Personnaliser son environnement de développement
Jouer avec les pionniers de l'informatique moderne
Et faire un petit tour dans le Far-West

Exercice 1 : Personnaliser son environnement de développement

Cela fait maintenant plusieurs jours que vous cherchez une *solution* pour faire de la **rétro-conception** (***reverse engineering***) d'un projet java directement dans votre IDE Eclipse (c.-à-d. transformer automatiquement le code du projet en diagramme de classe en repassant).
Vous êtes donc à la recherche d'un plug-in plus simple qui génère seulement le diagramme de classes, sans risquer d'avoir des impacts sur votre code 😉.
Et pour l'instant, vos recherches n'ont pas été très fructueuses :

→ **Object Aid UML Explorer** (https://www.objectaid.com) était le plug-in le plus facile à utiliser pour la rétro-conception sous Eclipse au vu de votre usage, malheureusement son site ne répond plus ☹
→ **UML Lab Modeling (https://www.uml-lab.com)** est un plug-in payant qui, de plus, va au-delà de l'usage que vous souhaitez en faire puis des modifications sur le diagramme peuvent engendrer des modifications dans le code (*round-trip enginneering)* ☹.
Tips : Avec votre statut étudiant, vous pouvez très souvent bénéficier d'une licence dite « academic » ou « education » en vous inscrivant avec l'adresse mail de l'université et/ou un scan de votre carte étudiant , et c'est le cas pour UML Lab Modeling , mais en tant que professionnel c'est différent il faut s'acquitter de la licence.

A la machine à café, vous discutez donc de ce *problème* avec vos collègues qui vous conseille d'installer et de tester **plant UML** qui leur semble très simple d'utilisation, facile à prendre en main et pourrait donc correspondre à vos besoins.

❑ Installer un nouveau plug-in dans l'IDE Eclipse via `Install New Software`

Pour installer un nouveau plug-in, rien de mieux que de chercher sur le site de ce dernier, s'il existe une rubrique qui explique l'installation et c'est le cas pour le plug-in Eclipse de plant UML !

Rendez-vous donc à l'adresse suivante : https://plantuml.com/fr/eclipse

1. **Installation :** Commencez par suivre les instructions de la rubrique (au milieu de la page) : **How to install it ?** *Veillez bien à avoir redémarrer Eclipse pour que l'installation soit bien finalisée …*

   Remarque : il y a deux manières d'installer un plug-in sous Eclipse :
   - soit à partir du menu **`Help → Install New Software...`**
   - soit à partir du menu **`Help → Eclipse Market`**

   D'après la documentation fourni dans la rubrique **How to install it ?** , le plug-in doit ici être installé via le menu **`Help → Install New Software...`** en indiquant l'adresse mentionnée dans la documentation. Procédez à l'installation du plug-in de cette manière.

   Quoi qu'il en soit, retenez bien que si vous ajoutez un plug-in à votre IDE,
   vous devez impérativement redémarrer votre IDE après l'installation du plug-in pour que ce dernier puisse être utilisé !!!

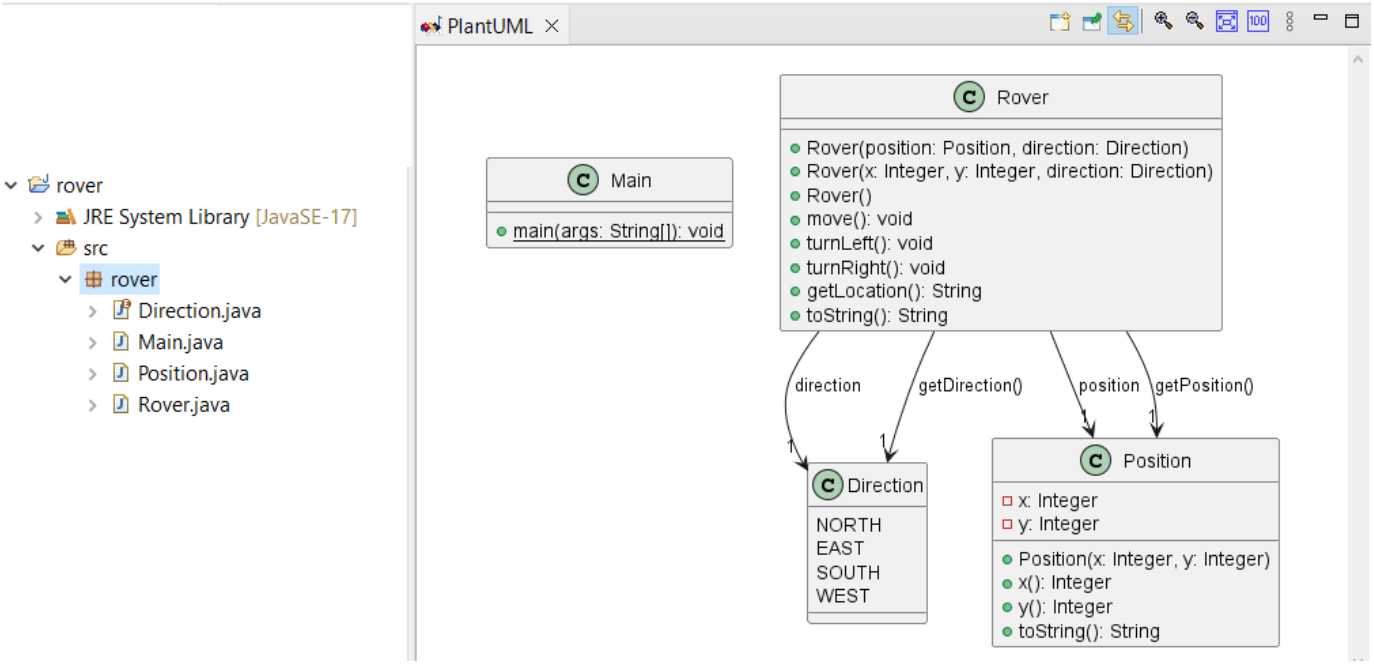
2. **Documentation :** Suivre la rubrique **How to use it ?** pour comprendre comment utiliser **Plant UML**. Notez bien que la rétroconception va s'afficher dans une ***Vue***.
   Il faut donc passer par le menu **`Window → Show view → Other…`**
   pour trouver et double-cliquer sur **la vue `Plant UML`**

3. **Utilisation : Première rétroconception avec `PlantUML` :**

   Pour tester la rétro-conception avec `plantUML`, nous allons utiliser un projet existant.

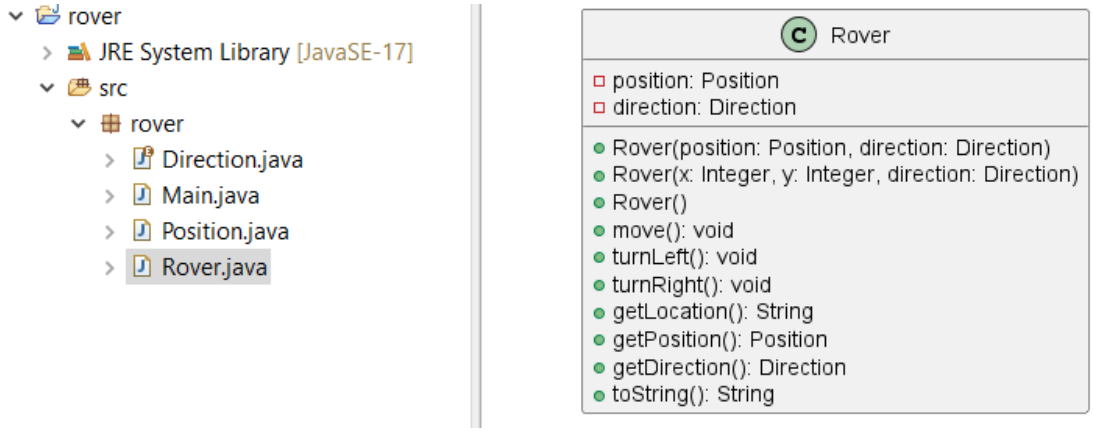
   ✓ Ouvrir **le projet `rover`** du TP précédent
   ✓ Ouvrir une vue **`PlantUML`** comme indiquée dans la documentation.
   (**`Window → Show view → Other…`** puis double-clic sur **la vue `Plant UML`**)
   ✓ Depuis la vue **Package Explorer**, sélectionnez le **code à rétro-concevoir** :

   a. **<u>Soit tout le code du projet</u>**
   Dans la vue Package Explorer placez-vous sur la package **`rover`** et la vue **`PlantUML`** est automatiquement mis à jour avec les classes contenues dans le package.

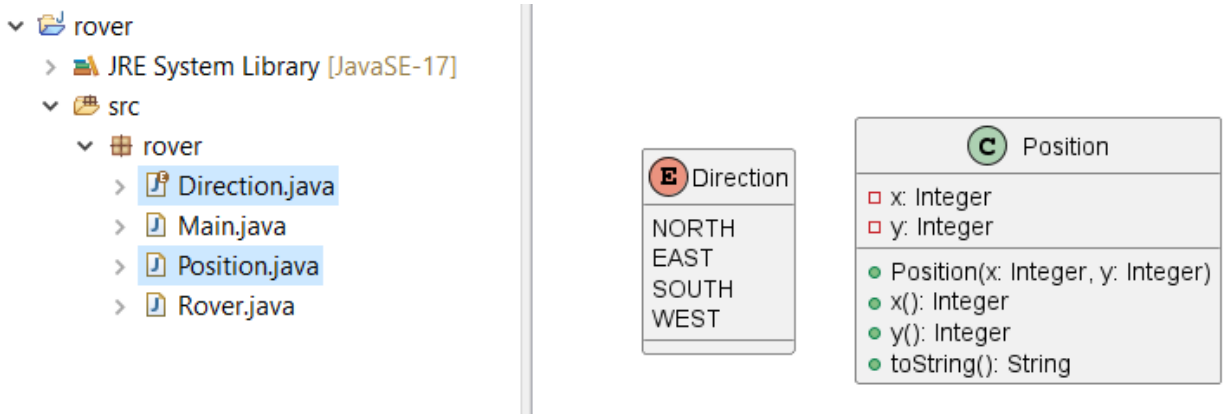


b. Soit un extrait du code du projet (par une sélection de classe(s))

➔ Dans la vue **Package Explorer** si vous cliquez sur **une seule classe**, cette seule classe s'affichera dans la vue **Plant UML**

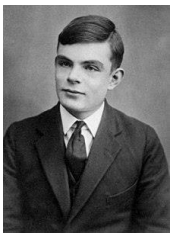


➔ Dans la vue **Package Explorer** si vous **sélectionnez plusieurs classes** (celles que vous souhaitez) **en laissant appuyée la touche CTRL** entre chaque sélection, seule les classes sélectionnées s'afficheront dans la vue : essayez d'affichez simplement la classe **`Position`** et l'énumération **`Direction`**



Remarque : Le plug-in de **rétro-conception** est un plus de PlantUML.
L'objectif initial de **PlantUML** est de permettre de *dessiner* des diagrammes UML, à l'aide d'un langage simple à lire pour un utilisateur.
Tout est indiqué sur le site de référence de plantUML : https://plantuml.com/fr/
Il existe même une page Wikipédia où vous pouvez retrouver tous les outils logiciels qui peuvent utiliser sur **`PlantUML`** : https://en.wikipedia.org/wiki/PlantUML

Exercice 2 : Retour vers les pionniers de l'informatique moderne

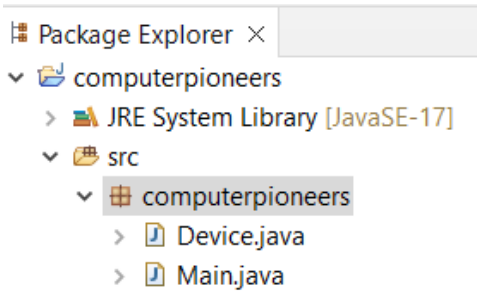


Depuis la vue **Package Explorer**, ouvrez le projet **`computerpioneers`** en double cliquant dessus, puis fermez tous les autres projets (**clic droit**→ ***Close Unrelated Project***).

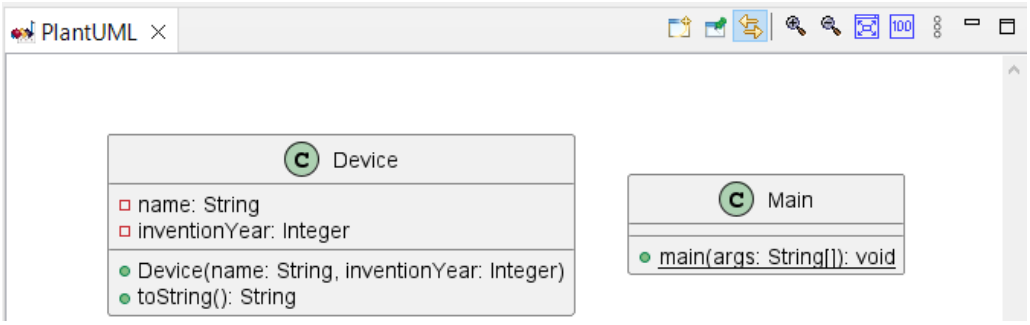
Pour vous refamiliarisez rapidement avec ce projet, vous pouvez :

→ dans un premier temps, **déplier la vue Package Explorer** pour faire un **rapide état des lieux** et visualiser l'architecture du projet et constater/vous rappeler que vous aviez précédemment travaillé sur deux classes :

- Une classe métier **`Device`**
- Une classe contenant des jeux d'essais pour tester la bonne implémentation de votre classe métier 😉



→ puis vous **(re)plonger rapidement dans la conception de ce projet** et visualiser en un clin d'œil l'ampleur du code écrit en procédant à une petite rétro-conception sur tout le projet à l'aide de votre nouveau meilleur ami Plant UML 😉



→ Demandez-vous ensuite si le **code disponible est fonctionnel ?**
Pour cela, rien de mieux que de **lancer les jeux de tests**, en l'occurrence ici ce sera la méthode **`main`** de la classe **`Main`** 😉
Que visualisez-vous sur la console ? Cela vous semble-t-il correct au vu des jeux d'essais ?

→ Si vous avez fini la première séance de TP, vous devriez voir apparaître sur la console :

```
Babbage Analytical Machine was invented in 1837.
Turing Engine was invented in 1936.
```

⇨ Si tel n'est pas le cas, reprenez la fin de l'énoncé de la partie précédente (premier TP) : peut-être n'aviez-vous pas encore implémenté la méthode `toString` 😉

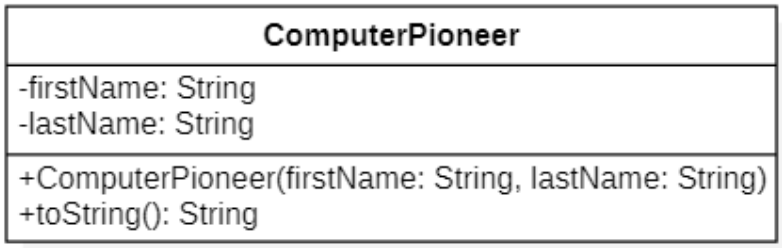
⇨ Il ne vous reste plus qu'à (re)mettre les mains/yeux dans le code.
Ouvrez et parcourez les fichiers **`Device.java`** et **`Main.java`**

La première séance de TP nous a permis de nous focaliser sur la classe **`Device`**.

Nous allons maintenant nous focaliser sur le classe **`ComputerPioneer`**.

*Pour continuer cet énoncé, vous veillerez bien évidement **avoir sous la main le TD « Les débuts de l'informatique moderne : la classe ! (suite) »** c'est l'implémentation de ce TD qui vous est demandé dans cet exercice de manière guidée et en vous montrant que l'IDE peut vous être d'une aide précieuse* 😉

❑ Focus sur la classe `ComputerPioneer` :



1. Implémentation :
Implémentez maintenant la classe `ComputerPioneer` qui respecte le *design* ci-contre
Pour l'implémentation de `toString` voir ci-dessous ce qui est attendu comme affichage dans le test…

2. Test

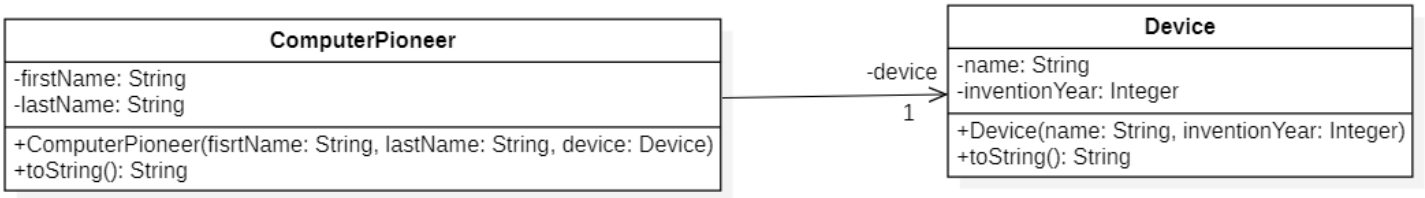
En vous inspirant de ce qui a été fait pour la classe `Device`, implémentez dans la méthode `main`, deux nouveaux **jeux d'essais relatifs à la classe `ComputerPioneer`** pour un affichage attendu en mode console :

```
Ada Lovelace is a pioneer in Computer Science.
Alan Turing is a pioneer in Computer Science.
```

❑ Relation entre les classes métier `ComputerPioneer` et `Device` sachant que *un pionnier travaille sur un et un seul appareil*

3. **Implémentation**

Complétez votre code de manière à ce qu'il respecte le *design* ci-dessous



4. Test

Réécrire entièrement la méthode **`main`** de manière à obtenir les deux jeux d'essais suivants, suite à l'instanciation et à la demande d'affichage des deux *objets* de type **`ComputerPioneer`** que vous nommerez **`adaLovelace`** et **`alanTuring`**. Les *objets* de type **`Device`** devront être instanciés avant d'être passés en paramètre. Vous nommerez ces objets `babbageMachine` et `turingEngine` comme précédemment.

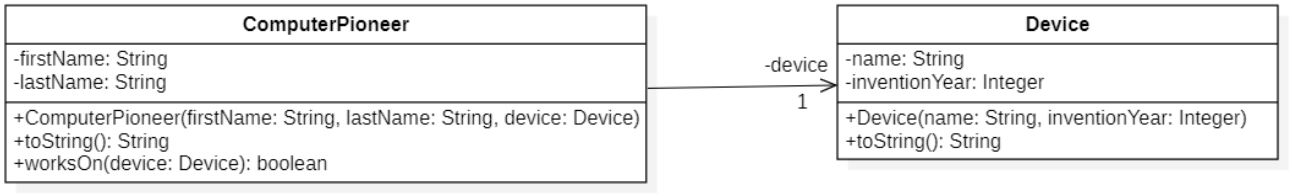
```
The Babbage Analytical Machine was invented in 1837. Ada Lovelace is a pioneer in Computer Science who worked on it.

The Turing Engine was invented in 1936. Alan Turing is a pioneer in Computer Science who worked on it.
```

❑ Ajouter du comportement à la classe `ComputerPioneer` : *Être capable de dire si un pionnier travaille sur un device donné*

5. Implémentation et Test

Complétez votre code de manière à ce qu'il respecte le *design* ci-dessous
c.-à-d. que vous devez implémenter un **nouveau service** (**méthode**) **`worksOn`** dans la classe `ComputerPionner` qui prendra un `Device` en paramètre et dont le comportement consistera à dire si le *pionneer* travaille sur un certain *device*.



Bien sûr l'implémentation de worksOn dépend de ce que vous entendez par :
le *pioneer* travaille sur un certain *device* en question.

Dans un premier temps, complétez votre méthode `main` avec les deux jeux d'essais suivants :

```
public static void main(String[] args) {

      //… code déjà écrit précédemment

      System.out.println("Test case 3 ");
      System.out.println("--------------------");
      System.out.println(adaLovelace.worksOn(babbageMachine));
      System.out.println(adaLovelace.worksOn(turingEngine));
      System.out.println(alanTuring.worksOn(babbageMachine));
      System.out.println(alanTuring.worksOn(turingEngine));
      System.out.println("--------------------");

      System.out.println("Test case 4 ");
      System.out.println("--------------------");
      Device babbage = new Device ("Babbage Analytical Machine",1837);
      Device turing = new Device ("Turing Engine",1936);
      System.out.println(adaLovelace.worksOn(babbage));
      System.out.println(adaLovelace.worksOn(turing));
      System.out.println(alanTuring.worksOn(babbage));
      System.out.println(alanTuring.worksOn(turing));
      System.out.println("--------------------");

}
```

8.a Implémentez ensuite la méthode **worksOn** dans la classe **ComputerPioneer** pour faire sorte d'obtenir l'affichage suivant à la console :

```
--------------------
Test case 3
--------------------
true
false
false
true
--------------------
Test case 4
--------------------
false
false
false
false
--------------------
```

⇨ Comment avez-vous implémenté la méthode **worksOn** ?

……..
……..

Autrement dit la méthode **worksOn** compare la …… de deux objets de type Device (celui utilisé lors de l'instanciation de l'objet ComputerPioneer et celui passé en paramètre de la méthode **worksOn**)

8.b Implémentez ensuite la méthode **worksOn** dans la classe **ComputerPioneer** pour faire sorte d'obtenir l'affichage suivant à la console :

```
--------------------
Test case 3
--------------------
true
false
false
true
--------------------
Test case 4
--------------------
true
false
false
true
--------------------
```

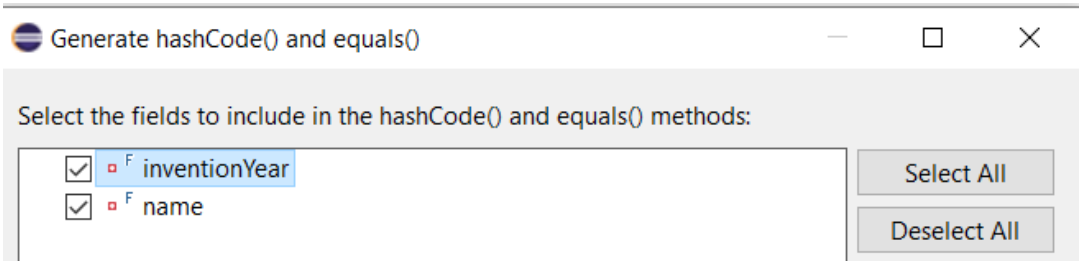
⇨ Comment avez-vous implémenté la méthode **worksOn** ?

……..
……..

Autrement dit la méthode **worksOn** compare …… de deux objets device (celui utilisé lors de l'instanciation de l'objet ComputerPioneer et celui passé en paramètre de la méthode **worksOn**)

📣 Bonne pratique !!! En principe, on n'écrit pas directement le code des méthodes `equals` (et `hashCode`) d'une classe, ce code est généré 😉.

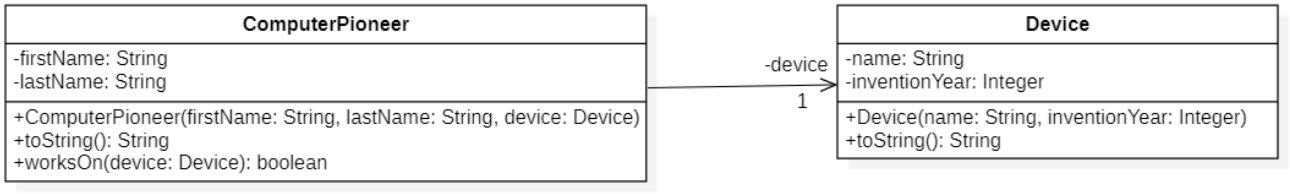
Pour faire générer le code des méthodes **`equals`** (et **`hashCode`**) par l'IDE Eclipse, placez-vous dans la classe `Device`, puis allez dans le menu **`Source → Generate hasCode et equals`**, vérifiez que tous les attributs de la classe soient bien cochés, puis cliquez sur **`Generate`**.



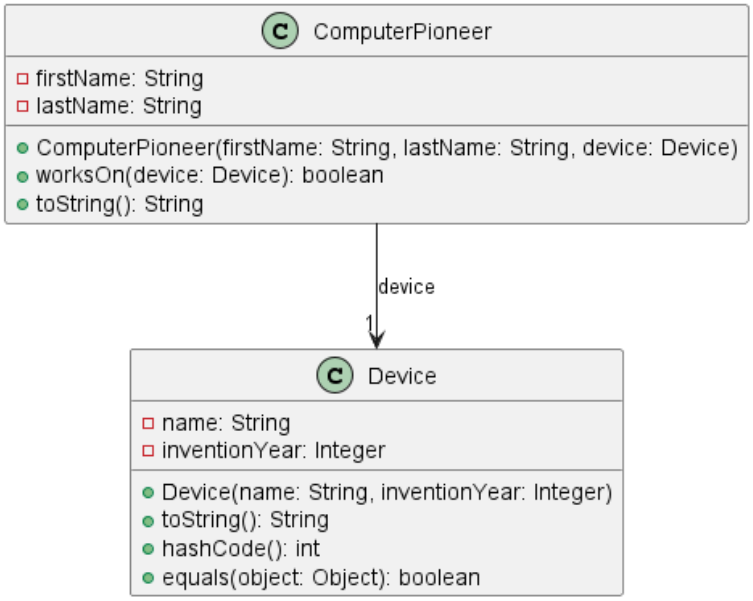
Le code des méthodes **`equals`** et **`hasCode`** apparaît dans votre classe, n'oubliez pas de sauvegarder !

❑ Et une dernière petite rétro-conception …

Pour finir, vous pouvez vérifier si votre ***implémentation est bien cohérente avec la conception*** demandée (diagramme de classes issu de la phase de conception est similaire au diagramme ci-dessous) :



→ Pour cela, procédez à la **rétro-conception** (reverse engineering) l'aide de **Plant UML** en sélectionnant uniquement les classes ComputerPioneer et Device (à l'aide de CTRL+clic), vous devriez obtenir un diagramme similaire au diagramme ci-dessous



→ Vous pouvez ensuite comparer le diagramme de classes issu de la phase de conception et de le diagramme de classes généré à la fin de la phase d'implémentation….

Que constatez-vous ?

A retenir :

⇨ Les méthodes **`equals/hashcode`** (et parfois **`toString`**) n'apparaissent habituellement pas dans les diagrammes de classes de la phase de conception.
En effet, ces méthodes sont propres au langage Java et sont habituellement générées automatiquement par l'IDE… mais rappelons que ces méthodes sont indispensable dans les classes lors de l'implémentation 😉

⇨ la conception n'est en principe 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 😉, donc bien souvent ne soyez pas étonnés que les diagrammes que vous générer soient plus riche que les diagrammes que vous aviez imaginés lors de la phase de conception…

A retenir ….

Une bonne pratique de conception pour le diagramme de classes consiste
à ne **représenter que les opérations (services)**
qui apportent un comportement métier pertinent
(comme `worksOn` par exemple)

c.-à-d. qu'habituellement, pour ne pas alourdir la lecture des diagrammes de classes,
on ne représente pas sur le diagramme de classes :
ni les constructeurs, ni les getteurs/setteurs,
ni méthodes `toString`, `equals` et `hascode` :
ces méthodes pouvant être **générées automatiquement via l'IDE** :
par exemple sous Eclipse avec le Menu **`Source`** qui propose
une série de sous-menus **`Generate`…**

Une bonne pratique d'implémentation Java consiste pour chaque nouvelle classe métier
à **redéfinir pour les 3 méthodes de la classe `Object` :**
`toString, hascode, equals`
(pour le moment via une génération automatique de l'IDE,
ou comme on le verra plus tard via lombok
ou en utilisant pertinemment la nouvelle notion *`record`* disponible depuis la version Java 14)

Pour implémenter une égalité en Java,
il est indispensable de bien comprendre la **différence entre** :

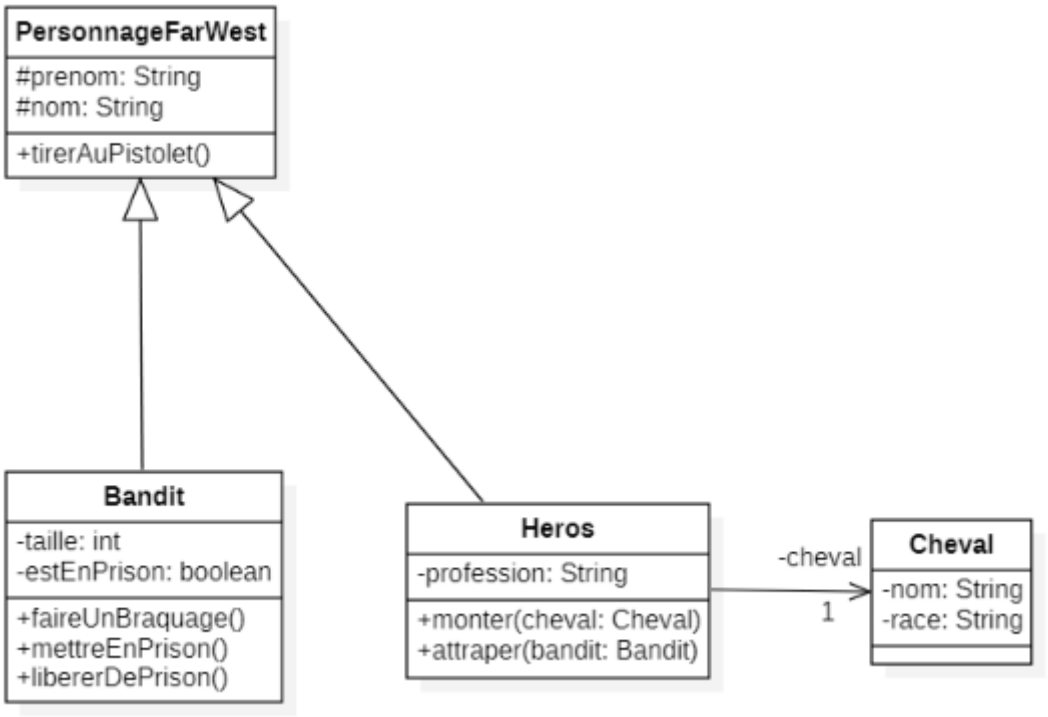
`==`	Comparaison des **références** de deux objets
`equals`	Comparaison du **contenu** de deux objets (**état**)

Exercice 3 : Un petit tour au Far-West

Pour cet exercice, vous veillerez à avoir sous la main le TD « En route vers l'héritage… »

☑ Phase d'analyse et de conception :

Vous venez de rejoindre l'équipe de développement du projet Far-West.
Les besoins du client ont déjà été analysés et la phase de conception, qui s'en est suivie, a conduit au diagramme de classes suivant :

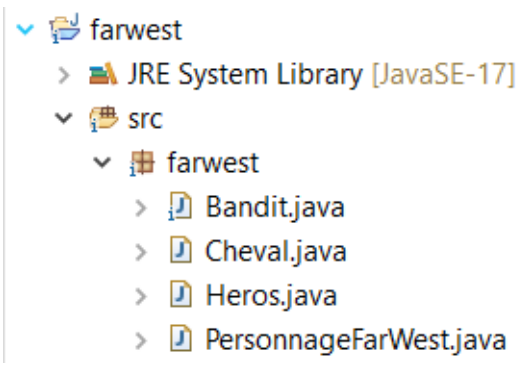


Votre équipe vous charge de l'implémentation Java de cette conception 😉

❑ Phase d'implémentation

⇨ Commencez par créer, sous Eclipse un **nouveau projet java** (**`File→ New→ Java project`**), **toujours sans module**, que vous appellerez **`farwest.`**

Une fois le projet créé, avant de continuer, pour plus de confort,
assurez-vous que tous les autres projets de votre workspace sont bien fermés.
(dans la vue Package Explorer, placez-vous sur le nouveau projet,
puis clic droit puis **`Close Unrelated Project`**)



⇨ Implémentez ensuite dans le ***`src`*** de ce projet, les différentes classes composant le diagramme de classes précédent.
→ ***Pour chaque classe***, n'oubliez pas d'implémenter un **constructeur primaire** (constructeur prenant en paramètres tous les attributs de la classe). Pour les autres méthodes faites apparaître des / / TODO pour le moment.
→ ***Pour chaque attribut***, posez-vous la question de savoir si sa *valeur de cet attribut évolue* au cours du temps ou s'il s'agit d'un *attribut immuable* afin d'utiliser à bon escient le mot clé **`final`.**

⇨ Pour vérifier que vous avez bien respecter l'architecture du diagramme de classes qui vous a été transmis, procédez à une petite **rétro-conception** (reverse engineering) l'aide de **Plant UML** soit en sélectionnant directement le package `farwest`, soit en sélectionnant une à une toutes les classes à l'aide de CTRL+clic.
Veillez également à bien vérifier que l**e principe d'encapsulation** est bien respecté…

❑ Phase de tests

⇨ Pour vérifier et valider la *bonne* implémentation du projet, vous allez créer une classe **`Main`** dans laquelle vous implémenterez le jeu d'essais suivant :

```
public static void main(String[] args) {
   Cheval jollyJumper= new Cheval ("Jolly Jumper","appaloosa");
   System.out.println(jollyJumper.decrire());

   Heros luckyLuke = new Heros("Lucky","Luke","cow-boy",jollyJumper);
   System.out.println(luckyLuke.decrire());

   Bandit joeDalton = new Bandit("Joe","Dalton",150);
   System.out.println(joeDalton.decrire());
   joeDalton.mettreEnPrison();
   System.out.println(joeDalton.decrire());
   joeDalton.libererDePrison();
   System.out.println(joeDalton.decrire());
}
```

Afin d'obtenir sur la console, l'affichage suivant :

```
Jolly Jumper de race appaloosa
Lucky Luke! Je suis cow-boy et mon cheval est Jolly Jumper de race appaloosa
Joe Dalton! Je mesure 150 cm et je suis Libre
Joe Dalton! Je mesure 150 cm et je suis en Prison
Joe Dalton! Je mesure 150 cm et je suis Libre
```

📣 *Rappel :* pour écrire plus rapidement `System.out.println()`,
Eclipse vous propose l'auto-complétion via le raccourci suivant : `syso` suivi de `CTRL+Espace`

📣 Pour pouvoir jouer ce jeu d'essai, vous devez bien sûr, à la place des `//TODO`, proposez une implémentation adéquate pour les méthodes `decrire` et `mettreEnPrison` et `libererDePrison`.

⇨ Et pour terminer, vérifiez que vous avez écrit du **code qualité** qui minimise la duplication de code :
→ les constructeurs d'objets du Far-West c-a-d des classes **`Bandit`** et **`Heros`** doivent directement **faire appel au constructeur de la classe mère `PersonnageFarWest`.**
Si cette question n'a pas eu le temps d'être traitée durant la séance de TD, c'est le moment d'en discuter avec votre enseignant de TP 😉
→ les méthodes **`décrire`** des classes **`Bandit`** et **`Heros`** doivent **réutiliser le code de la méthode `décrire` de la classe mère `PersonnageFarWest`.**

Travail à faire pour la séance prochaine :

❑ Les exercices 2 et 3 doivent être terminés

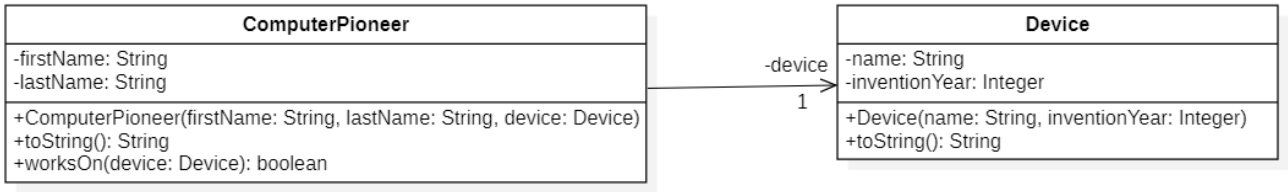
Pour prolonger cette séance, si le cœur vous en dit ….
Prise en main d'un AGL

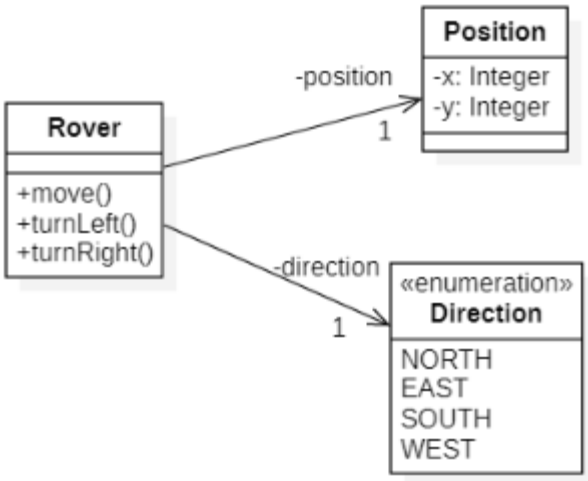
Vous en avez assez de dessiner vos diagrammes de classes à la main et vous préféreriez qu'il soit numérisé ? Nous vous avons sélectionné les outils UML suivants dont l'utilisation est assez intuitive :

- un **AGL** (**A**telier **G**énie **L**ogiciel) graphique comme **StarUML** : **https://staruml.io/** à **installer sur votre ordinateur.** Si vous utilisez les ordinateurs de l'IUT StarUML est déjà installé 😉 Familiarisez-vous avec cet outil **en consultant sa documentation** (menu Docs en haut à droite du site https://staruml.io/ qui vous conduira sur https://docs.staruml.io/)
- Un outil permettant de « *coder* » des diagrammes UML à l'aide d'un langage simple et intuitif comme **PlantUML** (**https://plantuml.com/fr**).
- **un outil en ligne collaboratif** comme **GenMyModel** (**https://app.genmymodel.com**).

Pour vous familiariser avec ces outils et choisir celui qui vous convient le mieux, nous vous proposons de reproduire les diagrammes suivants.
Nous vous conseillons de commencer par tester **StarUML** qui est l'AGL actuellement utilisé pour produite les diagrammes des énoncés de cette ressource 😉

- les **diagrammes de classes des projet `computerpioneers`** et **`rover`**





Remarque : Si le cœur vous en dit, une liste non exhaustive d'outils supplémentaires autour de la modélisation UML est proposée sur https://fr.wikipedia.org/wiki/Comparaison_des_logiciels_d%27UML