# TP 4

## Hotels

Le but de ce TP est :

- ▷ d'apprendre à écrire et à générer une documentation « javadoc »,
- ▷ d'apprendre à coder une classe en s'appuyant sur des tests,
- ▷ de se familiariser avec l'écriture de tests par l'étude de tests fournis.

Ce travail sera à rendre dans votre dépôt dans un dossier qui devra s'appeler `tp4/`. Voir les consignes en fin de sujet.

Pour ce TP, les tests sont fournis et vous n'aurez donc pas à écrire de tests. Mais, vous devez étudier leur code pour le comprendre et ainsi vous familiariser avec leur écriture. Ces exemples pourront être une source d'inspiration pour les tests que vous aurez à écrire à l'avenir. De plus vous devrez exécuter ces tests pour valider votre code.

Vous devez commencer par récupérer sur le portail l'archive fournie, en extraire les fichiers et les placer dans votre dossier nommé `tp4/`.
Votre dossier contient désormais un dossier `src/` qui est destiné à contenir les fichiers sources des classes et un dossier `test/` qui regroupe les classes de test. Parmi les fichiers extraits se trouve aussi le fichier `test4poo.jar` qui permet l'exécution des tests.

**Contexte.**
Dans ce TP on s'intéresse à des hôtels composés d'un certain nombre de chambres numérotées à partir de 1.
Un hôtel a également un nom et un « statut » (*confort*, *cosy* ou *premium*).
Il est possible de louer une chambre et de la libérer.
Deux chambres avec le même numéro seront considérées comme égales.

**Documentation.**

**Q 1 .** Dans un éditeur, ouvrez le fichier `src/Room.java` qui définit la classe `Room`.

Consultez ce code.

Comme vous le constatez la documentation et le code des méthodes sont fournis.

**Q 2 . Syntaxe de la documentation**.

La documentation dite « javadoc » d'un élément de code correspond au bloc de commentaire placé **juste avant** cet élément. Ce bloc est délimité par `/**` et `*/`.

Comme déjà vu dans les exemples des premiers TP, vous constatez que les paramètres et la valeur de retour sont identifiés par les *tags* `@param`[1] et `@return`. Les commentaires javadoc peuvent accueillir du code html qui sera interprété lors de la génération de la documentation (voir par exemple la « javadoc » de la méthode `isRent()`).

Le texte de cette documentation doit permettre l'utilisation de la méthode. Il faut donc décrire ce que fait la méthode, le rôle de chaque paramètre et la nature du résultat renvoyé, quand il y en a.

Familiarisez-vous avec la syntaxe de cette documentation en consultant celle rédigée pour chacun des éléments de cette classe.

**Q 3 . Génération de la documentation.**

L'intérêt de la syntaxe de ces commentaires est de permettre la génération d'une documentation formatée.

Cela se réalise à l'aide de l'outil `javadoc`.

Depuis le dossier `tp4/` exécutez la commande :

```
.../tp4> javadoc src/Room.java -sourcepath src -d docs
```

L'option « `-sourcepath src` » indique que les fichiers source se trouvent dans le dossier `src/`.

L'option « `-d docs` » permet de placer les fichiers de doumentation générés dans un dossier `docs/` qui est créé s'il n'existe pas (« `d` » comme *destination*).

[1] Dans le cas particulier de cette classe, il y a au plus un paramètre par méthode. Dans les cas à plusieurs paramètres, il faut alors répéter le tag `@param` pour chacun des paramètres.

**Q 4 . Consulation de la documentation.**

Consultez le contenu du dossier `docs/`. En particulier ouvrez dans un navigateur le fichier `Room.html` qu'il contient. Vous devez pouvoir faire le lien entre la page affichée et le code de documentation qui apparaît dans le fichier source `Room.java`.

Parcourez ce fichier.

N'hésitez pas modifier des éléments de la documentation dans le fichier `Room.java`, puis à générer à nouveau la documentation pour constater l'évolution dans le fichier `Room.html` après l'avoir actualisé dans le navigateur.

Dans la suite vous devrez systématiquement écrire des documentations au format javadoc pour chacune des classes que vous créerez.

**Premiers tests unitaires.**

**Q 5 .** Consultez le contenu du fichier de test `test/RoomTest.java`.

Il contient différentes méthodes de test, identifiables car préfixées par `@Test` :

- `numberAndRentAreCorrectAtCreation()` permet de tester que le processus de création est correct.
- `rentRoomTest()` et `freeRoomTest()` permettent de vérifier le bon comportement des méthodes `rent()` et `free()`.
- `roomsAreEqualsIfSameNumber()` et `roomsAreNotEqualsIfNotSameNumber()` permettent de tester le bon comportement de la méthode `equals()`.

Vous constatez que les méthodes de tests contiennent une ou plusieurs assertions identifiées par les méthodes `assertEquals()`, `assertTrue()` et `assertFalse()`, dont on comprend facilement le sens.

On remarque aussi que, comme vu en cours, ces méthodes adoptent la structure :

1. mise en place de la situation initiale, avec en particulier la création des objets nécessaires au test,
2. vérification de précondition, à l'aide d'assertions,
3. exécution de la méthode testée,
4. vérification de postcondition, à l'aide d'assertions, pour vérifier et valider l'effet de la méthode testée.

D'autres méthodes d'assertion existent[2].

Un test (c-à-d. une méthode de test) est réussi, et donc passé avec succès par le code qu'il évalue, quand toutes les assertions qu'il contient sont vérifiées.

**Remarque : notez les trois lignes à la fin du fichier. Vous devrez placer des lignes équivalentes à la fin de chacun de vos fichiers de test en remplaçant bien sûr le « `RoomTest` » qui apparaît ici par le nom de la classe de test définie par le fichier.**

**Q 6 . Compilation du code source de la classe.**

Avant de pouvoir exécuter les tests de `RoomTest` et ainsi vérifier si le code de la classe proposée pour la classe `Room` est correct, il faut avoir compilé le fichier `Room.java` .

Compilez la classe `Room.java` en exécutant la commande suivante depuis le dossier `tp4/` :

```
.../tp4> javac src/Room.java -sourcepath src -d classes
```

Le rôle des options « `-sourcepath` » et « `-d` » est le même que dans le cas de la commande `javadoc`.

L'utilisation de l'option `-d` permet de ne pas mélanger les fichiers sources (`.java`), la documentation (`.html`) et les fichiers compilés (`.class`). Cela facilite la gestion des fichiers d'un projet et notamment du dépôt git.

**Q 7 . Compilation des tests.**

Il est ensuite nécessaire de compiler le fichier en exécutant la commande suivante depuis le dossier `tp4/` :

```
.../tp4>javac -classpath test4poo.jar test/RoomTest.java
```

L'option `-classpath` permet de prendre en compte l'archive `test4poo.jar` qui est nécessaire pour les tests[3].

**Remarque : vous devrez toujours placer vos classes de test dans un sous-dossier `test/` du dossier qui contient le fichier `test4poo.jar`.**

[2] Consultez le document sur le portail, zone *Documents*, pour les découvrir.

[3] Pour réaliser nos tests unitaires, nous utilisons la bibliothèque `JUnit4` qui est très utilisée. Les tests tels que vous les apprenez sont des tests `JUnit4`.
Cependant le fichier `test4poo.jar` en est une adaptation locale pour vous permettre de disposer de la fenêtre graphique des résultats.

**Q 8 . Exécution des tests.**

On peut alors exécuter les tests `RoomTest` grâce à la commande suivante (toujours depuis le dossier `tp4/`) :

```
.../tp4>java -jar test4poo.jar RoomTest
```

Une fenêtre s'ouvre. Vous y trouvez une barre verte qui signifie que tous les tests ont été passés avec succès. Ce qui est confirmé par les indications fournies juste dessous :

- le nombre de tests exécutés (*Runs*) (ici 5),
- le nombre de tests en échecs (*Errors*) (ici 0).

**Q 9 .** Nous allons voir ce qu'il se passe lorsqu'un test est en échec.

Pour cela il faut introduire une erreur dans le code de la classe `Room`. Modifiez la ligne 35 du fichier `Room.java` ainsi : « `this.rent = false;` ».

Pour pouvoir prendre en compte la nouvelle définition de la classe `Room`, il faut la compiler. Compilez donc cette classe en reprenant la commande indiquée ci-dessus.

Les tests n'ayant pas été modifiés, il n'est pas nécessaire de les recompiler. Une fois la classe `Room` recompilée, on peut donc exécuter les tests en reprenant la commande ci-dessus.

Cette fois, la fenêtre qui s'ouvre contient une barre rouge qui signifie qu'il y a des erreurs[4], ce qui est confirmé par l'indication Errors : 2.

Dans la première zone de texte (Results:), les méthodes de test en échec sont citées (`rentRoomTest()` et `freeRoomTest()`). En sélectionnant l'une de ces méthodes, vous obtenez dans la seconde zone de texte la trace de l'erreur, avec en particulier l'indication de la ligne de test qui pose problème.

**Q 10 .** Restaurez la version correcte du code de `Room.java` puis recompilez la classe.

**La classe `Hotel`.**

**Q 11 .** Procédez comme précédemment pour **compiler le fichier source `Hotel.java`, puis le fichier de test `HotelTest` avant d'exécuter ces tests** :

```
.../tp4>java -jar test4poo.jar HotelTest
```

Cette fois la barre est rouge car il y a des échecs dans les tests. C'est en fait le cas de tous les tests.

**Pour la suite du TP l'objectif est « *obtenir une barre verte* »**

Dans la suite du TP l'objectif est d'avoir de moins en moins de tests en erreur après chaque question et donc à la fin du TP d'obtenir une barre verte qui indique que tous les tests ont été passés avec succès et donc que votre implémentation de la classe `Hotel` satisfait le cahier des charges.
Pour l'évaluation de votre TP, ces tests seront exécutés.

**Objectif « barre verte »...**
Dans les questions suivantes, pour chacune des méthodes à implémenter, vous commencerez par écrire la documentation de cette méthode, si elle n'existe pas encore, puis vous étudierez la ou les méthodes de test qui lui sont associées dans `HotelTest.java` **avant** de procéder à l'implémentation de la méthode.
Vous validerez votre implémentation en vérifiant que le test correspondant à cette méthode est passé avec succès.
Vous ne devez passer à la question suivante que si tous les tests liés à une question ont été passés avec succès.
**Attention : il est nécessaire de recompiler le code source après chaque modification de code pour que celle-ci soit prise en compte.**

**Implémentation des méthodes.**

**Q 12 .** Complétez le constructeur de la classe `Hotel` pour initialiser l'attribut `rooms` en créant les objets correspondant aux chambres de cet hôtel.

**Q 13 .** Ecrivez le code de la méthode `getRoom()`. On rappelle que la première chambre doit avoir le numéro 1.

Compilez la classe `Hotel` et exécutez les tests de `HotelTest`.

La méthode de test `roomsAreCreatedAtCreation()` doit maintenant être passée avec succès.

**Q 14 .** Ecrivez le code de la méthode `numberOfRooms()`.

Compilez puis testez. La méthode de test `numberOfRoomsIsCorrectAtCreation()` doit maintenant être passée avec succès.

[4]Notez que la barre est soit toute verte, soit toute rouge, il n'y a pas coloration proportionnelle au nombre de tests réussis.

**Q 15 .** La méthode `rentRoom()` permet la location d'une chambre de l'hôtel. Le numéro de la chambre à louer est passé en paramètre. Le résultat de cette méthode est la chambre louée quand c'est possible. Ce résultat vaut `null` si la chambre demandée est déjà louée ou si le numéro de chambre n'est pas valide (négatif ou trop élevé).

Ecrivez la documentation de la méthode `rentRoom()`.

Identifiez dans `HotelTest` les méthodes de test qui permettent de valider la méthode `rentRoom` et étudiez leurs codes.

Ecrivez le **code** de la méthode `rentRoom()`.

Compilez puis testez pour valider votre implémentation.

**Q 16 .** Pour chacune des méthodes suivantes, écrivez la javadoc, étudiez les tests correspondant puis écrivez le code. A chaque fois, compilez puis testez.

- `leaveRoom()`, qui permet de rendre une chambre louée. Il n'y a aucun effet si la chambre n'était pas louée.
- `nombreOfFreeRooms()` dont le résultat est le nombre de chambres libres dans l'hôtel.
- `firstFreeNumber()` dont le résultat est le plus petit numéro d'une chambre libre ou 0 si aucune chambre n'est libre.

**Q 17 .** Définissez une classe `HotelMain` qui définit une méthode `main` (voir TP 3) qui permette d'exécuter cette classe en prenant en argument un entier. Cet entier représentera un numéro de chambre.

Depuis le dossier `tp4/`, après compilation, l'exécution de cette classe devra se faire avec la commande (ici pour la chambre 42) :

```
.../tp4> java -classpath classes HotelMain 42
```

L'option « `-classpath classes` » permet d'indiquer au programme `java` dans quel dossier trouver les définitions compilées des classes.

Le programme devra se comporter ainsi :

- créer un objet `Hotel` correspondant à l'*Hôtel California prémium* de 100 chambres,
- afficher le nombre de chambres de cet hôtel,
- louer (et récupérer) la chambre dont le numéro a été passé en argument,
- afficher la chambre louée,
- afficher le nombre de chambres libres dans l'hôtel,
- libérer la chambre louée,
- afficher à nouveau le nombre de chambres libres dans l'hôtel.

Compilez puis exécutez votre programme avec différents numéros de chambre.

Vous ferez attention de prendre en compte l'absence de paramètre dans la ligne d'exécution en affichant dans ce cas un message précisant le bon usage du programme.

## Rendre le travail sur gitlab (complété)

Dans votre dépôt local créez un dossier `tp4/` et placez-y :

- un fichier `readme.md`[5] où vous indiquerez de manière structurée (ce sera à faire pour chaque TP) :
  - les noms des membres du binômes
  - un paragraphe présentant le TP et ses objectifs
  - un paragraphe précisant comment générer et consulter la documentation
  - un paragraphe précisant comment compiler les classes du projet
  - un paragraphe précisant comment compiler et exécuter les tests
  - un paragraphe précisant comment exécuter le programme (le ou les `main` inclus), en donnant des exemples,

  **A chaque fois vous indiquerez précisément la ou les commandes à exécuter.**
- les différents fichiers de votre projet :
  - les fichiers `.java` dans le dossier `src/`
  - le fichier `test4poo.jar`
    **NB :** si votre `.ignore` est correctement paramétré, il faudra forcer l'ajout avec l'option `-f` (pour « **f**orcer ») lors du `add` : « `git add -f test4poo.jar` »

  **Ne pas déposer dans votre dépôt GitLab** les contenus des dossiers `docs/` et `classes/`, car les fichiers `.class` et de documentation peuvent être générés. L'exclusion de ces fichiers doit être gérée par le fichier `.gitignore` de votre dépôt comme présenté dans le document d'introduction à GitLab.

[5] voir les fiches de synthèse de la syntaxe markdown dans la zone document du portail