantes :

- `assertNull(résultat)` : le résultat doit être égal à `null`
- `assertNotNull(résultat)` : le résultat ne doit pas être égal à `null`

5.4. Ajout de JUnit5 à un projet Maven

On repasse à la pratique.

1. Complétez le `pom.xml` de la manière suivante. Attention, il y a des balises en plus dans `<properties>`, `<dependencies>` et `<build>` (attention à bien laisser tout le reste à sa place) : 

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.iutlan.ql</groupId>
  <artifactId>tp1</artifactId>
  <version>0.0.1</version>
  <description>premier TP du module R4.02 qualité logicielle</description>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>${maven.compiler.source}</maven.compiler.target>
    <!-- voir https://stackoverflow.com/a/51838295 -->
    <junit-jupiter.version>5.11.4</junit-jupiter.version>
    <maven-surefire-plugin.version>3.5.2</maven-surefire-plugin.version>
    <maven-assembly-plugin.version>3.7.1</maven-assembly-plugin.version>
  </properties>
```



```
<dependencies>
  <!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-engine -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>${junit-jupiter.version}</version>
    <scope>test</scope>
  </dependency>
  <!-- autres dépendances s'il y en a -->
  <!-- ... -->
</dependencies>

<build>
  <plugins>
    <!-- https://mvnrepository.com/artifact/org.apache.maven.plugins/maven-surefire-plugin -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>${maven-surefire-plugin.version}</version>
    </plugin>
    <!-- autres plugins s'il y en a -->
    <!-- ... -->
  </plugins>
</build>
</project>
```

Si vous êtes perdu.e, voici [le fichier pom.xml complet](#) .

Dans les TP suivants, il y aura encore d'autres dépendances pour avoir des assertions encore plus faciles à écrire.

2. Que faut-il faire après avoir modifié le fichier `pom.xml` ? Voir l'étape 6, page 5.

5.5. Ajout de tests

Vous allez programmer deux tests simples pour commencer.

1. Faites créer une classe de tests unitaires par le menu contextuel de la classe `StringUtils`, *new JUnit Test Case* (il faut cliquer-droit sur le nom de la classe dans le panneau de gauche et si le menu n'apparaît pas, choisir `Other...` puis chercher `JUnit`).

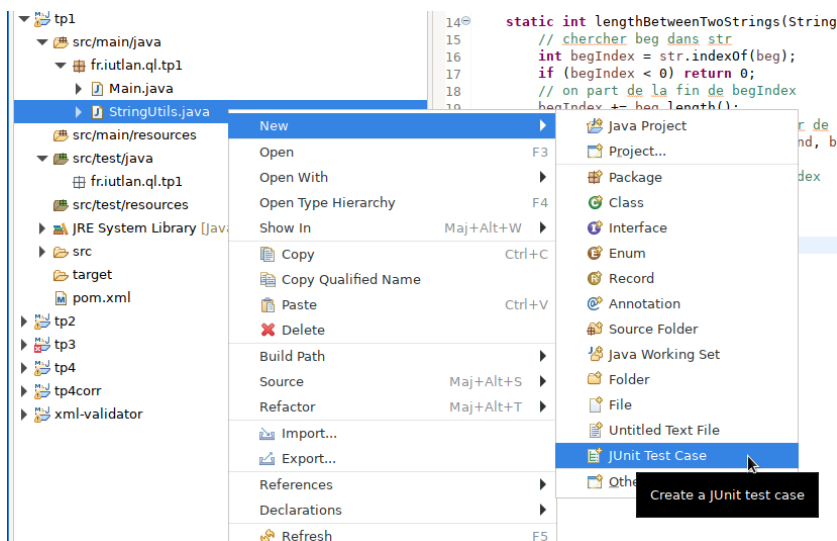


Figure 13: Menu pour créer une classe *JUnit Test Case*

- Il suffit de valider le formulaire de la figure 14, après avoir vérifié en haut que ça sera une *New JUnit Jupiter test*, c'est à dire JUnit 5.

Figure 14: Nom de la classe

Donc maintenant, vous avez les classes suivantes :

- `StringUtils` dans le dossier `tp1/src/main/java/fr/iutlan/ql/tp1`,
- `StringUtilsTest` dans le dossier `tp1/src/test/java/fr/iutlan/ql/tp1`.

- Supprimez la méthode `test` et ajoutez celles-ci :



```
@Test
void lengthBetweenOnAllEmptyMustReturn0() {
```

```
// ARRANGE

// ACT
int result = StringUtils.lengthBetween("", "", "");

// ASSERT
assertEquals(0, result);
}

@Test
void lengthBetweenOnAllOk() {
    // ARRANGE

    // ACT
    int result = StringUtils.lengthBetween("BonjOur tOut Le mOndE", "jOur", "mOn");

    // ASSERT
    assertEquals(9, result);
}

@Test
void lengthBetweenOnMissingBegMustReturnM1() {
    // ARRANGE

    // ACT
    int result = StringUtils.lengthBetween("BonjOur tOut Le mOndE", "JOUR", "mOn");

    // ASSERT
    assertEquals(-1, result);
}
```

NB: il n'y a rien dans *Arrange*, parce que la méthode testée est une méthode de classe.

5.6. Lancement des tests

5.6.1. Avec Eclipse

Il suffit de cliquer droit sur la classe de test ou sur le dossier des tests, et choisir **Run as...** puis **JUnit Test**

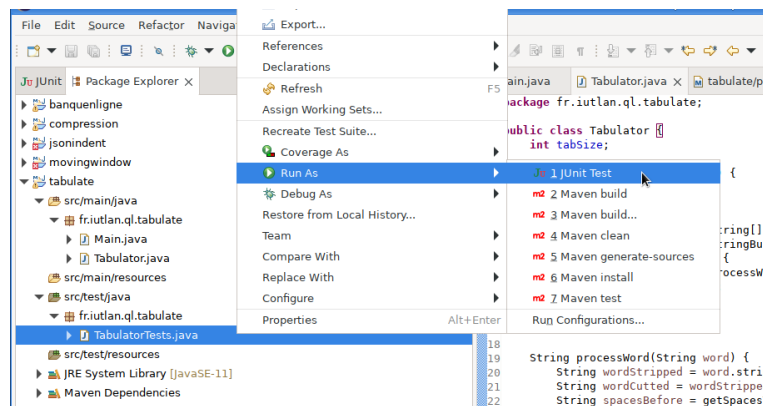


Figure 15: Lancement des tests

Les résultats sont affichés dans un onglet spécifique. Les tests réussis ont un petit icône vert, les échoués sont en bleu foncé, ou rouge s'il y a eu une exception imprévue.

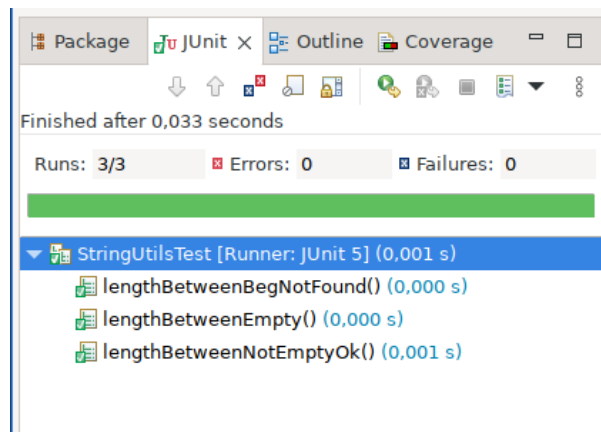


Figure 16: Réussite des tests

Quand il y a un échec, Eclipse affiche toute la pile des fonctions concernées, mais en général, seul le bilan est utile : le résultat n'est pas celui qui est attendu. En cliquant droit sur le message `expected <...>, but was <...>`, on fait apparaître un menu où il faut choisir **Compare Result** pour avoir une fenêtre montrant les différences détaillées.

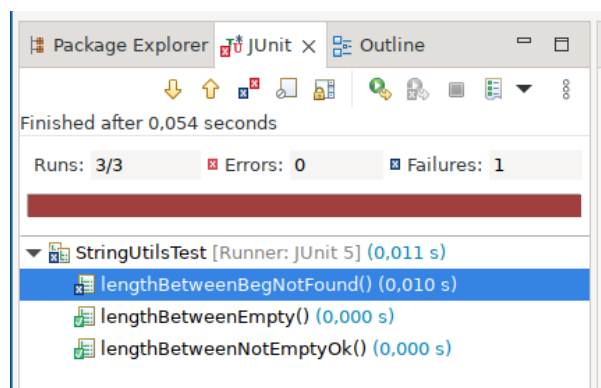



Figure 17: Tests échoués:testsresults}

5.6.2. Avec Maven

C'est un plugin (voir la balise `<plugin>` ajoutée à `pom.xml`) appelée [Maven Surefire Plugin](#) qui se charge de lancer les tests quand on demande la cible `assembly:single`, ou seulement `test`.

Les tests sont tellement cruciaux que Maven ne construit le *jar* que si tous les tests passent.

Comme l'affichage des tests échoués dans le shell est peu pratique, on utilisera Eclipse pour faire les tests tant qu'ils ne passent pas.

1. Lancer les tests dans Eclipse. Vérifiez que tout est au vert (en notant bien que `lengthBetween` ne va pas rester comme ça).
2. Ajoutez la méthode suivante : 

```
@Test
void lengthBetweenOnBegAndEndNullMustReturnM1() {
    // ARRANGE

    // ACT
    int result = StringUtils.lengthBetween("Bonjour tout le monde", null, null);


    // ASSERT
    assertEquals(-1, result);
}
```

6. Le test échoue. Cela nous apprend qu'un cas particulier a été oublié dans `lengthBetween`.
7. Modifiez la méthode `lengthBetween` pour qu'elle retourne -1 et pas une exception quand on lui fournit des `null` en paramètre.

C'est une chose qui n'est pas dans le cahier des charges, car c'est un choix interne plus ou moins involontaire au cours du développement. La personne qui programme les tests doit penser à faire ce genre de tests, pas vraiment prévus, mais qui seront probablement tentés par le client. Cette personne aura l'idée de fournir des paramètres saugrenus, pointant chaque faiblesse possible du programme.

Il pourrait être intéressant de programmer les tests avant les méthodes testées, en suivant le cahier des charges, et en notant les incertitudes et ambiguïtés.

5.7. Travail à faire

Programmez les tests suivants et s'ils prouvent une anomalie, vous devez corriger `lengthBetween`. Consultez la [documentation de `String.indexOf`](#) . Attention, il y a quatre méthodes portant ce nom, et il y a notamment `indexOf(String str)` et `indexOf(String str, int fromIndex)`.

Prenez l'habitude de bien nommer vos tests, de manière à ce que le nom de la méthode permette de comprendre ce que vous testez.

1. Ajoutez tous les tests qui permettent de vérifier que `lengthBetween` résiste à des `null` en paramètre(s). Corrigez `lengthBetween` tant que ces tests ne passent pas.
2. Ajoutez aussi les tests qui vérifient ce qui se passe quand ce sont des chaînes vides qu'on fournit. Idem (corrigez...).

3. Complétez les tests qui vérifient ce qui se passe quand `beg` et/ou `end` ne sont pas dans la chaîne. Idem.
4. Il faudrait tester ce qui se passe s'il y a plusieurs occurrences des chaînes `beg` et `end`. Idem.
5. Avez-vous essayé de voir ce qui se passe quand `beg` et `end` sont consécutifs ?

6. Tests d'exceptions

On va maintenant écrire quelques tests pour une nouvelle méthode, `substringBetween`, qui retourne une exception dans certains cas.

1. Ajoutez cette nouvelle méthode à la classe `StringUtils`



```
/**
 * extrait une sous-chaîne de str comprise entre beg et end
 * c'est à dire qu'avec str = "XXXXXbegYYYYYYendZZZZZ", il faut retourner
 * la sous-chaine YYYYYY
 * @throws NullPointerException si str, beg ou end sont null
 * @throws IllegalArgumentException si beg ou end sont absents de str
 * @param str chaine dans laquelle on cherche beg puis end
 * @param beg sous-chaîne définissant le début (première occurrence)
 * @param end sous-chaîne définissant la fin (première occurrence après beg)
 * @return la sous-chaine entre beg et end (exclus) dans str
 */
static String substringBetween(String str, String beg, String end)
throws RuntimeException {
    // chercher beg dans str
    int begIndex = str.indexOf(beg);
    if (begIndex < 0) throw new IllegalArgumentException(beg+" not found");
    // on part de la fin de begIndex
    begIndex += beg.length();
    // chercher end dans str à partir de beg
    int endIndex = str.lastIndexOf(end);
    if (endIndex < 0) throw new RuntimeException(end+" not found");
    // sous-chaine entre endIndex et begIndex
    return str.substring(begIndex, endIndex);
}
```

On souhaite vérifier que cette méthode émet les exceptions prévues si les paramètres sont incorrects.

C'est plus compliqué, car une exception fait normalement échouer la méthode de test, or là, ce qu'on veut, c'est que le test réussisse quand il y a une exception précise, et qu'il échoue s'il n'y a pas d'exception. Cela s'écrit ainsi en JUnit5, en réorganisant le patron *AAA* :



```
@Test
void method2OnNullMustThrowNullPointerException() {
    // ARRANGE
    CLASSE obj = new CLASSE();

    // ASSERT
```

```
assertThrows(NullPointerException.class, () -> {  
  
    // ACT  
    obj.method2(null);  
});  
}
```

Ça fait appel à une fonction *lambda* `() -> {...}`. C'est un bloc d'instructions qui est appelé par `assertThrows`.

2. Programmez le test `substringBetweenOnDebNullMustThrowNullPointerException` qui vérifie que `substringBetween` émet une `NullPointerException` quand le paramètre `beg` est `null`.
3. Programmez le test `substringBetweenOnMissingDebMustThrowIllegalArgumentException` qui vérifie que `substringBetween` émet une `IllegalArgumentException` quand le paramètre `beg` est absent de la chaîne.
4. Programmez le test `substringBetweenOnMissingEndMustThrowIllegalArgumentException` qui vérifie que `substringBetween` émet une `IllegalArgumentException` quand le paramètre `end` est absent de la chaîne. Constatez que l'exception émise n'est pas bonne. Corrigez la méthode fautive.

7. Rendu du TP

Vous devrez remettre votre travail à la fin de chaque séance.

Ouvrez un terminal bash dans le dossier de votre TP, là où il y a le fichier `pom.xml`, puis tapez ceci :

```
mvn clean  
cd ..  
tar cfvz tp1.tgz tp1
```

Puis déposez le fichier `tp1.tgz` dans la zone de dépôt du TP1 sur [Moodle R4.02 Qualité de développement](#) .

ATTENTION votre travail est personnel. Si vous copiez pour quelque raison que ce soit le travail d'un autre, vous serez tous les deux pénalisés par un zéro et la menace de finir par un devoir sur table (avec tout à apprendre par cœur).

En cas de souci quelconque, envoyez un mail à pierre.nerzic@univ-rennes1.fr pour expliquer le problème.