

TP R2.01 & R2.03:

Introduction aux exceptions

Apprendre à versionner un projet (Maven sous Eclipse)

Partie 1 : Introduction aux exceptions

1. Quelques bases sur les exceptions :

✓ Quand on développe une application, deux types de problèmes peuvent apparaître :

❑ **les erreurs** apparaissant lors de la phase d'**implémentation détectables à la compilation (et indiquées par l'IDE)** : Le programme ne peut pas être exécuté, tant que ces erreurs ne sont pas traitées.

❑ **les erreurs** apparaissant à l'**exécution**

Exemples : lecture / écriture en dehors des limites d'un tableau, écriture sur un disque plein, division par zéro.

⇒ A ce moment une **exception** est générée (on dit aussi qu'elle est « lancée ») et si l'exception n'est pas **capturée**, l'application s'arrête en affichant une **pile d'erreurs**.

✓ Une **exception** est donc un « signal » :

- qui **indique** que **quelque chose d'exceptionnel s'est produit (à l'exécution)**
- et qui **interrompt** le flot normal d'exécution du programme

✓ **Terminologie** :

→ **Lancer (lever: throw)** une exception consiste à **signaler quelque chose d'exceptionnel**

→ **Attraper (capturer: catch)** une exception permet **d'exécuter les actions nécessaires pour la traiter**

✓ **Retour sur un exemple d'Exception** :

La semaine dernière dans le projet **calculatrice**, vous avez écrit une classe **Sandbox** pour « jouer » avec les exceptions :

→ Le programme ci-contre compile sans problème, par contre lorsque vous l'exécutez, le programme s'interrompt

brusquement en affichant, en rouge, dans la console la pile d'erreurs (*stack traces*) suivante :

```
1 package calculatrice;
2
3 public class Sandbox {
4
5     public static void main(String[] args) {
6
7         Calculatrice calculatrice = new Calculatrice();
8         int resultat = calculatrice.diviser(42, 0);
9         System.out.println("Le résultat de la division : " + resultat);
10    }
11 }
```

Exception in thread "main" [java.lang.ArithmeticException: / by zero](#)
at calculatrice.Calculatrice.diviser([Calculatrice.java:18](#))
at calculatrice.Sandbox.main([Sandbox.java:8](#))

Cette *trace* indiquera toujours :

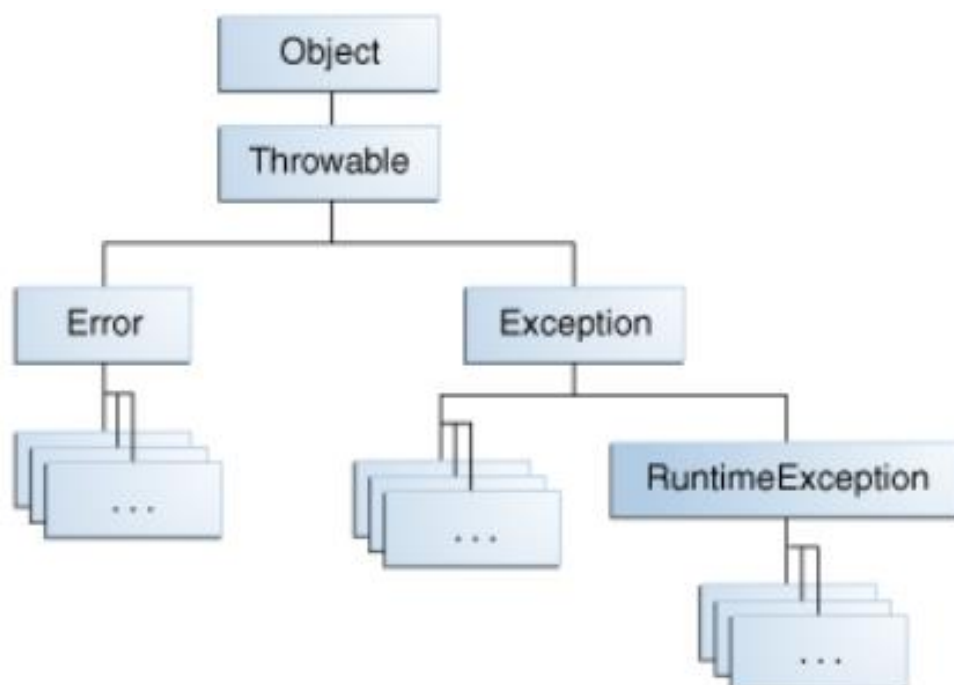
- **sur la première ligne**, le **nom de l'exception levée** : [java.lang.ArithmeticException](#)
- **sur la dernière ligne**, l'instruction jusqu'où l'exception est remontée sans être attrapée et donc cette instruction représente la dernière instruction exécutée, celle qui a fait « planter » l'application.
- ensuite, la trace pourra comporter un certain nombre de lignes intermédiaires qui permettront de remonter ligne par ligne à partir de l'avant-dernière jusqu'à la **l'instruction qui a lancé l'exception** (ici c'est dû à la division par zéro dans la classe Calculatrice c-a-d la ligne d'instruction qui contient /)

→ Pour éviter que le programme s'arrête intempestivement, il faut **capturer/attraper l'exception** à l'aide un **try...catch** pour la traiter. Une fois capturée, on peut éventuellement, si on le souhaite, afficher le message « portée » par l'exception :

```
try {  
    resultat = calculatrice.diviser(42, 0);  
} catch (ArithmeticException e) {  
    System.out.println("La division par zéro est impossible");  
    System.out.println("L'exception capturée est :b"+e.getMessage());  
}
```

✓ **En Java les exceptions sont des objets :**

En Java, les exceptions sont des objets qui héritent de Throwable.



(Hiérarchie extraite de : <https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html>)

Il y a deux grandes familles de **Throwable** :

❑ les **Error** qui sont de **graves erreurs fatales à l'exécution de l'application** c-a-d impossible à récupérer (attraper & traiter). Ce sont les erreurs internes ou manque de ressources avec par exemple des classes comme : `NoSuchMethodError`, `StackOverflowError`, `OutOfMemoryError`, `IllegalAccessError`...

❑ les **Exception** qui sont **des erreurs récupérables** c-a-d des erreurs pouvant être attrapées et traitées afin de poursuivre proprement l'exécution de l'application.

Ce type d'Exception récupérable se sous-divise encore en 2 familles :

❑ la famille des exceptions non contrôlées par le compilateur appelées **Unchecked Exception** est dont la classe mère est **RuntimeException**. Ces exceptions sont souvent déclenchées par une erreur de programmation : l'`ArithmeticException` en est un bel exemple : le code a compilé sans problème (*unchecked*), mais l'exécution a posé un problème 😊
D'autres classes comme `NumberFormatException`, `IndexOutOfBoundsException`, ... font parties des *Unchecked Exception*.

❑ la famille des exceptions contrôlées par le compilateur appelées *Checked Exception* qui regroupent toutes les autres exceptions (qui ont **juste Exception** comme classe mère et ne font pas partie de la branche de `RuntimeException`)

Le compilateur provoquera une erreur de compilation (*checked*) tant que ce type d'exception n'aura pas été traitée c.-à-d. soit attrapée (**catch**) ou propagée (**throws**).

→ le JDK définit un certain nombre de *Checked Exception*, notamment dans le cas d'entrée/sortie (*Input/Output : IO*)

→ ... mais il est également possible de concevoir ses propres exceptions en implémentant **une classe qui hérite de Exception** ! Toutes les exceptions que vous créerez seront des *Checked Exception* et devront être attrapées à moment donné 😊

Dans la javadoc : <https://docs.oracle.com/en/java/index.html>

- ➔ recherchez la documentation de la classe **Exception** et la classe **Throwable**
 - Pourquoi avons-nous pu utiliser la classe `getMessage`, de quelle classe cette méthode provient-elle ?
- ➔ recherchez la documentation de la classe **ArithmeticException** et complétez le document suivant :

Class `ArithmeticException`

```
java.lang. ....
  java.lang. ....
    java.lang. ....
      java.lang. ....
        java.lang.ArithmeticException
```

La documentation précédente justifie également le fait que dans le TP précédent, nous avons pu écrire indifféremment :

```
try {
    resultat = calculatrice.diviser(42, 0);
} catch (ArithmeticException e) {
    System.out.println("La division par zéro est impossible");
    System.out.println("L'exception capturée est : "+e.getMessage());
}
```

ou

```
try {
    resultat = calculatrice.diviser(42, 0);
} catch (Exception e) {
    System.out.println("La division par zéro est impossible");
    System.out.println("L'exception capturée est : "+e.getMessage());
}
```

Faites en sorte que le **main** de la classe **SandBox** de votre projet **calculatrice** contienne désormais ces deux **try ... catch** 😊

2. Implémenter sa première exception

RuntimeException est une **UncheckedException** qui laisse le code compiler (pas d'erreur de compilation) et qui, si cette exception n'est capturée fait « planter » le programme à l'exécution.

Si ce comportement de base ne vous convient pas et si vous préférez que **le problème de la division par zéro soit explicite**, c-a-d signaler *explicitement* que quelque chose d'exceptionnel se produit dans votre code et surtout traiter ce problème dans votre code à l'endroit que vous jugez judicieux pour éviter que votre programme s'arrête brusquement, vous pouvez mettre en place **votre propre exception**, qu'on appelle aussi « **exception métier** » (puisqu'elle permet de rendre **plus sûr le code métier**).

Et comme dit dans le cadre de la page précédente, ce type d'exception doit **hériter de **Exception**** ce qui en fait une **Checked Exception** (le code ne pourra pas être compilé sans avoir au préalable traité cette exception).

- a. Implémenter la classe **DivisionParZeroException** de la manière suivante :

```
@SuppressWarnings("serial")
public class DivisionParZeroException extends Exception {

    public DivisionParZeroException(String message) {
        super(message);
    }
}
```

... et apprenez par cœur ce petit bout de code car vous utiliserez toujours ce modèle pour implémenter vos propres **exceptions métiers de type *Checked Exception***, au nom de la classe près...

Pour la qualité de code, pensez bien à nommer vos propres exceptions métiers avec des noms explicites qui reflètent l'intention de l'exception à déclencher : ici l'exception sera uniquement utilisée pour signaler des divisions par zéro 😊

b. Cohérence du code : util

Les exceptions sont des **classes utilitaires** qui permettent d'améliorer la qualité et la sûreté de votre code. En principe ces classes sont implémentées dans le package **util**.

Réorganisez votre projet **calculatrice**, de manière à avoir :

- La classe **Calculatrice** dans le package **model**
- La classe **DivisionParZeroException** dans le package **util**

```
▼ calculatrice.model
  > Calculatrice.java
▼ calculatrice.util
  > DivisionParZeroException.java
```

3. Lancer sa première exception (**throw** & **throws**)

a. Lancer l'exception (**throw**)

En vous inspirant de la rubrique **5.2 Throwing Unchecked Exception** sur le site de baeldung (<https://www.baeldung.com/java-exceptions>), faites en sorte en à l'aide de **new throw** de

lancer une nouvelle **DivisionZeroException** lorsque le diviseur est égal à 0

Cette instruction devra être la première de la méthode **diviser** de la classe **Calculatrice**

Et vous utiliserez pour l'exception le message suivant :

Division impossible car le diviseur est égal à zéro !

Une erreur de compilation apparaît une fois l'instruction du déclenchement de l'exception écrite.

- Quelle est-elle ?
- Que vous propose le compilateur pour la corriger ?

b. Déclarer qu'une *Checked exception* est susceptible de se lancer dans la signature de la méthode (**throws**)

Faites en sorte que votre code compile en signalant la présence de votre **Checked Exception** dans la méthode **diviser** comme vous l'indique le compilateur en faisant en sorte de faire apparaître dans la signature de la méthode un **throws** pour continuer à propager l'exception.

4. Attraper sa première exception métier « à tout prix » sous peine de ne pas pouvoir compiler

Que se passe-t-il maintenant dans la classe **SandBox** ?

Des erreurs de compilation apparaissent lorsque l'exception **DivisionParZeroException** n'est pas gérée ...

Dans la classe **SandBox**, constatez que même un appel à **diviser** qui est censé bien se passer engendre une erreur de compilation...

```
int dividende = 42;
int diviseur = 2;
int resultat = calculatrice.diviser(dividende, diviseur);
System.out.println("Le résultat de la division de " + dividende + " par "
                    + diviseur + " est : " + resultat);
```

Et oui, c'est le principe même d'une *Checked Exception*, cette exception doit **absolument être gérée (handled)** quelque part dans le code pour que ce dernier puisse compiler !

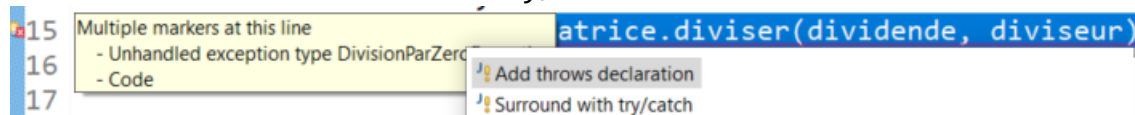
Il y a deux manières de « gérer » (handle) une exception :

- **Cas n°1 : Gérer l'exception en la propageant** c-a-d en déléguant son traitement au niveau au-dessus. Pour réaliser cela, la méthode où apparaît l'erreur de compilation liée à l'éventuelle présence de l'exception doit déclarer qu'une (*Checked*) Exception est susceptible de passer par là en indiquant un **throws** suivi du nom de l'exception dans sa signature.
- **Cas n°2 : Gérer l'exception en la capturant et la traitant** en l'englobant (*surround*) dans un bloc **try...catch**

a. Illustration du cas gérer l'exception en la propageant (**throws** dans la signature)

Dans la classe **SandBox**, revenez sur la première erreur de compilation donnée par le code précédent. Cliquez sur la croix rouge, l'IDE :

- vous indique l'erreur de compilation liée à l'exception : *Unhandled exception type DivisionParZero*
- vous propose de gérer cela, en choisissant
 - le cas 1 : Add throws declaration
 - ou le cas 2 : Surround with try/catch



Cliquez sur le **Add throws declaration** de manière à ce que votre **main** ait maintenant la signature suivante :

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

Sauvez votre classe **SandBox** : cette solution vous a permis de supprimer toutes les erreurs de compilation. Votre code compile bien, mais que se passe-t-il maintenant à l'exécution ?

Exécutez ce code et constatez que ... le programme « plante » c-a-d s'arrête brusquement en affichant en rouge la pile des erreurs...

En effet, si vous décidez de **propager une exception dans la signature du main, le main étant le point d'entrée du programme : le programme va planter !!!! Il ne faut jamais faire cela....**

A retenir : jamais de **throws dans la signature d'un main !!!!**

La méthode **main** est vraiment le **dernier moment** où il faut absolument attraper les exceptions, qui pourraient remonter jusque-là, pour les traiter et éviter que l'application s'arrête brusquement.

Bien évidemment, vous pouvez attraper vos exceptions *avant* le dernier moment (avant le **main**), à vous de voir dans votre programme, quelle méthode sera la plus pertinente pour au moment de l'exécution de l'application traiter au mieux vos exception pour continuer l'exécution de votre application dans les meilleures conditions possibles.

Effacez le **throws `DivisionParZeroException`** de la signature du **main** et sauvegardez : vous devez retrouver toutes vos erreurs de compilation que nous allons maintenant gérer par des **try...**

catch 😊

b. Illustration du cas gérer l'exception en la capturant et la traitant (try...catch)

Pour pouvoir compiler, il va falloir résoudre une à une toutes les erreurs de compilation liées à la **DivisionParZeroException** ...

❑ Mise en place d'un simple try...catch classique :

Pour corriger l'erreur de compilation de l'instruction :

```
int resultat = calculatrice.diviser(dividende, diviseur);
```

Choisissez cette fois ci-ci l'option **Surround with/try catch** et complétez le catch par les deux instructions suivantes qui se trouvent également dans les autres catches :

```
System.out.println("La division par zéro est impossible");
```

```
System.out.println("L'exception capturée est : "+e1.getMessage());
```

Et n'oubliez pas d'instancier la variable **resultat** à 0 pour pouvoir compiler 😊

❑ Mise en place d'un try...catch pour attraper multiples exceptions :

La deuxième erreur de compilation se situe autour de ce bloc d'instructions :

```
try {  
    resultat = calculatrice.diviser(42, 0);  
} catch (ArithmeticException e) {  
    System.out.println("La division par zéro est impossible");  
    System.out.println("L'exception capturée est :"+e.getMessage());  
}
```

Il faut « à tout prix » attraper l'exception **DivisionParZeroException** :

⇒ **option 1** : Si vous utilisez l'IDE et choisissez l'option **Add exception to existing clauses**, vous constaterez qu'il est possible d'attraper **plusieurs exceptions dans un même catch à l'aide de l'opérateur |**

⇒ **option 2** : Si vous utilisez l'IDE et choisissez l'option **Add catch clause to surrounding try catch**, vous pouvez constater qu'il est possible de faire une suite de **catch** pour un même **try**. L'exception ne tombera que dans un seul catch, c'est pour cela qu'il est important d'ordonner plusieurs catches de **l'exception la plus spécifique à la plus générale**.

Remarque : Cet exemple d'introduction ne contient qu'une seule instruction à l'intérieur du **try**. Il n'est donc pas très pertinent ici de capturer à la fois **ArithmeticException**, puis **DivisionParZeroException** font le même job. Cet exemple est juste là pour vous illustrer comment s'organiser quand on fait face à des exceptions multiples.

Par contre, dans des programme plus complexes, vos try seront couramment amenés à englober plusieurs lignes d'instructions qui pourront déclencher plusieurs types d'exceptions, vous devrez alors réfléchir s'il est plus pertinent de :

- traiter toutes les exceptions dans le même catch avec même comportement (option 1)
- Ou de proposer différents catch pour certaines exceptions plus spécifiques avec certains comportements particuliers à traiter (option 2)

Pour continuer cet exercice, faites en sorte que ce bloc de code compile avec l'option 1 (opérateur |)

❑ Cas de la capture d'une exception « générale »

Le code suivant ne soulève aucune erreur de compilation car la **DivisionParZeroException** EST-UNE *Exception* : elle tombe donc dans le **catch** déjà existant et est donc bien gérée par le programme 😊

```
try {  
    resultat = calculatrice.diviser(42, 0);  
} catch (Exception e) {  
    System.out.println("La division par zéro est impossible");  
    System.out.println("L'exception capturée est :b"+e.getMessage());  
}
```

Votre projet compile désormais sans erreur de compilation !

Vous pouvez exécuter le **main** de la classe **SandBox** sans problème et vérifier que le programme ne s'arrête plus brusquement, mais que les exceptions sont bien gérées dans le **catch**...

Et maintenant on peut se poser la question : **Quid des tests ?** ... parce qu'en réalité des erreurs de compilations sont apparus dans les tests...

5. Exception & Test

Revenez dans le fichier **CalculatriceTest**.

❑ Cas de test autour d'un fonctionnement « normal » (sans risque de déclencher l'exception)

C'est le cas pour la méthode `doitDiviserDeuxEntiers` qui maintenant ne compile plus puisqu'il faut gérer l'exception `DivisionParZero` à tout prix...

Dans cette méthode, l'**objectif** de l'exemple choisi était **d'illustrer le comportement « normal » de la division de deux entiers**.

Les paramètres passés à la méthode **diviser** ne déclencheront donc jamais l'exception (ce n'est pas l'objectif de ce test), c'est pourquoi dans ce cas-là le choix le plus pertinent est de **gérer l'exception en la propageant avec un throws...**

```
@Test
public void doitDiviserDeuxEntiers() throws DivisionParZeroException {...}
```

❑ Cas de test autour du « bon » déclenchement de l'exception

La mise en place de l'exception dans la méthode **diviser** de la classe **Calculatrice** apporte un nouveau comportement à notre programme. Il est donc de bon ton d'écrire un test autour de cette nouvelle exception et de ces nouvelles lignes de code dans la code métier 😊

En vous inspirant du test écrit précédent, écrire le test `doitLeverUneDivisionParZeroException` qui permet vérifier que l'exception **DivisionParZero** est bien levée

Ce petit exercice a permis d'introduire la notion d'exception. Nous retravaillerons sur les exceptions la semaine prochaine et d'ici là, voici de quoi en savoir plus sur cette notion ...

Travail à faire pour la semaine prochaine : Lire tranquillement chez vous :

- ✓ La rubrique sur les exceptions de <https://www.baeldung.com/java-exceptions>
- ✓ La Javadoc sur la présentation de la notion d'exceptions :
<https://docs.oracle.com/javase/tutorial/essential/exceptions/>

Partie 2 : Prise en main de maven

HelloMaven : votre premier projet maven

⇒ Rendez-vous sur : <http://unil.im/eclipsemaven>

Remarque : L'URL non raccourcie est accessible depuis le dépôt Back2Basics_Developpement :
https://github.com/iblasquez/Back2Basics_Developpement/blob/master/CreerProjetMavenEclipse.md

Gardez bien ce lien pour pouvoir récupérer rapidement le contenu du fichier **pom.xml** qui vous servira de base dans tous les projets maven que vous créerez dorénavant.

... Il faut bien avoir fini le tutoriel sous maven avant de continuer ...

- ⇒ **Pour créer une classe**, placez-vous sur le source folder **src/main/java** et comme pour un projet java classique à l'aide d'un clic droit, choisissez **New → Class**
Créez une classe Main avec une méthode main qui affichera un **Hello Maven** dans la console 😊
- ⇒ **Pour exécuter un projet**, nous continuerons pour le moment à le faire depuis l'IDE comme pour un projet classique avec un **Run As → Java Application**
Exécutez votre **main** pour voir s'afficher dans la console le fameux Hello Maven
Remarque : il est également possible de lancer les projets maven en ligne de commande à l'extérieure de l'IDE, mais c'est une autre histoire que l'on vous racontera plus tard 😊

Pour cette ressource, nous utiliserons Maven de la manière la plus simpliste qu'il soit c.-à-d. juste pour juste pour travailler avec une structure où le code de production et le code de test sont bien séparés, et aussi pour nous faciliter l'ajout et la gestion des dépendances grâce aux balises `<dependency>` du **pom.xml**.

Il faut toutefois savoir que Maven permet d'aller beaucoup plus loin et de gérer tout le cycle de vie d'un projet 😊

Partie 3 : Configurer la gestion de version d'un projet Maven (avec git sous Eclipse)

Exercice 1 : Créer un projet maven canard

Créer un **projet maven** que vous appellerez **canard** et dont le **pom.xml** exposera :

- encodage UTF-8,
- une version de Java : **1.8** (comme cela il sera impossible d'écrire du code dans les interfaces)
- et une dépendance vers **JUnit5**.

Vous pourrez continuer quand votre arborescence de projet ressemblera à celle-ci-donnée dessous, notamment avec **[JavaSE-1.8]** indiqué après **JRE System Library**

N'oubliez pas de faire un *Close Unrelated Project* pour n'avoir que le projet **canard** actif.

Exercice 2 : Créer un dépôt git local sur un projet existant (avec Eclipse)

Git est intégré à Eclipse, vous allez pouvoir utilisé *git* sans sortir d'Eclipse à l'aide du plug-in **Egit** installé de base dans cet IDE 😊

Pour faire l'équivalent d'un « **git init** » sous Eclipse, c.-à-d. dire à Eclipse qu'on souhaite que notre projet soit géré par git, placez-vous sur le projet **canard** dans la vue **Package Explorer**, puis d'un clic droit sélectionnez **Team -> Share project**

Il faut ensuite dire à Eclipse de créer un dépôt local git dans le répertoire du projet
Pour cela, une fenêtre s'ouvre alors pour configurer les options du dépôt git c-a-d choisir à quel endroit l'on souhaite créer son dépôt Git.

Remarque : Sous git, on travaille avec un **dépôt** (ou **repository** en anglais).

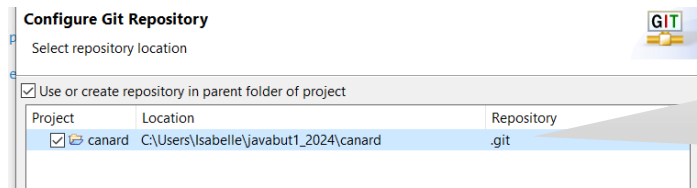
**Pour tous nos TP, on souhaite que le répertoire caché .git
soit créé à la racine du répertoire du projet**

pour que seules les modifications faites dans ce répertoire (dépôt) soit versionnées.

Pour cela, depuis l'IDE Eclipse, vous devrez donc toujours suivre la procédure suivante :

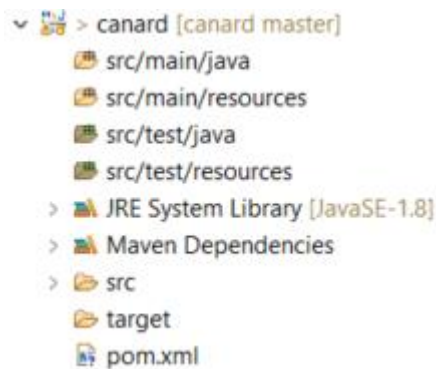
- Vérifiez que votre projet (pour cet exercice c'est le projet **canard**) est bien proposé dans la fenêtre **Configure Git Repository** qui vient de s'ouvrir.
- Pour rendre accessible le bouton **Create Repository**, vous devez cocher la case **Use or create repository in parent folder of project** et décocher la case devant le nom du projet dans la colonne **Project**
- Cliquez ensuite sur le bouton **Create Repository** de manière à voir apparaître uniquement **.git** dans la colonne **Repository** ce qui signifie que le répertoire **.git** va être créé dans le

répertoire courant, donc dans le répertoire du projet. Soyez très attentif à ce point, sinon vous aurez de mauvaises surprises par la suite 😊



Ne pas continuer tant qu'il n'y a pas uniquement .git dans la colonne Repository !!!

- Une fois que **.git** apparaît seul dans la colonne Repository vous pouvez cliquer sur **Finish**.



Si vous avez bien respecté les consignes ci-dessus, il ne vous reste plus qu'à jeter un petit coup d'œil dans la **vue Package Explorer**, le nom de votre projet a dû être modifié (si ce n'est pas la cas, procédez à un petit rafraîchissement à l'aide de F5) et est désormais suivi d'un prompt entre crochet :

>canard [canard master]

La présence prompt **[canard master]** montre que le projet est géré par un gestionnaire de version et que vous êtes

actuellement sur la branche **master**

*... Avant de continuer, vérifiez que tous les autres projets de votre workspace sont bien fermés et que le projet **canard** est bien le seul projet ouvert de votre workspace ...*

Exercice 3 : A propos du fichier **.gitignore**

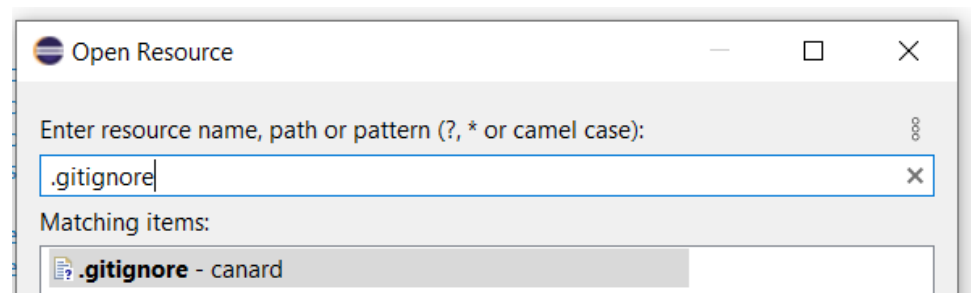
❑ Mais où se cache le **.gitignore** ? :

Dans la vue **Package Explorer**, nous ne voyons pas les fichiers cachés, mais où se cache le fichier **.gitignore** et comment l'afficher dans l'éditeur de l'IDE.

⇒ **Pour (re)trouver le fichier .gitignore**, nous allons utiliser la **boîte de dialogue Open Resource** qui s'ouvre :

- Soit à partir du menu **Navigate -> Open Resource ...**
- Soit à partir du raccourci : **Ctrl + Shift + R** comme indiqué dans le menu Navigate

Il ne vous reste plus qu'à entrer le nom du fichier à chercher :



⇒ **Pour afficher le fichier dans l'éditeur**, double-cliquer sur le fichier recherché (qui est censé apparaître) dans la rubrique **Matchings items** de la boîte de dialogue Open Resource.

❑ Contenu actuel du fichier .gitignore

Le fichier **.gitignore** est un fichier technique utilisé par *git* pour spécifier les fichiers et répertoires à ne pas prendre en compte dans la gestion des versions (c-à-d ceux qui ne seront pas versionnés). Pour l'instant, ce fichier contient **/target/** ce qui signifie que les fichiers contenus dans le répertoire **target** ne seront pas soumis au gestionnaire de version.

Il est à noter que le répertoire **target** contient le **byte code** des classes c.-à-d. le code java compilé au travers de fichiers **.class** : le répertoire **target** contiendra donc le résultat issu de la compilation des classes.

Cela ne sert à rien de confier à git le répertoire target c.à.d. la gestion des versions compilées car les versions compilées pourront toujours être recompilés à partir du code source.

Exercice 3 : Effectuer un premier commit (Team → Commit)

L'opération de « sauvegarde » avec un gestionnaire de version s'appelle un **commit**.

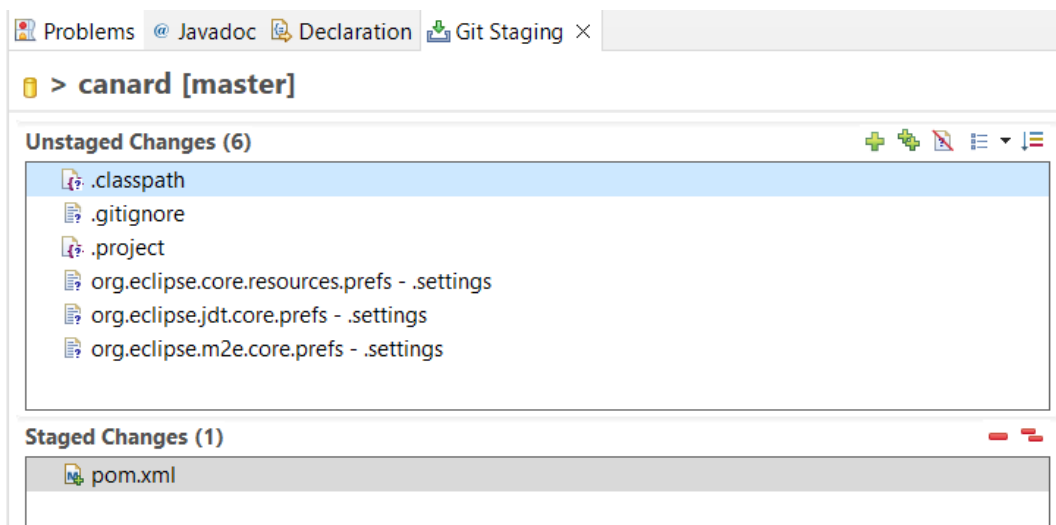
Pour **commiter** les modifications apportées au projet, il suffit de revenir sur la **vue Package Explorer** de se positionner sur **le projet (canard pour nous)**, puis d'un clic droit de sélectionner **Team -> Commit...**

Une fenêtre s'ouvre qui n'est autre que la **vue Git Staging** qui permet de configurer le commit :

❑ pom.xml : un fichier à versionner :

Dans le cadre de la rubrique **Unstaged Changes** se trouvent les fichiers susceptibles d'être enregistrés par git au moment où le commit est réalisé. Comme indiqué sur la copie d'écran ci-dessous, on souhaite que le fichier de configuration **pom.xml** puisse être versionné tout au long du projet.

Pour cela à l'aide d'un **drag & drop faites glisser pom.xml** de la rubrique **Unstaged Changes** vers la rubrique **Staged Changes** :



Notez que désormais le fichier **pom.xml** a une **petite croix verte sur son icône en bas à droite** qui indique qu'il est *tracké* par git 😊

❏ des fichiers à ignorer ...

Prenons les fichiers dans l'ordre d'apparition dans le cadre de la rubrique **Unstaged Changes** qui pour l'instant ont un **point d'interrogation (?)** sur leur icône en bas à droite ce qui indique qu'il faut prendre une décision de les *tracker/stager* en les basculant dans la case de la rubrique Staged Changes ou de les *ignorer* en les faisant apparaître dans le fichier `.gitignore` 😊

⇒ le fichier `.classpath` est un fichier technique utilisé dans les projets java « simple ».
En effet, tout développeur connaît le « **classpath** qui permet de préciser au compilateur et à la JVM où ils peuvent trouver les classes dont ils ont besoin pour la compilation et l'exécution d'une application. C'est un ensemble de chemins vers des répertoires ou des fichiers `.jar` dans lequel l'environnement d'exécution Java recherche les classes (celles de l'application mais aussi celles de tiers) et éventuellement des fichiers de ressources utiles à l'exécution de l'application. Ces classes ne concernent pas celles fournies par l'environnement d'exécution incluses dans le fichier `rt.jar` qui est implicitement utilisé par l'environnement. » (extrait <https://www.jmdoudoux.fr/java/dej/chap-techniques-base.htm>)
Notre `pom.xml` a été configurée dans ce sens.

Nous n'avons donc pas besoin du `.classpath` quand nous versionnons un projet maven 😊

✓ Nous allons donc maintenant faire en sorte que le fichier `.classpath` soit Ignoré 😊

Pour indiquer dans le fichier `.gitignore` que le fichier `.classpath` ne doit pas être soumis au gestionnaire de version, placez-vous dans la rubrique **Unstaged Changes** sur `.classpath`, puis à l'aide d'un clic droit choisir **Team -> Ignore**.

→ La ligne `/.classpath` est alors ajoutée dans le fichier `.gitignore`.

→ Le fichier `/.classpath` a disparu de la rubrique **Unstaged Changes** et n'est également pas présent dans la rubrique **Staged Changed** : donc il est bien ignoré maintenant par git.

⇒ le fichier `.gitignore` est un fichier technique qui est propre à l'IDE, nous le verrons rapidement en ignorant par la suite le répertoire `settings` propre à Eclipse. Comme chaque développeur d'une équipe développe avec son IDE préféré, le fichier `.gitignore` n'est en principe pas versionné : c'est ce que je vous demanderais de faire pour les TP de ce module même si cette année tout le monde utilise Eclipse 😊

✓ De la même manière que précédemment, faites-en sorte que le fichier `.gitignore` soit Ignoré

⇒ le fichier `.project` est un fichier technique propre à l'IDE Eclipse

✓ De la même manière, faites-en sorte que le fichier `.project` soit Ignoré

⇒ le répertoire **.settings** est un *répertoire technique propre à l'IDE Eclipse* utilisé pour enregistrer des informations sur le projet

✓ Placez-vous sur le premier fichier du répertoire .settings qui apparaît dans votre rubrique **Unstaged Changes** (org.eclipse.core.resources.prefs - .settings)

- puis à l'aide d'un clic droit choisir **Team -> Ignore**
- continuez cette manipulation pour tous les fichiers : ils disparaissent alors de Unstaged Changes sans apparaître dans Staged Changes 😊

✓ Remarque : si vous voulez faire apparaître explicitement dans le fichier .gitignore que vous Souhaitez ignorer tout le contenu du répertoire settings, vous pouvez ajouter la ligne suivante de la même manière que target qui était aussi un répertoire 😊

/settings/

Et n'oubliez pas de vérifier que le fichier **.gitignore** est bien sauvegardé 😊

Arrivé à cette étape, le contenu de votre fichier **.gitignore** doit ressembler à :

```
/target/  
/.classpath  
/.gitignore  
/.project  
/settings/
```

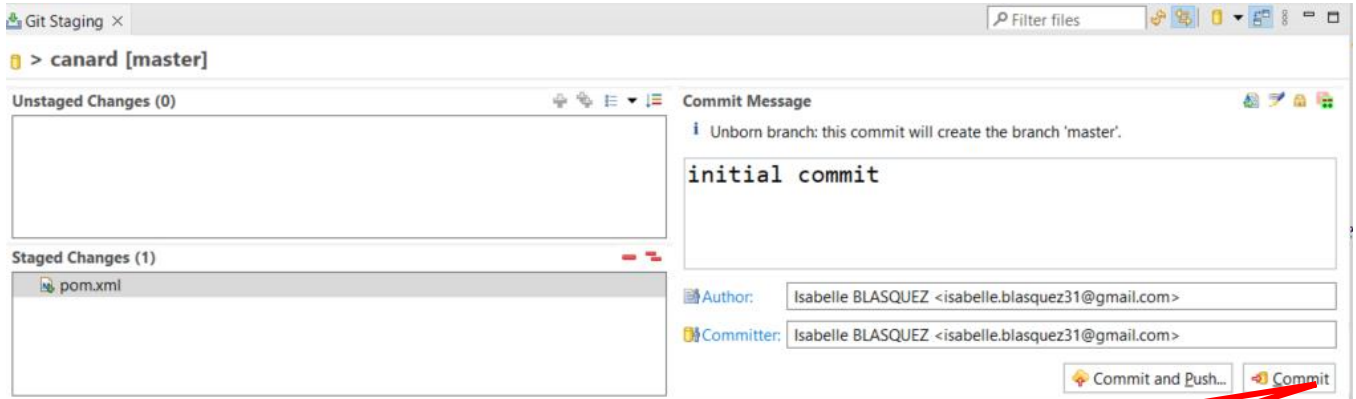
Remarque : Si on le souhaite, il est possible d'ajouter des commentaires dans ce fichier. Un commentaire doit commencer par le symbole #. Les commentaires sont optionnels.

Ajoutez par exemple les commentaires #Eclipse et #Maven dans votre fichier .gitignore afin que son contenu ressemble désormais à :

```
#Maven  
/target/  
  
#Eclipse  
/.classpath  
/.project  
/.settings/  
  
/.gitignore
```

❑ Pas de commit anonyme !

Rappelez-vous qu'avec *git* il n'y a pas de commit anonyme et que vous devez indiquer l'**auteur** des modifications et la personne qui a commité en complétant les rubriques **Author** et **Committer** avec votre nom et adresse mail.



Commit en local !!!

❑ Pas de commit sans message !

⇒ Enfin il faut saisir le **message** qui explicitera la teneur du commit dans la rubrique **Commit message**. Ce message est obligatoire sinon vous ne pourrez pas commiter 😊

Par convention, il est d'usage d'appeler le premier message de commit : **initial commit** car dans ce commit, on a juste créé le projet avec un **pom.xml** (et configurer le **.gitignore**)

Rappelez-vous que les messages de commit doivent être explicites pour pouvoir aisément se retrouver dans l'historique des commits (voir cours).

⇒ Validez le commit en appuyant sur **Commit** (attention, c'est bien le bouton où est **simplement écrit Commit** pour faire un historique en local)

⇒ Jetez alors un petit coup d'œil dans la vue **Package Explorer**, le fichier **pom.xml** est désormais précédé d'une **icône avec une petite croix verte en bas à droite** ce qui signifie que ce fichier est désormais sous contrôle du gestionnaire de version.

Partie 4 : Apprendre à versionner régulièrement son projet durant la phase d'implémentation

Le projet canard a été créé pour implémenter le diagramme de classes proposé comme corrigé de l'exercice : **Et si on cancanait un peu ?**

Dans ce diagramme, nous pouvons identifier trois parties :

- La hiérarchie autour de la classe **Canard**
- La hiérarchie autour de l'interface **ComportementVol**
- La hiérarchie autour de l'interface **ComportementCancan**

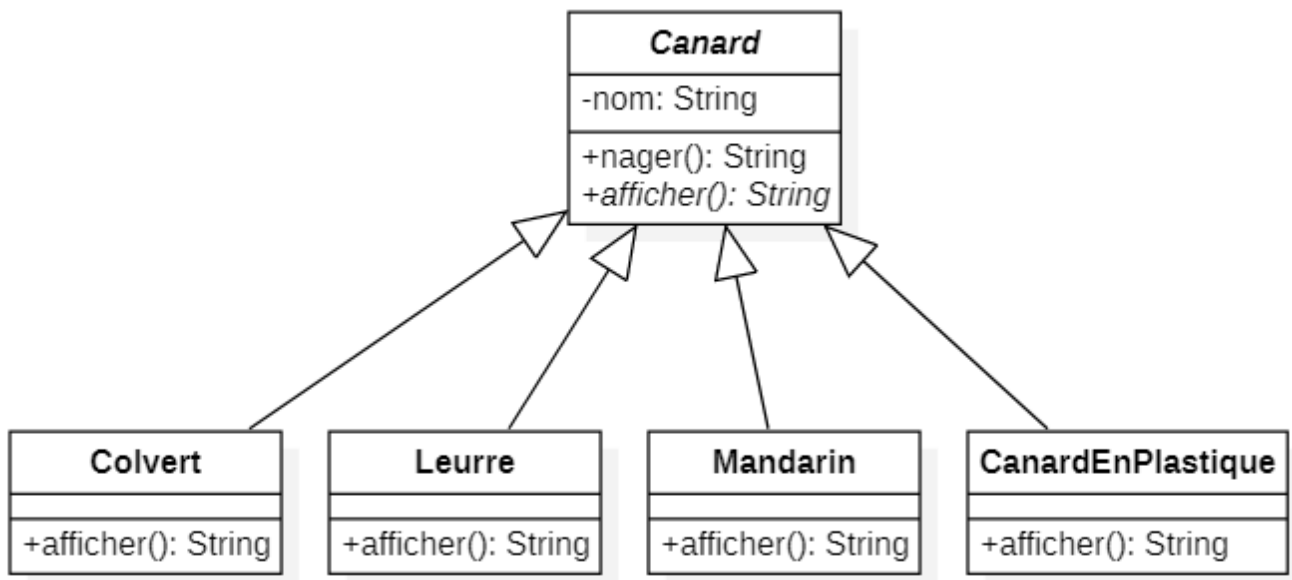
Nous allons organiser notre développement en différentes étapes autour de ces trois parties.

Etape 1 : Implémentation de la hiérarchie autour de la classe Canard.

Nous allons commencer par implémenter la hiérarchie autour de **Canard** sans tout ce qui concerne le cancan ou le vol.

a. Implémenter

Pour cette étape, dans un paquetage **canard.model**, implémentez le diagramme de classes suivant.



Notez que sur ce diagramme, nous avons ajouté un attribut **nom** à la classe **Canard**.

Pour ne pas alourdir le diagramme, nous avons choisi de faire apparaître ni les constructeurs, ni les éventuels getteurs et setteurs...

❑ b. Vérifier et Valider le « bon » comportement de l'implémentation (à l'aide d'un jeu d'essai) :

A noter, qu'on pourrait aussi très bien faire cette vérification à l'aide d'un jeu de tests automatisés 😊

Dans un package **canard.application**, récupérez la classe **Simulateur** disponible sur **gist** :

<https://unil.im/simucanard>
(<https://gist.github.com/iblasquez/3d8f3768de002965d0f97ef2ce5eb8f5>)

Une fois l'implémentation du diagramme de classes réalisée, lancez le **main** de la classe **Simulateur** afin de vérifier que vous obtenez bien sur la console l'affichage suivant :

```
-----  
Afficher et Nager  
-----  
Piero : Je suis un vrai colvert  
Tous les canards flottent, même les leurres!  
Danny : Je suis un leurre  
Tous les canards flottent, même les leurres!  
Oshidori : Je suis un vrai mandarin  
Tous les canards flottent, même les leurres!  
Rubber : Je suis un canard en plastique  
Tous les canards flottent, même les leurres!
```

❑ c. Commiter !!!

Une fois l'implémentation vérifiée et validée, il est alors possible de commiter !

Placez-vous sur le projet canard, puis clic droit puis **Team -> Commit...**

Transférez les fichiers **Canard.java**, **CanardEnPlastique.java**, **Colvert.java**, **Leurre.java** et **Mandarin.java** et **Simulateur.java** de **Unstaged Changes** vers **Staged Changed**

Astuce : Vous pouvez les transférer un à un ou tous d'un coup en cliquant sur la double croix verte 😊

Remplissez **Commit Message** de manière explicite, par exemple :

ajout héritage Canard et Simulateur

Vous pouvez alors appuyer sur le bouton **Commit** pour committer en local.

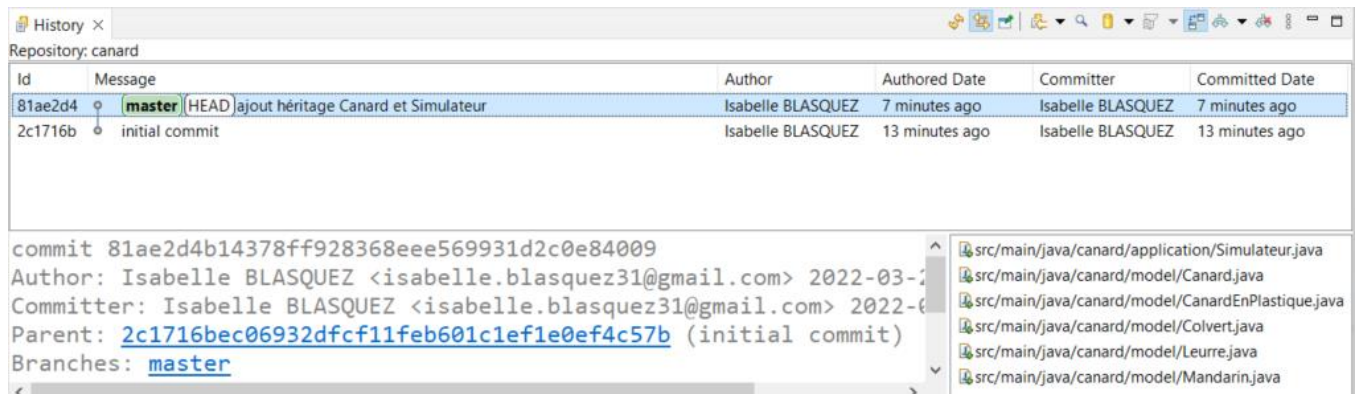
Remarque : On aurait aussi pu committer classe par classe, mais comme il y a pour le moment très peu de code dans les classes filles, le commit de **toutes les classes de la hiérarchie** est **cohérent** ici. Il faut retenir que les commits doivent être « petits » justement pour avoir des messages de commit qui les décrivent au mieux...

Bonnes pratiques de commit :

- ✓ Toujours écrire un **message de commit explicite** illustrant l'apport fonctionnel qui vient d'être implémenté.
- ✓ **Toujours committer un code qui marche** (une application fonctionnelle/opérationnelle). S'il n'est pas possible en fin de journée de committer un code qui marche car vous n'avez pas encore fini de développer la fonctionnalité, dans ce cas-là pensez à bien indiquer comme premier mot du message de commit **WIP** (pour **Work In Progress**) suivi du nom de la fonctionnalité que vous êtes en train de développer !

Visualiser l'historique des commits ...

Placez-vous sur le projet **canard** dans la vue **Package Explorer**, puis faites un clic droit afin de sélectionner **Team -> Show in History**



❑ La vue **History** s'ouvre. Elle permet de visualiser graphiquement l'ensemble des commits et leur message (ou plutôt le titre du message).

Cet historique montre l'importance d'avoir des messages de commit explicites.

Si ces messages ne sont pas explicites, il peut très difficile, ensuite, de se retrouver dans l'historique...

❑ Chaque commit est bien représenté par son **Id** qui est une version réduite du *hashcode complet* qui est consultable dans la case en bas à gauche (voir cours).

Dans la fenêtre en bas en droite, on retrouve le détail des fichiers qui ont été impactés par ce commit.

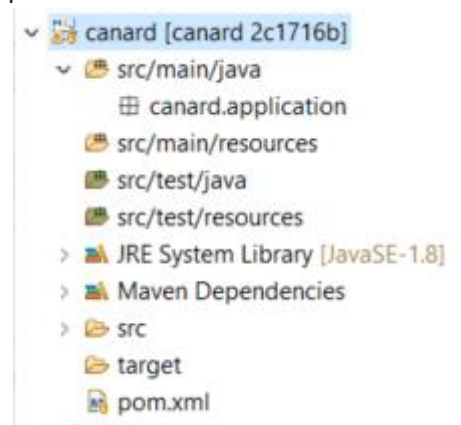
❑ Pour revenir sur un commit, il suffit de le sélectionner dans l'historique et demander un *checkout*. Par exemple, placez-vous sur le **initial commit** puis clic droit

Checkout

Eclipse affiche un warning **Detached HEAD**, ignorez-le pour l'instant et cliquez sur **Close**.

Jetez un petit coup d'œil dans la vue Package Explorer.

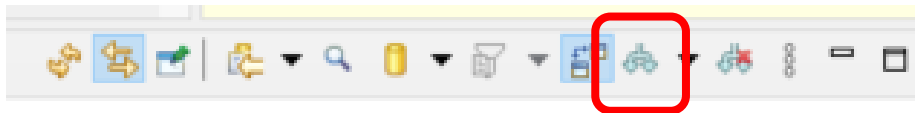
- Le projet est revenu dans l'état dans lequel il était au moment de ce commit : juste avec le fichier **pom.xml**
- le crochet qui suit le nom du projet n'indique plus master, mais **l'Id du commit sur lequel vous êtes** (vous n'aurez pas le même Id que moi 😊)



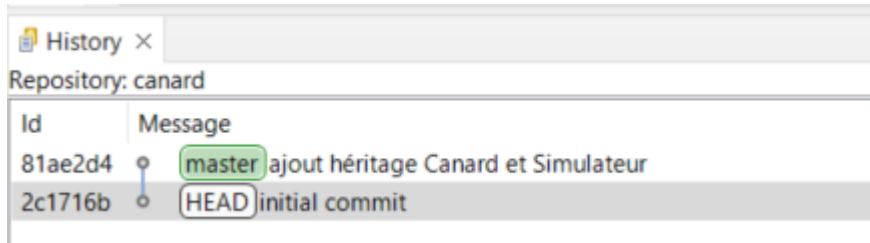
Attention, vous ne pouvez pas écrire de code sur ce commit, vous êtes juste là on observateur !!!

Regardez maintenant l'historique, le **marqueur HEAD** s'est déplacé sur le commit que vous êtes en train de regarder c'est justement le rôle du **HEAD** que de montrer le commit spécifique que l'on est en train de regarder.

La vue actuelle permet uniquement de voir le commit où on s'est déplacé et ses parents
Pour revenir sur la branche **master**, cliquez sur le bouton suivant (de la vue **History**)



qui permet de configurer la vue et de voir non seulement le commit sur lequel on est et ses parents, mais aussi les commits qui ont été réalisés depuis (c-a-d ses enfants).

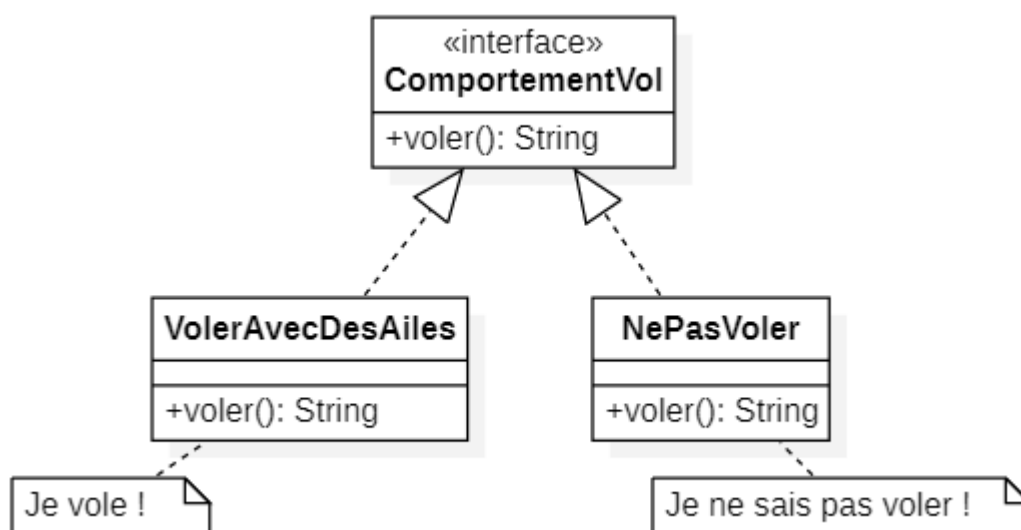


Pour revenir sur le dernier commit **master**, il suffit donc de se replacer sur master et de faire un clic droit puis **Checkout**
... et de vérifier dans la vue **Package Explorer** qu'on retrouve bien [**canard master**] à côté du nom de notre projet 😊

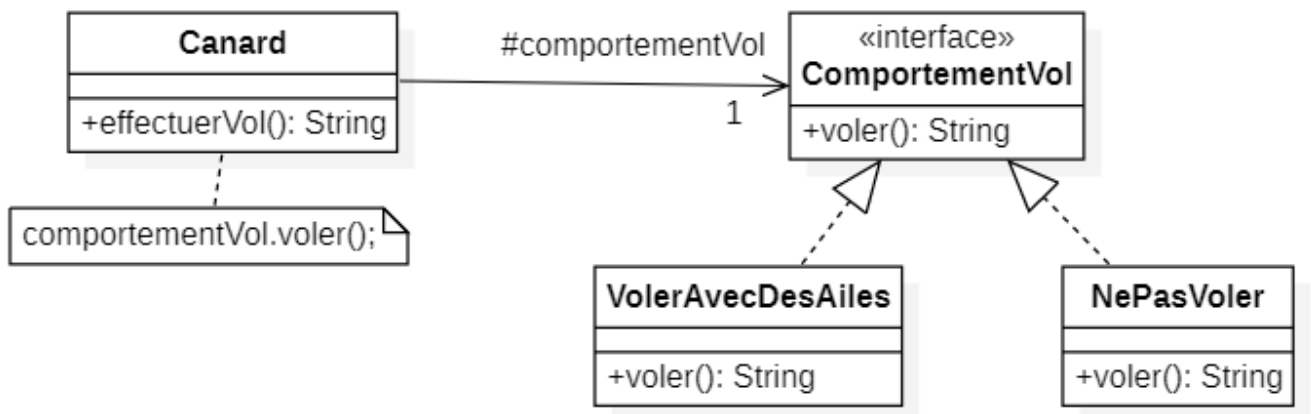
Etape 2: Implémentation de la hiérarchie autour de l'interface Comportement Vol

❑ a. Implémenter

- ➔ Pour cette étape, dans un paquetage **canard.model.vol**, implémentez le diagramme de classes suivant autour de l'interface **ComportementVol** :



➔ Modifiez ensuite la classe Canard pour qu'un canard puisse AVOIR-UN comportement de vol



Et remémorez-vous l'expression des besoins (l'énoncé) qui précisait que lorsqu'il vient au monde :

- un petit colvert est un *vrai* canard qui est capable de voler avec de ses propres ailes.
- un petit mandarin est un *vrai* canard est capable de voler avec de ses propres ailes.
- un petit leurre ne doit pas voler
- un petit canard en plastique ne doit pas voler

□ b. Vérifier et Valider le « bon » comportement de l'implémentation (à l'aide d'un jeu d'essai) :

A noter, qu'on pourrait aussi très bien faire cette vérification à l'aide d'un jeu de tests automatisés 😊

Dans la classe **Simulateur** :

- **ne touchez pas à la méthode `mettreDesCanardsDansMonSimulateur`**
- renommez (clic droit : Refactor → Rename) la méthode `faireAfficherEtNager` en **`faireAfficherNagerVoler`** et ajouter dans le **for** un appel à la méthode `effectuerVol` de manière à obtenir l'affichage suivant sur la console :

```
-----
Afficher, Nager et Voler
-----
Piero : Je suis un vrai colvert
Tous les canards flottent, même les leurres!
Je vole !
Danny : Je suis un leurre
Tous les canards flottent, même les leurres!
Je ne sais pas voler
Oshidori : Je suis un vrai mandarin
Tous les canards flottent, même les leurres!
Je vole !
Rubber : Je suis un canard en plastique
Tous les canards flottent, même les leurres!
Je ne sais pas voler
```

❑ c. Commiter !!!

Une fois l'implémentation vérifiée et validée, il est alors possible de committer !
Placez-vous sur le projet canard, puis clic droit puis **Team -> Commit...**

Transférez les fichiers de **Unstaged Changes** vers **Staged Changed**

Remplissez **Commit Message** de manière explicite, par exemple :
ajout ComportementVol à Canard

Vous pouvez alors appuyer sur le bouton **Commit** pour committer en local.

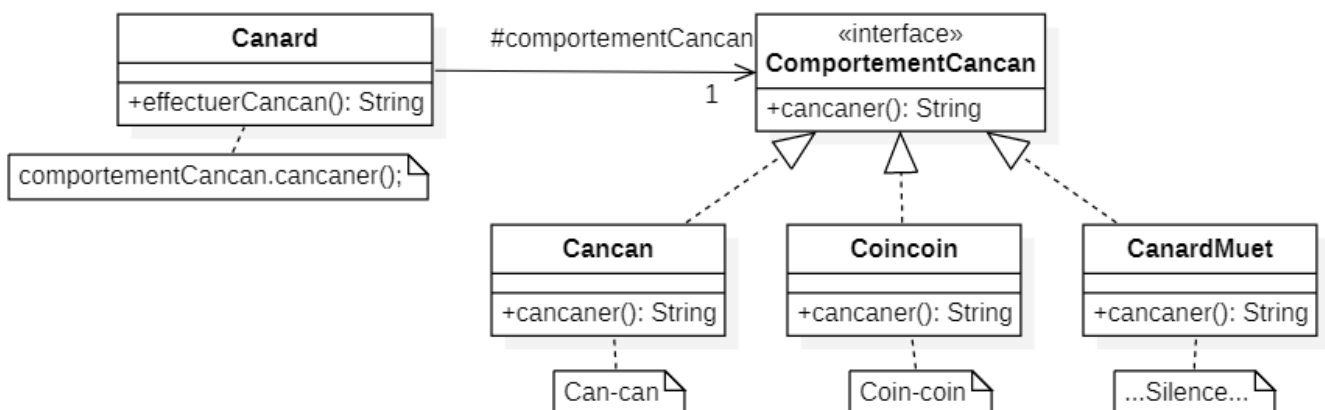
Etape 3: Implémentation de la hiérarchie autour de l'interface Comportement Cancan

❑ a. Implémenter

De la même manière que précédemment, implémentez dans un paquetage **canard.model.cancan**, la hiérarchie d'héritage autour de l'interface **ComportementVol** représentée sur le diagramme de classes ci-dessous.

Puis modifiez ensuite la classe Canard pour qu'un canard puisse AVOIR-UN comportement de cancan en tenant compte de l'expression des besoins (l'énoncé) qui précisait que lorsqu'il vient au monde :

- un colvert est capable d'émettre un cancanement (*can-can*)
- un mandarin qui est capable d'émettre un cancanement (*can-can*)
- un leurre n'émet aucun son (ce sera un canard muet 😊)
- un canard en plastique est capable d'émettre un couienement (*coin-coin*)



❑ b. Vérifier et Valider le « bon » comportement de l'implémentation (à l'aide d'un jeu d'essai) :

Dans la classe **Simulateur** :

- **ne touchez pas à la méthode mettreDesCanardsDansMonSimulateur**
- renommez (clic droit : Refactor → Rename) la méthode faireAfficherNagerVoler en **faireAfficherNagerVolerCancanner** et ajouter un appel à la méthode **effectuerCancan** et un saut de ligne de manière à obtenir l'affichage suivant sur la console :

Afficher, Nager, Voler et Cancaner

Piero : Je suis un vrai colvert
Tous les canards flottent, même les leurres!
Je vole !
Can-can

Danny : Je suis un leurre
Tous les canards flottent, même les leurres!
Je ne sais pas voler
...Silence...

Oshidori : Je suis un vrai mandarin
Tous les canards flottent, même les leurres!
Je vole !
Can-can

Rubber : Je suis un canard en plastique
Tous les canards flottent, même les leurres!
Je ne sais pas voler
Coin-coin

❑ c. Commiter !!!

Une fois l'implémentation vérifiée et validée, il est alors possible de commiter !

Placez-vous sur le projet **canard**, puis clic droit puis **Team -> Commit...**

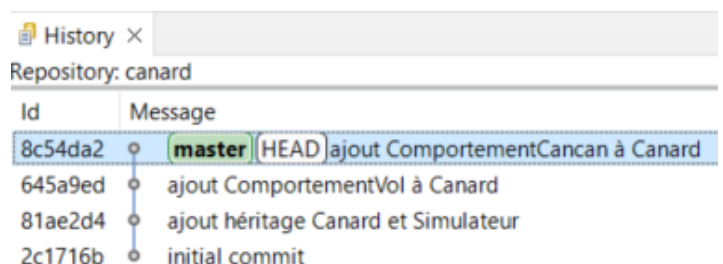
Transférez les fichiers de **Unstaged Changes** vers **Staged Changed**

Remplissez **Commit Message** de manière explicite, par exemple :

ajout ComportementCancan à Canard

Vous pouvez alors appuyer sur le bouton **Commit** pour committer en local.

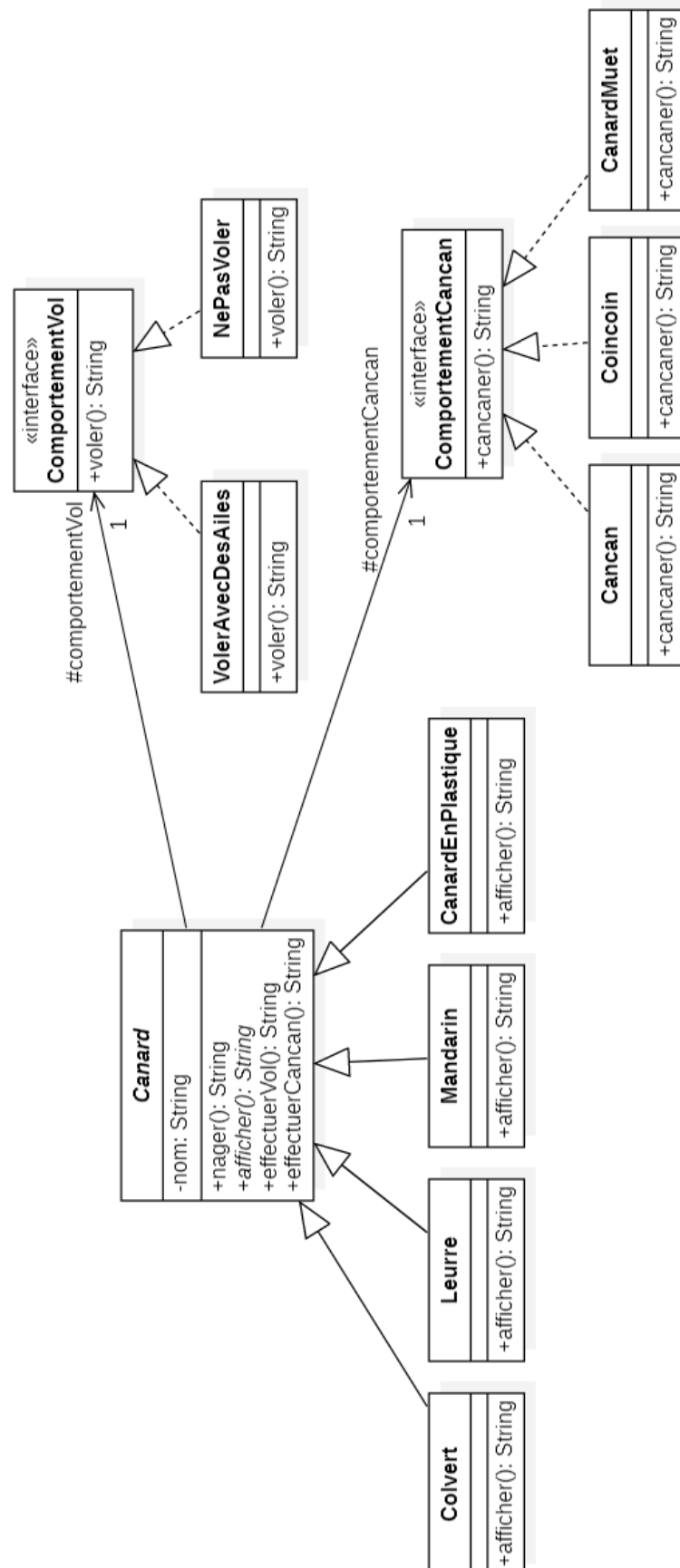
... Et jetez un petit coup d'œil sur **la vue History** pour vous convaincre que les messages de commit bien nommés permettent de retracer et retrouver facilement les différentes étapes de votre développement...



Un petit plus : Rétro-conception pour vérifier l'architecture du code implémenté...

En laissant le bouton CTRL appuyé, sélectionnez une à une les classes métier dans la vue package Explorer pour procéder à la rétro-conception à l'aide d'une vue PlantUML de votre code métier.

Vérifier que le diagramme de classes obtenu illustrant votre code est similaire au diagramme issu de la phase de conception donné ci-dessous :



Etape 4: Pousser son projet sur un remote pour la première fois

Il est possible de connecter un **remote** directement depuis Eclipse, mais la procédure est un peu lourde (longue et pénible : elle est décrite en détails ici : <https://unil.im/egit> si vous êtes intéressés),

Personnellement je préfère passer par **gitKraken**, c'est beaucoup plus rapide 😊

⇒ **Au préalable, Sous Eclipse**, commencez par **recupérer le chemin qui mène à votre projet**.

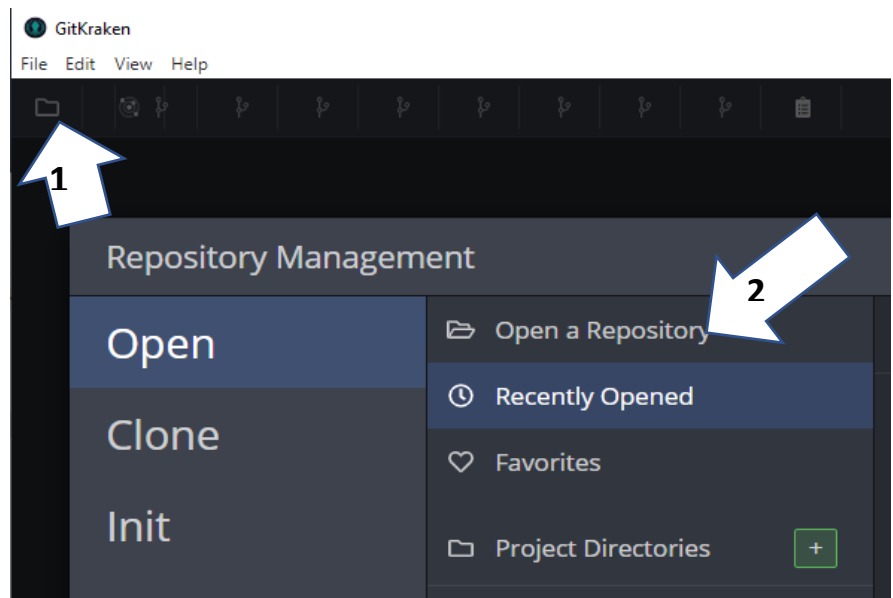
Nous l'avons déjà fait, il faut se placer sur le projet **canard** dans la **vue package Explorer**, puis clic droit, puis **Properties**. En cliquant sur le bouton au bout de la ligne **Location**, vous allez ouvrir votre explorateur de fichiers à l'emplacement de votre projet.

Copiez son chemin. Pour moi c'est : C:\Users\Isabelle\javabut1\canard

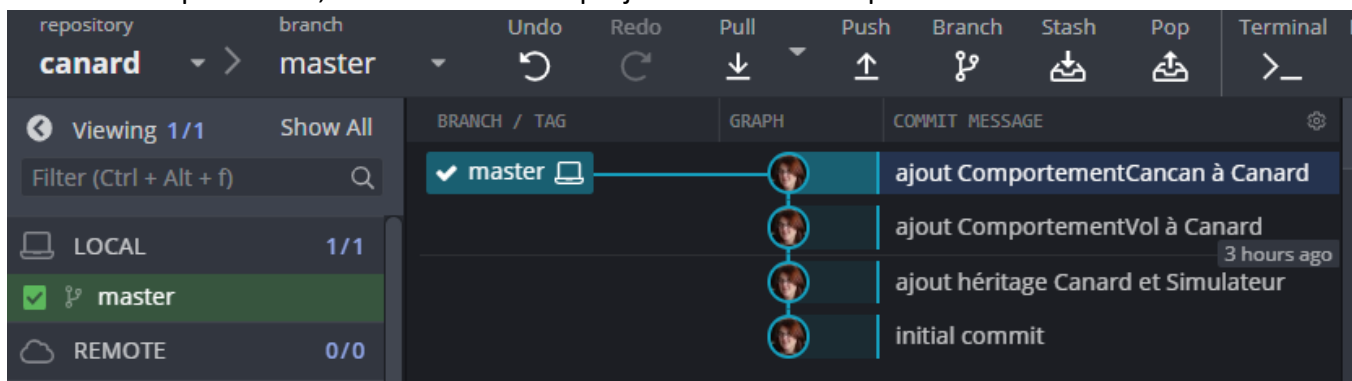
⇒ **Sous GitKraken**,

Cliquez sur l'icône en forme de répertoire en dessous du menu File (1) pour vous retrouver dans cette configuration et pouvoir choisir **Open a Repository** (2)

Rentrez bien le chemin qui mène jusqu'à votre projet canard (cliquez sur Sélectionnez Dossier quand vous êtes dans le projet **canard**)



Si tout se passe bien, vous devriez votre projet et votre historique dans GitKraken !



Pour ce TP, nous avons choisi **Github** comme **remote**

⇒ Avant de continuer, connectez-vous depuis votre navigateur web préféré à votre compte **Github** (que vous avez créé dans un tutoriel précédent) afin de bien vous rappelez de vos identifiants et mots de passe sur ce site 😊

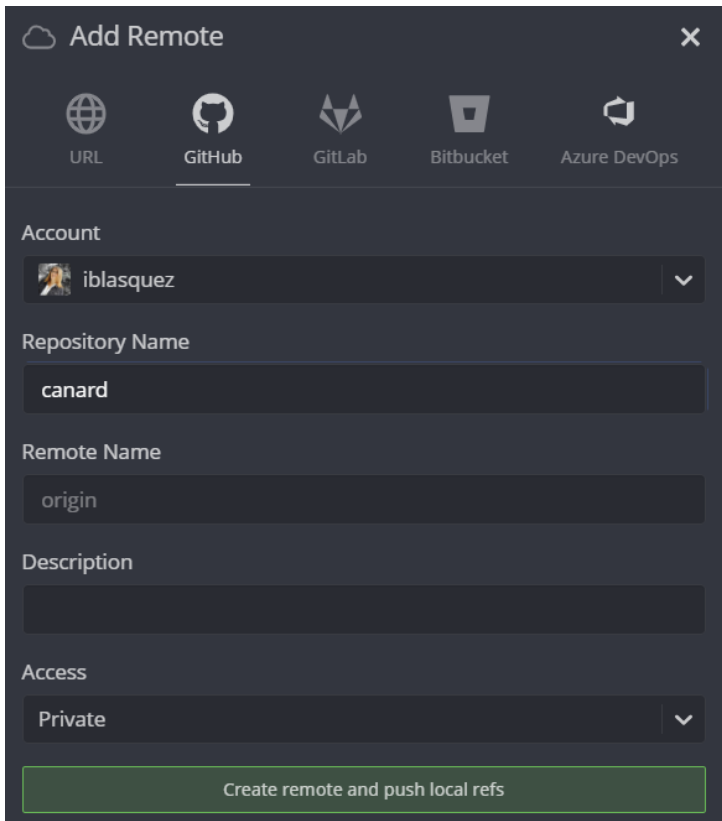
Nous allons maintenant procéder à la connexion du dépôt git local (que l'on visualise actuellement) à un dépôt sur le remote (qui va être créé lors du premier **push** vers ce dépôt : il est tout à fait possible de créer le dépôt avant sur le remote et de pousser après, mais c'est plus facile de pousser directement et de faire confiance au remote pour créer un dépôt du même nom 😊)

⇒ Depuis **GitKraken**, pour envoyer votre projet sur le remote (on dit plus communément **pousser**), il suffit de cliquer sur me

Pour cela, cliquez sur le **bouton Push** au-dessus de votre historique.

A la question *"There is no remote to push to, would you like to add one ?"*, répondez **Yes**

Une fenêtre **Add Remote** s'ouvre :



☐ Choisissez tout d'abord **l'icône de Github** puisque c'est le remote sur lequel vous voulez pousser votre travail 😊

☐ Dans **Account**, sélectionnez votre compte Github

☐ Dans **Repository Name**, laissez la valeur par défaut qui doit être le nom de votre projet. Ainsi, votre dépôt github aura le même nom de projet qu'en local.

☐ Dans **Remote Name**, laissez la valeur par défaut **origin**. Dans le cours, vous avez vu que c'était le nom utilisé habituellement pour désigner le nom du remote

☐ Laissez la **Description** vide, vous pourrez la compléter ultérieurement

☐ Dans **Access**, choisissez si vous souhaitez que votre dépôt soit public ou privé sous Github. Je vous conseille pour commencer de le mettre en **Private**, sachant que vous pourrez changer sa visibilité à tout moment depuis le site Github.

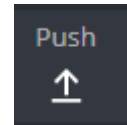
Il ne vous reste plus qu'à cliquer sur le bouton : **Create remote and push local refs**

Si c'est la première fois que vous poussez depuis GitKraken vers Github, vous devrez sûrement faire les étapes suivantes :

- renseignez votre **username** et **Password** de **Github** dans **gitKraken** et cliquez sur **Log In**

Please log in to your hosting service to continue: ☐ Remember me

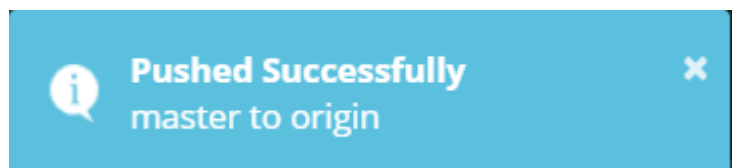
- Appuyez de nouveau sur le **bouton Push**, le message suivant apparaît



What remote/branch should "master" push to and pull from? /

Laissez tel quel et appuyez directement sur **Submit**, puisque c'est bien **master** qu'on souhaite pousser sur **origin**

⇒ Si ce **premier Push s'est bien**, un message du type **Pushed Successfully** s'affiche en bas à gauche de votre GitKraken



... sinon appelez votre enseignant de TP

⇒ Vérifiez ensuite que le dépôt local **canard** a bien été poussé sur le remote comme un nouveau dépôt nommé **canard** :

- Rendez-vous sur votre compte Github
- Cliquez sur **Repositories**
... et un dépôt **canard** de type **Private** qui a été mis à jour (updated) il y a quelques minutes doit apparaître au tout début de la liste de vos **dépôts (repositories)**



- Cliquez sur **canard** pour vous retrouver dans ce dépôt et baladez-vous pour bien constater que tous vos fichiers sont désormais présents sur le remote 😊

Comme dit précédemment, il y a d'autres manières de procéder pour connecter le local au remote ...

Pour les aficionados de la ligne de commande : Lorsqu'on crée un projet sur un remote, ce dernier indique en principe les lignes de commande à exécuter pour se connecter à ce projet. Vous pouvez donc faire directement ces manipulations en ligne de commande si le cœur vous en dit...

Etape 5 : Continuons notre développement en local

Ajout de tests automatisés

Pour vérifier et valider notre code, au lieu d'écrire des jeux d'essais dans la classe **Simulateur**, nous aurions pu écrire des jeux de tests automatisés dans une classe **CanardTest** par exemple.

Placez-vous sur le classe **Canard**,
à l'aide d'un clic droit **New → JUnit Test Case**.
Vérifiez bien que le source folder est bien
canard/src/test/java et cliquez sur **Finish**

de manière à faire apparaître comme dans la copie d'écran ci-contre la classe **CanardTest.java** dans le source folder **src/test/java**

Rendez-vous sur <https://unil.im/testcanard>
(<https://gist.github.com/iblasquez/0a72db1eeff312f446dad9d73b591e24>)

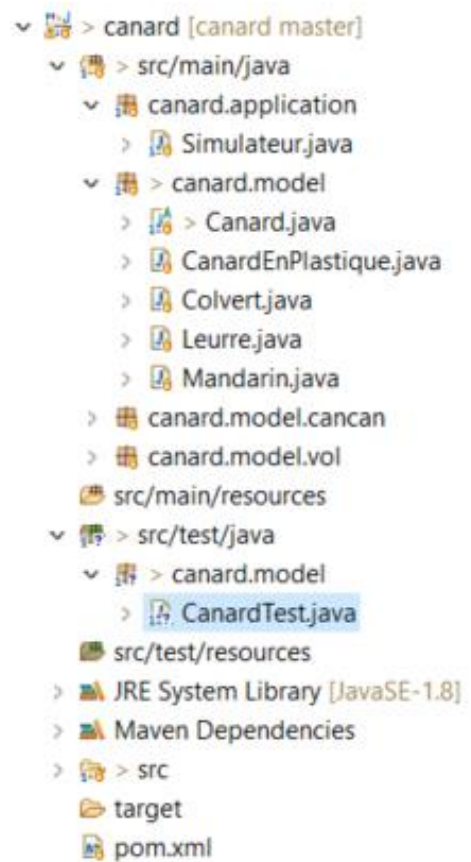
pour copier le code de la classe **CanardTest** qui aurait pu être écrit à la place de **Simulateur.java**

⇒ Consultez ce code et retrouvez comment les différentes fonctionnalités ont été testées.

⇒ **Lancez les tests et vérifiez qu'ils passent bien tous AU VERT.**

⇒ **Lancez la couverture de code par les tests** et constatez que le code métier est couvert à 100% (Le code de la classe **Simulateur** ne nous interesse pas dans la couverture 😊)

⇒ En local depuis Eclipse, effectuez un nouveau commit en passant dans **Staged Changes** la classe **CanardTest.java** et en nommant votre commit : **ajout CanardTest**



Etape 6 : Permettre à un canard de pouvoir changer de comportement de vol et de cancan durant sa vie de canard ...

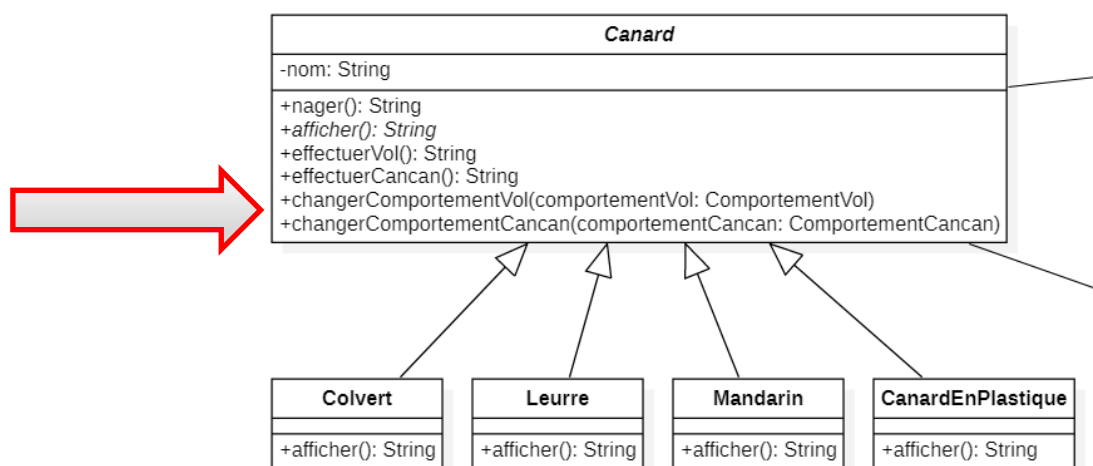
Les besoins initiaux du client mentionnaient les phrases suivantes :

« Et pour finir, sachiez-vous que durant l'hiver, le canard perd ses plumes de vol (rémiges) et ne peut donc pas voler pour un certain temps ?

Du coup, ce serait bien de faire en sorte qu'un canard ait un comportement de vol qui lui est propre et qui pourra être modifier dynamiquement au cours du jeu c-a-d n'importe quand durant le cycle de vie de cet objet. »

C'est justement pour pouvoir modifier le comportement de vol (ou le comportement de cancan) dynamiquement au cours du jeu que le diagramme de classe précédent à été proposé.

Il faut donc mettre en place maintenant cette fonctionnalité, en proposant deux nouvelles méthodes dans la classe Canard : **changerComportementVol** et **changerComportementCancan**, qui ne seront autres que des *setteurs* sur les attributs **comportementVol** et **ComportementCancan**.

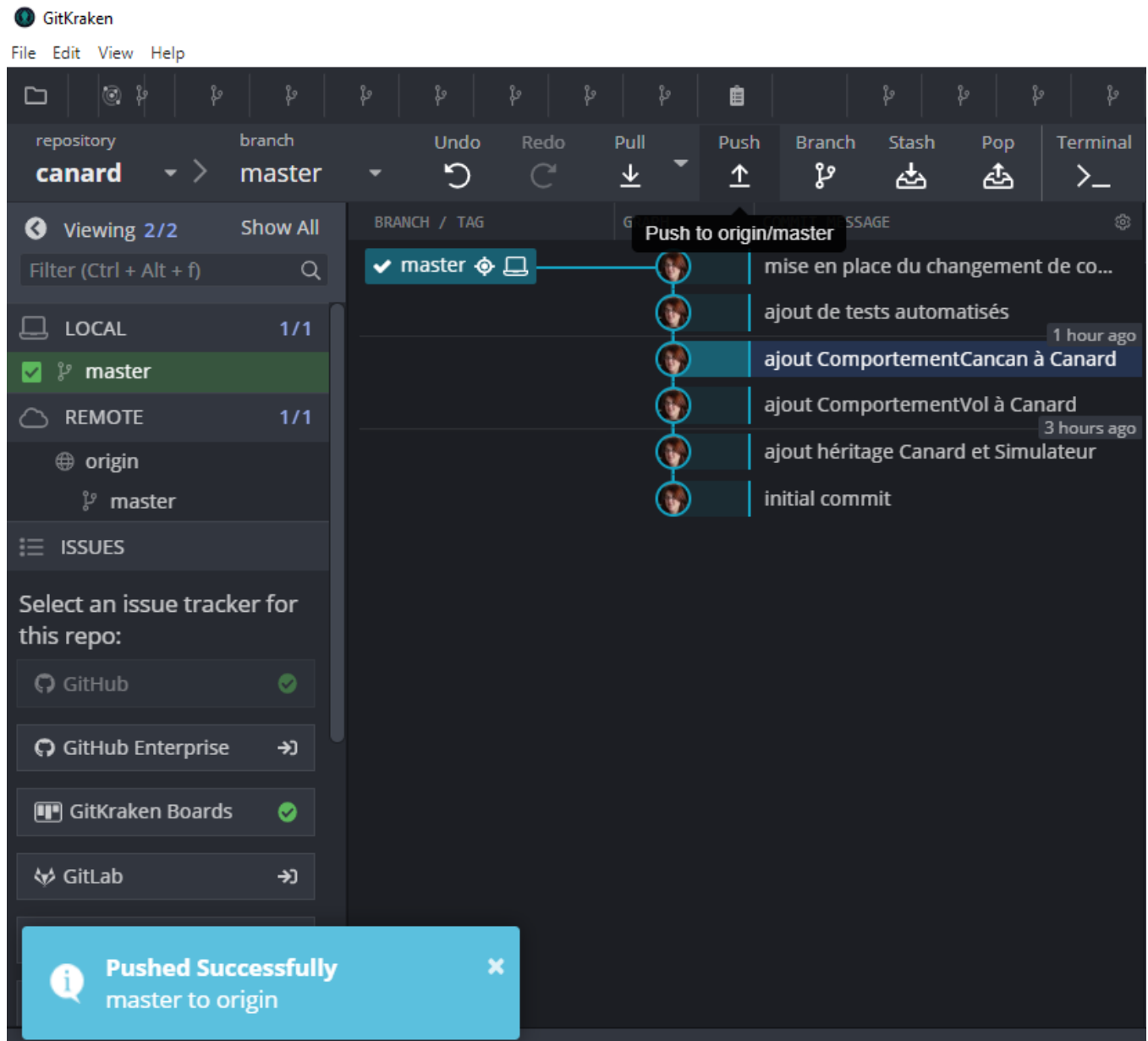


- ➔ Implémentez les méthodes **changerComportementVol** et **changerComportementCancan** dans la classe Canard
- ➔ Vérifiez et validez le « bon » comportement de ces deux nouvelles fonctionnalités en écrivant deux nouveaux tests dans la classe CanardTest.
- ➔ Une fois que ces tests passent, vous pouvez effectuer un nouveau commit que vous appellerez : **maj Canard avec possible changement du comportement de vol et de cancan**

Etape 7 : Pousser à nouveau sur le remote

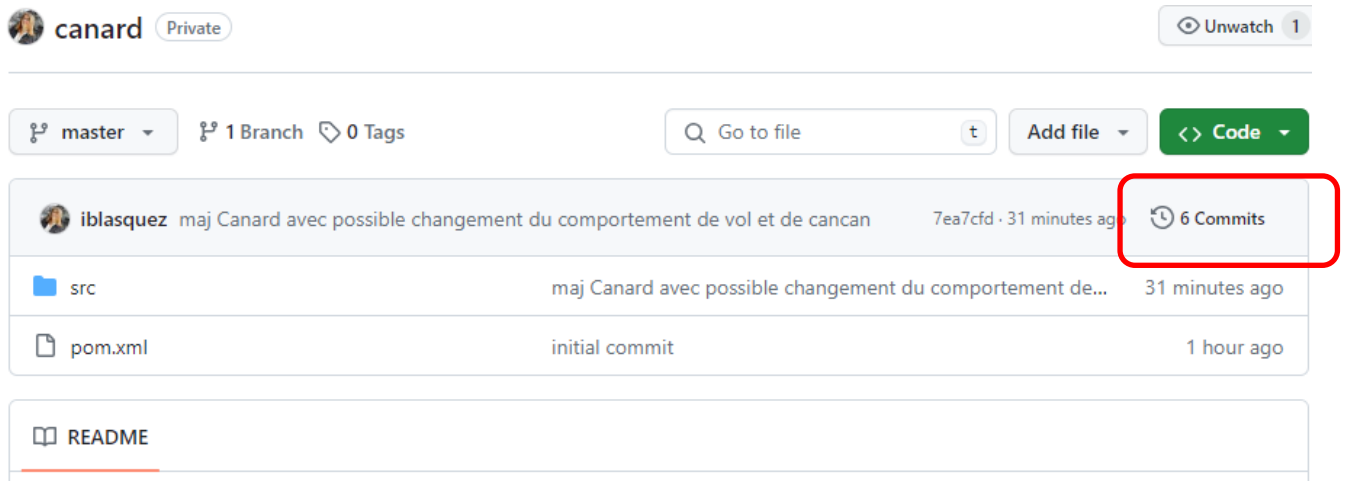
Pour l'instant, votre travail sur le projet canard est terminé : il ne vous reste plus qu'à pousser sur le **remote** votre nouveau code !

⇒ Pour cela, **rendez-vous sur GitKraken**, et utilisez le bouton **Push** et croisez-les doigts pour voir apparaître le message **Pushed Sucessfully** !



Sur GitKraken, vous voyons que nous avons **6 Commits en local**,

⇒ Rendez-vous dans le dépôt canard de votre compte Github pour vérifier que le code a bien été poussé et que vous avez maintenant également **6 commits sur le remote**.



The screenshot shows the GitHub interface for a repository named 'canard' (owner: iblasquez). The repository is private. At the top, there are buttons for 'master' (selected), '1 Branch', and '0 Tags'. There is a search bar 'Go to file', an 'Add file' button, and a green 'Code' button. Below this, the commit history is shown. The most recent commit is titled 'maj Canard avec possible changement du comportement de vol et de cancan' by user 'iblasquez', committed 31 minutes ago. This commit is highlighted with a red box and labeled '6 Commits'. Below the commit list, there are links for 'src' (31 minutes ago) and 'pom.xml' (1 hour ago). At the bottom, there is a link for 'README'.

Cliquez ensuite sur **6 Commits** pour visualiser, sur le **remote**, l'historique des commits avec leur message explicite 😊



The screenshot shows the 'Commits' page for the 'canard' repository. The page title is 'Commits'. Below the title, there is a dropdown menu for 'master'. The page shows a list of commits on 'Mar 19, 2024'. The commits are listed in reverse chronological order:

- maj Canard avec possible changement du comportement de vol et de cancan**
iblasquez committed 33 minutes ago
- ajout CanardTest**
iblasquez committed 37 minutes ago
- ajout ComportementCancan à Canard**
iblasquez committed 1 hour ago
- ajout ComportementVol à Canard**
iblasquez committed 1 hour ago
- ajout héritage Canard et Simulateur**
iblasquez committed 1 hour ago
- initial commit**
iblasquez committed 1 hour ago

Remarque : Quand on travaille à plusieurs sur un même projet, il peut y avoir des conflits à résoudre (comme évoqué en cours) quand le code est poussé sur le remote, c'est pour cela qu'il est utile de resynchroniser de temps en temps le remote avec le code local avec un **pull** ... mais ce sera une autre histoire 😊

Bonnes pratiques pour la gestion de version de votre développement :

- ✓ **Etape 1 : Implémenter** un nouveau apport fonctionnel à votre application (nouveau comportement, nouvelle fonctionnalité, nouvelle classe ou hiérarchie de classes, ...)
- ✓ **Etape 2 : Vérifier et valider** ce nouvel apport fonctionnel à l'aide de jeux d'essais ou de tests automatisés. On dit alors que cet apport fonctionnel est « opérationnel »
- ✓ **Etape 3 : Commiter en local** ce nouveau apport fonctionnel opérationnel avec un **message de commit explicite !!!**

- ✓ **Le matin,** commencez par synchroniser votre projet avec le remote (**pull**).
- ✓ **Le soir,** finissez par envoyer vos modifications sur le remote (**push**).
- ✓ De temps en temps dans la journée quand vous avez fini de développer une étape fonctionnelle, vous pouvez vous synchroniser avec le remote (pull / push)

Pour revoir ce que l'on vient de faire et aller plus loin...

- Une série de vidéos de José Paumard qui explique l'utilisation de git dans Eclipse :
<https://www.youtube.com/watch?v=23GTxe1sQcQ>
- Quelques tutoriels et ressources en ligne à retrouver sur le dépôt :
https://github.com/iblasquez/tuto_git