

# TP : Un robot fortement typé ...

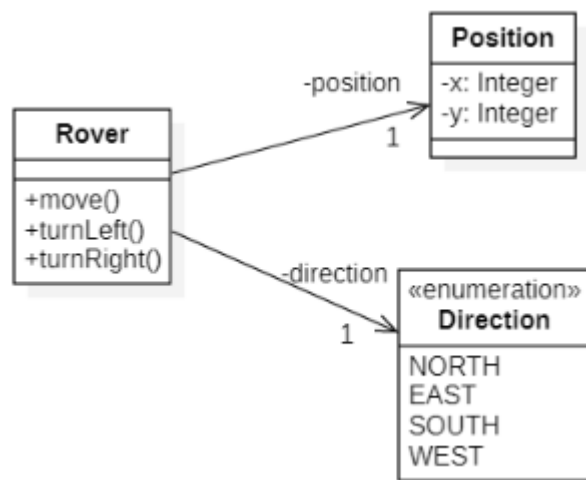
*Vous allez participer au développement du projet qui va piloter le prochain robot (rover) sur la lune.*

*Pour simplifier le problème, on considère dans un premier temps que le robot se déplace sur une carte 2D illimitée. Le robot démarre toujours du centre de la carte c.-à-d. en (0,0) et il est dirigé vers le nord.*

*Le robot (rover) doit pouvoir avancer, tourner à droite et gauche.*



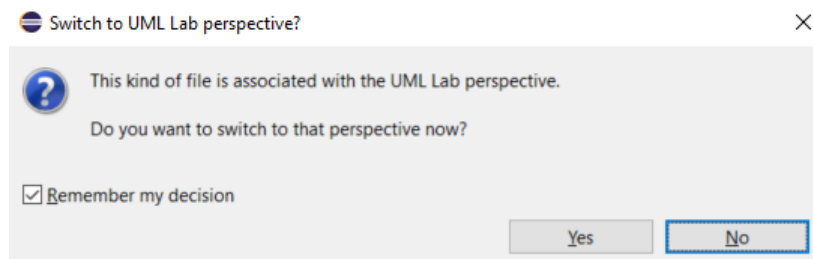
**La phase de conception** menée en collaboration avec votre équipe de développement vous a conduit au diagramme de classes suivant :



Votre équipe a décidé de vous confier **l'implémentation java** de cette conception.

➔ Commencez par créer, sous Eclipse, un **nouveau projet java** que vous appellerez **rover**.

Pour finir, cliquez sur **Don't create** quand l'IDE vous demandera si vous souhaitez un module.  
( toujours pas de module : **Don't create** 😊 )



**Remarque :** au démarrage d'Eclipse, **ne switchez pas dans la perspective UML Lab**, restez dans la perspective Java et demandez à ce que votre décision soit mémorisée.

Comme vu au TP précédent, pour travailler plus sereinement, pensez bien fermer dans la vue **Package Explorer** tout autre projet non relatif au projet en cours c.-à-d. **rover** (Le clic droit est votre ami 😊).

→ **Implémentation :** Implémentez ensuite dans le **src** de ce projet, les différences classes de manière à respecter le diagramme de classes précédent.

❑ A propos de **Direction** :

Vous pouvez directement créer une **énumération** en choisissant **New → Enum** (au lieu de **New → Class** pour créer une **classe**).

Comme c'est votre première énumération, nous vous fournissons le code suivant qui est l'implémentation la plus simple pour l'énumération **Direction** :

```
public enum Direction {  
    NORTH, EAST, SOUTH, WEST;  
}
```

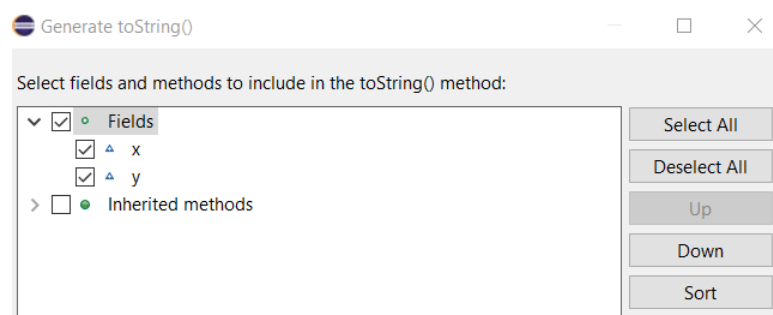
❑ A propos de **Position** :

→ N'oubliez pas d'implémenter **un constructeur**.

→ Pour cette classe, est-il intéressant de déclarer les attributs en **final** ?  
Suivant votre réponse, implémentez (ou **générez** à l'aide du menu **Source**) :

- Soient les **getters/setters**
- Soient seulement les **getters**

→ Générez **toString** à l'aide de votre IDE : **Source → Generate toString()...**



Contentez-vous pour cet exercice de cocher **simplement les Fields** pour que **toString** renvoie uniquement pour le moment l'état interne de l'objet (sa valeur en quelque sorte) 😊.

**Rappel :** les **getters**, **setters** et la méthode **toString** n'apparaissent généralement pas dans le diagramme de classes pour ne pas alourdir la notation, mais il faut prendre l'habitude de les ajouter à l'implémentation (si nécessaire : se poser notamment la question pour les **setters** 😊).

❑ A propos de **Rover** :

→ Les **attributs** doivent-ils être déclarés en **final** ?

→ Implémentez 3 **constructeurs** pour pouvoir lors des jeux d'essais, implémenter les robots de la manière suivante :

```
Rover viper = new Rover();  
Rover python = new Rover(new Position(5, 10), Direction.EAST);  
Rover anaconda = new Rover(20, 30, Direction.SOUTH);
```

**Qualité de code** : Pour éviter le **duplication de code** dans les constructeurs, vous ferez en sorte les constructeurs s'appellent les uns et les autres 😊

→ Implémentez les **getters** (pour les **setteurs**, on verra plus tard s'ils sont vraiment nécessaire) 😊.

→ Générez la méthode **toString**.

→ Pour les méthodes offrant les services **move**, **turnLeft** et **turnRight**, contentez-vous pour l'instant d'écrire juste un commentaire **//TODO** dans le corps de ces méthodes.

Notez bien que **TODO** est un mot connu par l'IDE car il s'affiche en bleu 😊)

Nous reviendrons implémenter ces méthodes dans quelques minutes 😊.

## → Tests (et exécution)

Exécutez et testez les trois jeux d'essai suivant :

**Astuce raccourci IDE** : Pour éviter de taper à chaque fois `System.out.println()` ; Utilisez et abusez du raccourci suivant : **syso** suivi de **CTRL + Espace**

```
public class Main {  
  
    public static void main(String[] args) {  
  
        Rover viper = new Rover();  
        System.out.println("viper is actually : ");  
        System.out.println(viper.getPosition());  
        System.out.println(viper.getDirection());  
        System.out.println("-----");  
    }  
}
```

```

Rover python = new Rover(new Position(5, 10), Direction.EAST);
System.out.println("python is actually : ");
System.out.println(python.getPosition());
System.out.println(python.getDirection());
System.out.println("-----");

Rover anaconda = new Rover(20, 30, Direction.SOUTH);
System.out.println("anaconda is actually : ");
System.out.println(anaconda.getPosition());
System.out.println(anaconda.getDirection());
System.out.println("-----");

}

}

```

Ces jeux d'essais doivent vous conduire à l'affichage console suivant :

```

viper is actually :
Position [x=0, y=0]
NORTH
-----
python is actually :
Position [x=5, y=10]
EAST
-----
anaconda is actually :
Position [x=20, y=30]
SOUTH
-----

```

Ajoutez ensuite dans chacun des trois jeux d'essais l'appel à la méthode `toString` après l'appel à `getDirection` de manière à visualiser sur la console pour **viper** par exemple :

```

viper is actually :
Position [x=0, y=0]
NORTH
Rover [position=Position [x=0, y=0], direction=NORTH]
-----

```

## → Demander sa localisation à un robot

Votre client vient vous voir en catastrophe car il a oublié de vous demander une méthode qui permet à un robot de donner sa localisation 😊. Pouvez-vous faire cela rapidement ?

Vous êtes alors tenté de lui dire : « Pas de problème, j'ai déjà ce service, il s'appelle **toString** »  
Votre collègue vous pose alors la question suivante : « Trouves-tu que le terme **toString** reflète la demande du client c.-à-d. la *demande de localisation* ? Ne trouves-tu pas que **toString** est un peu technique et pas très *user-friendly* dans son affichage ? »

Il vous fait part de la bonne pratique suivante :

**Bonne pratique** : la méthode **toString** ne doit pas être utilisée à des fins fonctionnelles.  
Cette méthode est plutôt *technique* et doit être utilisée par le développeur pour **récupérer un état de l'objet** (et/ou son adresse mémoire comme on le verra avec le concept d'héritage)

Il n'est donc pas recommandé d'utiliser **toString** pour répondre à cette demande du client.  
**Donner la localisation** est bien un nouveau service qui doit être offert par votre classe **Robot**.

⇒ Il ne vous reste donc plus qu'à **implémenter une nouvelle méthode** pour répondre à ce besoin fonctionnel de votre client dont la signature sera : **public String getLocation()**

⇒ Il ne vous reste donc plus qu'à **tester cette méthode** en ajoutant un appel à la méthode **getLocation** au début des 3 jeux d'essais à la place des **:** de manière à avoir maintenant :  
`System.out.println("viper is actually " + viper.getLocation());`

pour pouvoir visualiser sur la console :

```
viper is actually at position (x=0, y=0) towards the NORTH
Position [x=0, y=0]
NORTH
Rover [position=Position [x=0, y=0], direction=NORTH]
-----
python is actually at position (x=5, y=10) towards the EAST
Position [x=5, y=10]
EAST
Rover [position=Position [x=5, y=10], direction=EAST]
-----
anaconda is actually at position (x=20, y=30) towards the SOUTH
Position [x=20, y=30]
SOUTH
Rover [position=Position [x=20, y=30], direction=SOUTH]
-----
```

## → Permettre à un robot de se déplacer et de tourner

c.-à-d. implémenter les **services/méthodes métier** offert(e)s par la classe Rover

Outre la localisation, d'après le diagramme de classes initial, il doit être possible de **demandeur à un robot d'avancer et de tourner à droite ou à gauche**.

Mais où en est-on avec l'implémentation de ces services fonctionnels de la classe **Rover** ?

Imaginez-vous que le code du projet **rover**, que vous avez actuellement sous vos yeux, ait été écrit la semaine dernière par un de vos collègues. A ce moment-là, vous étiez sur autre projet (celui des pionniers) et vous n'avez pas participé à l'écriture de ce code 😊.

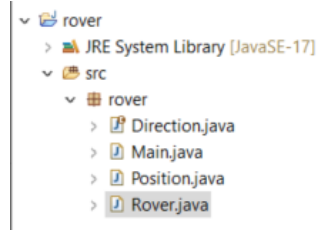
Aujourd'hui, votre équipe vous demande de prendre le relais sur le projet **rover** pour finir son implémentation. Bien sûr on vous a fourni le diagramme de classes de la question 1.

Vous vous demandez : mais où en est l'implémentation de ce projet ?... parce que vous ne savez pas ce qu'il vous reste à implémenter sur ce projet.

### Bonnes pratiques pour se (re)plonger dans un projet existant :

#### - 1. Que contient le projet ?

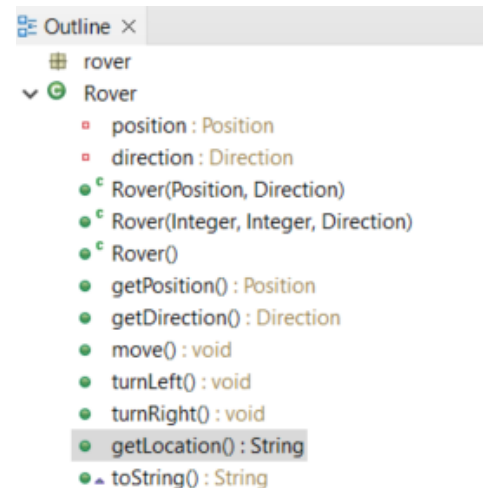
⇒ Commencez par consulter la **vue Package Explorer** et déroulez **src** pour savoir quelle(s) **classe(s) existe(nt) déjà** dans le projet.



⇒ Pour visualiser **en un coup d'œil le contenu de chaque classe**, vous pouvez utiliser la **vue Outline**, qui normalement apparaît sur le côté droit de l'IDE.

Si ce n'est pas le cas, deux solutions s'offrent à vous :

- Remettez-vous dans la perspective Java initiale. Pour cela, rien de plus simple : **Window → Perspective → Reset Perspective** puis cliquez sur **Reset Perspective**. La perspective Java par défaut propose la vue **Outline** à droite.
- Vous pouvez faire apparaître directement la vue à partir du menu **Window → Show View → Outline**



Ensuite, il vous suffit de vous placer sur une classe sur la vue **Package Explorer**, par exemple la classe **Rover** et vous visualiserez en un clin d'œil dans la vue **Outline**, les attributs et les signatures des méthodes que contient cette classe.

**Bon à savoir** : si vous cliquez sur un élément de la vue **Outline**, l'éditeur de code positionnera votre curseur à l'endroit de ces éléments dans le code 😊 et si vous déplacez un élément dans la vue **Outline**, il sera également déplacé dans le code 😊.

⇒ Vous pourriez également procéder à un petit **reverse-engineering sur le code** du projet pour vous donner une idée de l'architecture du projet et des relations/dépendances entre les classes : Nous ne vous demandons pas de le faire maintenant, continuez la lecture 😊.

## - 2. Qu'est-ce qui est fonctionnel/opérationnel (qu'est-ce qui marche actuellement dans le projet) ?

Pour savoir ce qui marche actuellement dans le projet, il vous suffit d'espérer que des jeux d'essais aient bien été écrits par le développeur précédent (pensez à ceux qui se replongeront dans votre code dans quelques mois...) et de les lancer !

Pour nous, la classe **Main** fait actuellement office pour le moment de classe de tests avec ses jeux d'essais manuels....

- En lisant les jeux d'essais, vous pouvez commencer à faire un point sur le code déjà écrit et celui qui reste à implémenter...
- En exécutant les jeux d'essais, vous vérifiez que dans le code écrit tout fonctionne normalement. Si vous faites face à un bug à l'exécution des jeux d'essais, il faudra corriger ce bug, avant de procéder à toutes extension du programme c.-à-d. avant d'ajouter du nouveau code 😊.

Heureusement pour vous aujourd'hui, la classe **Main** s'exécute sans problème 😊.

## - 3. Que reste-t-il à faire ?

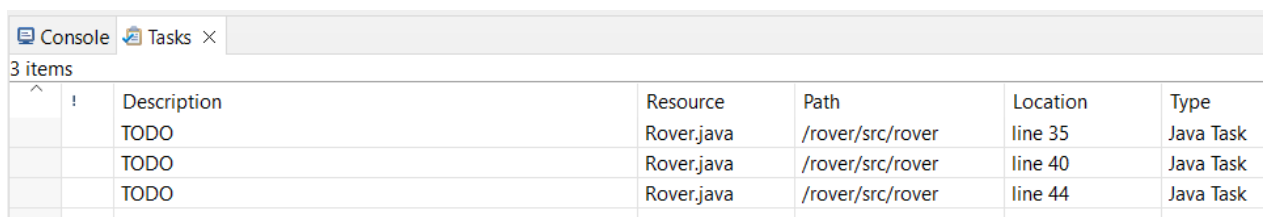
⇒ En reprenant la spécification (qui est actuellement pour nous le **diagramme de classes** élaborés pendant la phase de conception) et en tenant compte de tout ce qui précède, vous devriez commencer à avoir une idée du travail qu'il reste à réaliser pour finaliser cette implémentation.

⇒ On ne sait jamais, le développeur précédent (ou vous-même) a peut-être laissé des marqueurs sur le travail qu'il reste à terminer dans le code 😊

A tout hasard, vous pouvez donc utiliser le menu suivant de votre IDE :

**Window → Show View → Tasks**

Si vous avez laissé des commentaires **//TODO** dans votre code (comme précédemment) toutes les lignes de code contenant **//TODO** seront listés dans la vue **Task** comme celle présentée ici :



The screenshot shows the IDE's Task view with a tab labeled 'Tasks'. Below the tab, it says '3 items'. A table lists the tasks:

	Description	Resource	Path	Location	Type
!	TODO	Rover.java	/rover/src/rover	line 35	Java Task
	TODO	Rover.java	/rover/src/rover	line 40	Java Task
	TODO	Rover.java	/rover/src/rover	line 44	Java Task

Cliquez sur un **TODO** vous amènera directement dans le code à l'endroit de ce commentaire.

Il ne vous reste plus maintenant qu'à procéder à l'implémentation et à l'exécution et tests des méthodes **move**, **turnLeft**, **turnRight** afin de pouvoir exécuter les deux jeux d'essais suivant :

-----  
Test cases about turnRight and move  
-----

viper is actually at position (x=0, y=0) towards the NORTH  
viper is turning right  
now at position (x=0, y=0) towards the EAST  
viper is moving twice  
now at position (x=2, y=0) towards the EAST  
viper is turning right  
now at position (x=2, y=0) towards the SOUTH  
viper is moving once  
now at position (x=2, y=-1) towards the SOUTH  
viper is turning right  
now at position (x=2, y=-1) towards the WEST  
viper is moving twice  
now at position (x=0, y=-1) towards the WEST  
viper is turning right  
now at position (x=0, y=-1) towards the NORTH  
viper is moving once  
now at position (x=0, y=0) towards the NORTH

-----  
Test cases about turnLeft and move  
-----

viper is actually at position (x=0, y=0) towards the NORTH  
viper is turning left  
now at position (x=0, y=0) towards the WEST  
viper is moving twice  
now at position (x=-2, y=0) towards the WEST  
viper is turning left  
now at position (x=-2, y=0) towards the SOUTH  
viper is moving once  
now at position (x=-2, y=-1) towards the SOUTH  
viper is turning left  
now at position (x=-2, y=-1) towards the EAST  
viper is moving twice  
now at position (x=0, y=-1) towards the EAST  
viper is turning left  
now at position (x=0, y=-1) towards the NORTH  
viper is moving once  
now at position (x=0, y=0) towards the NORTH

**Hypothèse** : Pour cette première implémentation de **move**, on considère que l'espace de déplacement du robot est infini et qu'il n'y a pas de limite pour x et y 😊 .



## → Une lecture « plus fluente » du code : Un petit renommage pour la forme ?

Dans votre code, vous devriez avoir écrit à moment donné quelque chose comme :

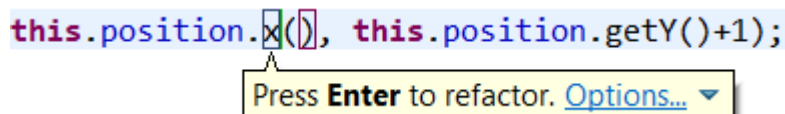
```
this.position = new Position (this.position.getX()+1, this.position.getY());
```

peut-être ne trouvez-vous pas cette instruction très lisible et peut-être préféreriez-vous lire :

```
this.position = new Position (this.position.x()+1, this.position.y());
```

Rien de plus simple, l'IDE va vous aider à **renommer** tous les `getX()` et `getY()` en `x()` et `y()` dans tous les fichiers de votre projet en un seul clic !

Pour cela, placez le curseur sur un `getX()` de votre code (n'importe lequel) et à l'aide d'un **clic droit**, sélectionnez le menu **Refactor** → **Rename** et entrer **x** à la place de `getX` puis taper sur Entrée.



Bravo, vous venez de renommer automatiquement votre première méthode !!!

Via ce menu, l'IDE a fait les changements automatiques dans le fichier **Rover**, mais aussi dans le fichier **Position** (vous pouvez aller jeter un petit coup d'œil à ces deux fichiers pour vérifier 😊)

**Bonne pratique** : le menu **Rename** de votre IDE vous sera très utile.

Vous devriez dorénavant l'utiliser très souvent,

il est donc recommandé d'apprendre son **raccourci clavier** sous votre IDE préféré :

Le raccourci clavier de **Rename** pour Eclipse est **ALT + SHIFT + R**

⇒ A l'aide du raccourci clavier **ALT + SHIFT + R**, renommez maintenant `getY` en `y` !

**Remarque** : Comment écrire les getteurs : `getX()` ou directement `x()` ?

- Dans les premières versions de java, il était d'usage de toujours nommer les getteurs et setteurs : **getX** et **SetY**. Cette norme est/était d'ailleurs obligatoire si vous développ(i)ez des **EJB** (**Entreprise Java Bean**), des composants qui respectent les spécifications d'un modèle édité par Sun (l'entreprise qui a développé java à ses débuts avant d'être rachetée par Oracle). *Le respect de ces spécifications permet d'utiliser les EJB de façon indépendante du serveur d'applications J2EE dans lequel ils s'exécutent, du moment que le code de mise en œuvre n'utilise pas d'extensions proposées par un serveur d'applications particulier.* (extrait : <https://www.jmdoudoux.fr/java/dej/chap-ejb.htm> )
- Aujourd'hui, si des EJB ne sont pas utilisées la tendance est plutôt d'essayer d'écrire son code dans un **langage métier plus fluent** ...

**Bonne pratique (qualité de code)** :

Pour savoir si vous devez écrire vos **getteurs** en respectant la norme EJB (**getX**) ou de manière « fluente » (**x**), renseignez-vous sur les **standards de codage** de votre entreprise et du projet sur lequel vous travaillez .

**La qualité de code passe par la cohérence de code sur un projet !**

## → Reverse-engineering (rétro-conception)

Pour finir, vous pouvez vérifier si votre ***implémentation est bien cohérente avec la conception*** demandée (diagramme de classes de la première page de cet énoncé).

Pour cela, procédez à la **rétro-conception** (reverse engineering) dans un fichier **umc1d** en utilisant, comme vu dans le précédent TP, le plug-in **UML Lab Modeling**.  
(New → Other... → UML Model and Class Diagram)

Vous pouvez maintenant comparer le diagramme de classes issu de la phase de conception (celui de la page 1) et de le diagramme de classes à la fin de la phase d'implémentation....

Que constatez-vous ?

Et oui, la conception n'est jamais complète du premier coup, c'est pour cela qu'il faut travailler à plusieurs reprises sur une conception (travail itératif et incrémental) et bien souvent l'implémentation nécessite encore de faire apparaître de nouvelles méthodes 😊

**Bonus :** Pour les plus rapides (et ceux qui le souhaitent) :

Créez un projet **pokemon** et implémentez les exemples du cours disponible sur :

<https://www.onete.net/teaching.html>