

IFT232 – Méthodes de conception orientées objet

Laboratoire 2

Enseignant: Mikaël Fortin

Date de remise: indiquée sur [Turnin](#)

Modalités de remise: [Turnin](#)

À réaliser: personne seule ou en équipe de deux

Le but de ce laboratoire est de vous faire pratiquer l'écriture de tests unitaires avec JUnit.

Mise en situation. Lorsque vous écrivez un programme, vous pensez normalement aux résultats que celui-ci devrait produire. Avec les tests unitaires, on écrit les résultats attendus sous forme de petits programmes de test visant chacun une méthode d'une classe. Ces tests peuvent ensuite être exécutés automatiquement afin de vérifier si le programme a le comportement attendu. Les test sont précieux, comme l'énonce la loi de la conservation des tests:

«Un test qui a été détruit sera probablement réécrit plus tard dans des circonstances plus frustrantes»

Code de base. Le code actuel peut être utilisé pour représenter et résoudre des systèmes d'équations linéaires. Pour ce faire, il contient quatre classes:

- **Vecteur** qui sert à conserver les coefficients d'une équation;
- **Matrice** qui regroupe plusieurs **Vecteurs** en un système d'équations pour le résoudre avec l'élimination de Gauss;
- **UtilitairesAlgebre** qui sert à transformer un **String** en équation et vice-versa. Sa méthode **main** contient également quelques exemples d'utilisation des classes **Vecteur** et **Matrice**.
- **TestVecteur**, une première classe de tests visant les méthodes de la classe **Vecteur**.

Vous pouvez exécuter la méthode **main** de la classe **UtilitairesAlgebre** et voir quelques résultats dans la console, ou exécuter les tests de la classe **TestVecteur** et obtenir un rapport, qui devrait vous indiquer que **testEquals** a échoué.

Tâche. Complétez chacune des modifications demandées aux pages suivantes afin d'obtenir les résultats attendus. Le dossier complet contenant votre code final avec toutes les modifications complétées doit être remis par [Turnin](#) sous forme d'un fichier **.zip** nommé **labo2.zip**.

Pointage. Vous pouvez obtenir jusqu'à 10 points répartis ainsi:

- Les modifications 1 et 3 valent 1 points pour l'écriture des tests.
- Les modifications 2, 4, 5 et 6 valent 2 points chacune, 1 point pour l'écriture de tests et 1 point pour les modifications dans le code;
- La modification 7 vaut 2 points bonus, 1 pour l'écriture de tests et 1 pour le code.

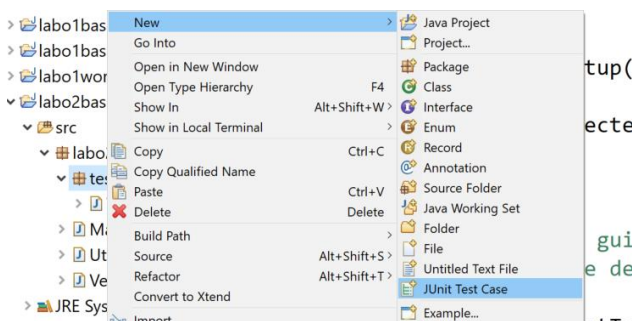
Modification 1. La classe `TestVecteur` contient déjà un test (`testToString`) qui vérifie si l'affichage d'un `Vecteur` produit le bon résultat. Ce test est une traduction directe sous forme de test unitaire du code qui se trouve dans la méthode `main` de la classe `UtilitairesAlgebre`:

The screenshot shows the `main` method of `UtilitairesAlgebre`. Two sections of code are circled in red:

- Top section:** `double[] s111 = { 1, 2, 3, 14 };`, `Vecteur l1 = new Vecteur(s111);`, and `System.out.println(l1);`. A red arrow points from this section to the text: "Code déjà remplacé par la méthode `testToString` de `TestVecteur`".
- Bottom section:** `double[][] systeme1 = { { 3, 5, -3, 15 }, { 7, 10, 1, 2 }, { -3, 2, -5, 6 } };`, `Matrice mat = new Matrice(systeme1);`, `mat.Gauss();`, and `System.out.println(mat);`. A red arrow points from this section to the text: "Code à **remplacer** par la méthode `testToString` de `TestMatrice`".

Remarquez qu'au lieu d'afficher le `Vecteur` dans la console et laisser une personne vérifier si le résultat est correct visuellement, la méthode `testToString` vérifie si la chaîne de caractères produite correspond à une valeur attendue, à l'aide de `assertTrue`, qui signalera que le test a réussi ou échoué.

Créez une classe de test JUnit nommée `TestMatrice` à l'aide du menu contextuel:



Dans cette classe, écrivez une méthode `testToString` qui vérifie que la méthode `toString` de la classe `Matrice` produit bien le résultat attendu. Le code déjà présent dans la méthode `main` de `UtilitairesAlgebre` devrait être adapté à cette fin, de la même façon que pour la méthode `testToString` de la classe `TestVecteur`.

Notez bien.

- Un second test (`testEquals`) échoue parce que la méthode `equals` n'est pas encore correctement implantée dans la classe `Vecteur` (à faire durant la modification 2);
- L'annotation `@Test` doit précéder chaque méthode de test dans les classes `TestVecteur` et `TestMatrice`;
- L'annotation `@Before` est placée devant la méthode qui répétera l'initialisation des variables avant l'exécution de chaque méthode annotée par `@Test`;
- La méthode `assertTrue` signale qu'un test a échoué si l'expression évaluée est fausse.
- La méthode `assertEquals` signale qu'un test a échoué si les deux objets reçus en paramètre sont différents selon la méthode `equals` de la classe du premier objet;
- `assertEquals (obj1, obj2)` est équivalent à `assertTrue(obj1.equals(obj2))`.

Modification 2. La méthode `testEquals` de la classe `VecteurTest` échoue son test parce que vous n'avez pas encore implémenté une méthode `equals` dans la classe `Vecteur`.

Écrivez la méthode `boolean equals(Object other)` dans la classe `Vecteur`, qui vérifiera si les vecteurs ont la même longueur et si chaque élément d'un vecteur correspond aux éléments de l'autre. Une fois la méthode correctement écrite, le test devrait passer.

Étant donné que votre fonction `equals` va comparer des variables de type `double`, vous ne devriez pas utiliser une égalité stricte, mais bien une distance avec une précision. Afin de correctement le faire, vous pouvez ajouter la fonction suivante à votre classe `Vecteur`:

```
public static boolean egaliteDoublePrecision(double a, double b, double epsilon){  
    return (Math.abs(a-b) <= epsilon);  
}
```

La valeur que vous utiliserez pour `epsilon` (la marge de précision) peut être une constante que vous définissez dans votre `Vecteur`.

Ajoutez des vérifications supplémentaires dans la méthode `testEquals` pour des situations où deux vecteurs n'ont pas la même longueur, ou contiennent des éléments différents. Une méthode de test peut contenir plusieurs assertions (vérifications), lorsque celles-ci suivent un thème commun et se regroupent bien. Ici, on teste la méthode

Faites le même travail pour la classe `Matrice`, en ajoutant une méthode `testEquals` dans `TestMatrice` et en écrivant la méthode `equals` de la classe `Matrice`. N'oubliez pas de penser à plusieurs situations à tester.

Notez bien.

- Une méthode de test peut regrouper plusieurs vérifications (assertions);
- On écrit habituellement une méthode de test par méthode de la classe à tester.
- `assertFalse` et `assertNotEquals` doivent être utilisés lorsqu'on s'attend à avoir des résultats faux ou inégaux.
- Il est recommandable d'écrire et exécuter les tests avant d'écrire la méthode qu'on teste.
- Les IDE comme Eclipse et IntelliJ peuvent générer le code de la méthode `equals` d'une classe automatiquement. Vous pouvez utiliser cette fonctionnalité, mais vous devez être capable de comprendre le code généré et le corriger au besoin.

Modification 3. Écrivez des tests dans la classe `MatriceTest` pour la méthode `Gauss`. Le résultat obtenu devrait être une matrice identité carrée avec une colonne de plus qui contient les valeurs des variables.

Exemple. Pour le système d'équation qui est exprimé dans les tests originaux de la méthode `main` de `UtilitairesAlgebre`, on a:

$$3x+5y+-3z=15$$

$$7x+10y+z=2$$

$$-3x+2y-5z=6$$

Sous forme de matrice:

$$\begin{bmatrix} 3 & 5 & -3 & 15 \\ 7 & 10 & 1 & 2 \\ -3 & 2 & -5 & 6 \end{bmatrix}$$

Une fois solutionnée:

$$\begin{bmatrix} 1 & 0 & 0 & 4.15625 \\ 0 & 1 & 0 & -2.25 \\ 0 & 0 & 1 & -4.59375 \end{bmatrix}$$
 Ce qui équivaut à dire: $x=4.15625$, $y = -2.25$, $z= -4.59375$

Sous forme de matrice affichée dans votre programme:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 4.15625 \\ 0.0 & 1.0 & 0.0 & -2.25 \\ 0.0 & 0.0 & 1.0 & -4.59375 \end{bmatrix}$$

Pour créer plusieurs tests pertinents, vous pouvez utiliser un solveur d'équations linéaires en ligne:

http://wims.unice.fr/wims/en_tool~linear~linsolver.en.html

Notez bien.

- Étant donné que vous allez réutiliser plusieurs fois les mêmes objets initialisés de la même façon dans votre classe `MatriceTest`, il devient important d'utiliser l'annotation `@Before` pour alléger le code;
- Maintenant que vous avez une méthode `equals`, il est pertinent de s'en servir pour vos tests.
- Il faut entrer les équations dans la même forme que celle au début de l'exemple pour faire fonctionner le solveur sur le site web.

Modification 4. Ajoutez une méthode `sousVecteur (int taille)` à la classe `Vecteur` et une méthode `sousMatrice (int lignes, int colonnes)` à la classe `Matrice`. Ces méthodes retourneront un nouveau `Vecteur` ou une nouvelle `Matrice` contenant seulement une partie de l'original(e).

Vous devez émettre des exceptions lorsque les dimensions du sous-vecteur ou de la sous-matrice sont inadmissibles (dépassent les dimensions originales ou sont négatives). Pour émettre une exception, vous devez utiliser la clause `throw`, comme ceci:

```
throw new IllegalArgumentException("Dimensions inadmissibles");
```

Dans vos tests, vous devez écrire des cas où des exceptions se produiront. Afin de vérifier si une exception se produit, votre méthode de test doit être annotée en indiquant l'exception qui est prévue:

```
@Test(expected=IllegalArgumentException.class)
```

Exemple. Étant donnée la matrice suivante:

$$\begin{bmatrix} 3 & 5 & -3 & 15 \\ 7 & 10 & 1 & 2 \\ -3 & 2 & -5 & 6 \end{bmatrix}$$

Un appel à `sousMatrice(2,3)` produirait le résultat suivant:

$$\begin{bmatrix} 3 & 5 & -3 \\ 7 & 10 & 1 \end{bmatrix}$$

De même, pour le vecteur suivant:

$$[3 \quad 5 \quad -3 \quad 15]$$

Un appel à `sousVecteur(2)` produirait le résultat suivant:

$$[3 \quad 5]$$

Notez bien.

- Vous ne pouvez tester qu'une seule exception par méthode de test, et celle-ci doit être causée par le dernier énoncé de votre méthode de test;
- Vous pouvez tenter d'utiliser `assertException` pour regrouper vos tests dans la même méthode mais son utilisation n'est pas exigée.
- Vous devez tester au moins une application correcte de chaque méthode (`sousVecteur` et `sousMatrice`);
- Vous devez tester plusieurs cas d'exception pour chaque méthode (`sousVecteur` et `sousMatrice`).

Modification 5. Écrivez des méthodes de classe qui génèrent des vecteurs nuls et de matrices nulles, c'est-à-dire, qui sont initialisés avec des 0. Écrivez également une méthode de classe qui crée une matrice identité. Une matrice identité est une matrice carrée contenant des 0 sauf sur la grande diagonale, où elle contient des 1.

Ces méthodes devraient être accompagnées des tests suivants:

- Un test pour la méthode `creerVecteurNul`;
- Un test pour la méthode `creerMatriceNulle`;
- Un test pour la méthode `creerMatriceIdentite`;
- Un test qui vérifie qu'une matrice à laquelle on a appliqué la méthode `Gauss` contient bien une sous-matrice identité, si on lui retire une colonne.

Une méthode de classe est une méthode précédée du mot-clé `static`. Elle peut être invoquée sans qu'il existe un objet de cette classe. Dans cette situation particulière, l'objectif de ces méthodes est justement de créer un `Vecteur` ou une `Matrice`.

Pour la classe `Vecteur`, vous auriez la méthode:

```
public static Vecteur creerVecteurNul (int taille)
```

Pour l'invoquer, vous pouvez faire:

```
Vecteur vec = Vecteur.creerVecteurNul(3);
```

Pour la classe `Matrice`, vous auriez la méthode:

```
public static Matrice creerMatriceNulle (int lignes, int colonnes)
```

Pour l'invoquer, vous pouvez faire:

```
Matrice mat = Matrice.creerMatriceNulle(3);
```

Pour la classe `Matrice`, vous auriez aussi la méthode:

```
public static Matrice creerMatriceIdentite (int taille)
```

Pour l'invoquer, vous pouvez faire:

```
Matrice mat = Matrice.creerMatriceIdentite(3);
```

Ceci produirait la matrice suivante:

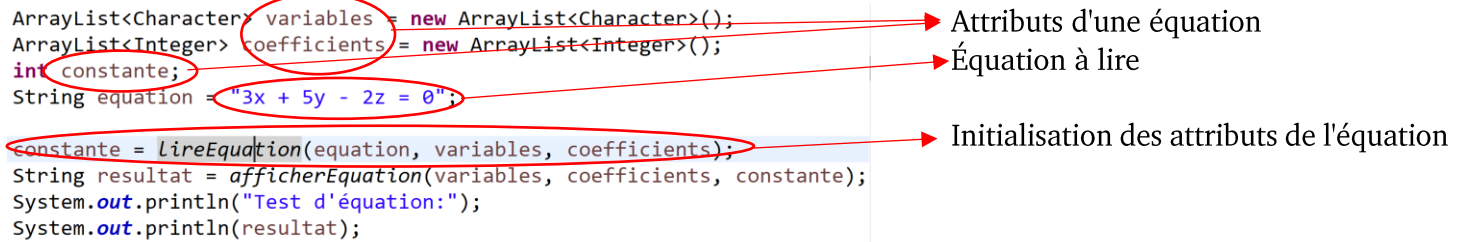
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Notez bien.

- Vous aurez peut-être besoin d'un nouveau constructeur pour réaliser cette tâche. Celui-ci devrait demeurer privé.

Modification 6. Écrivez une classe `Equation` qui servira à remplacer le code des méthodes `lireEquation` et `afficherEquation` de la classe `UtilitairesAlgebre`.

Dans la méthode `main` de `UtilitairesAlgebre`, plusieurs données sont initialisées pour tester la lecture et l'affichage d'une équation:



```

ArrayList<Character> variables = new ArrayList<Character>();
ArrayList<Integer> coefficients = new ArrayList<Integer>();
int constante;
String equation = "3x + 5y - 2z = 0";
constante = lireEquation(equation, variables, coefficients);
String resultat = afficherEquation(variables, coefficients, constante);
System.out.println("Test d'équation:");
System.out.println(resultat);

```

Attributs d'une équation

Équation à lire

Initialisation des attributs de l'équation

Ces données devraient maintenant être les attributs de votre nouvelle classe `Equation` (une équation se définit avec une liste de variable, la liste de leurs coefficients et une constante).

La méthode `afficherEquation` devrait devenir la méthode `String toString()` de votre nouvelle classe `Equation`.

La méthode `lireEquation` devrait devenir la méthode `void lire(String source)` de votre nouvelle classe `Equation`.

Par exemple, l'équation qu'on obtiendrait à partir de `"3x + 5y - 2z = 0"` devrait avoir les attributs suivants:

variables = x, y, z

coefficients = 3, 5, -2

constante = 0

La création et la lecture d'un objet `Equation` prendrait donc la forme suivante:

```
Equation eq = new Equation();
```

```
eq.lire("3x + 5y - 2z = 0");
```

Votre nouveau code devrait être accompagné des tests suivants:

- Une classe `TestEquation` qui regroupera vos tests pour la classe `Equation`;
- Un test correct pour la méthode `lire`;
- Un test correct pour la méthode `toString`;
- Un test pour chaque situation qui peut causer une exception dans la méthode `lire`.

Notez bien.

- Votre classe `Equation` **doit hériter** de la classe `Vecteur`;
- Vous pouvez réaménager la classe `Vecteur` au besoin pour réaliser cette tâche;
- Vos anciens tests doivent encore fonctionner si vous modifiez la classe `Vecteur`.

Modification 7 (bonus). Écrivez une classe `SystemeEquations` qui servira à solutionner des systèmes d'équation, tout comme le fait la fin de la méthode `main` de `UtilitairesAlgebre`.

Votre nouveau code devrait remplir les exigences suivantes:

- La classe `SystemeEquations` hérite de `Matrice`;
- Elle comprend une méthode `lire(String)`;
- Elle comprend une méthode `toString()` qui affiche le système d'équations;
- `toString` doit pouvoir simplifier l'affichage lorsque les coefficients sont à 0.
- Une méthode `equals(Object)`;
- La classe `Equation` aura également besoin d'une méthode `equals(object)`.

Votre nouveau code devrait être accompagné des tests suivants:

- Un classe `TestSystemeEquation` qui regroupera vos tests pour la classe `SystemeEquation`;
- Un test correct pour la méthode `lire`;
- Un test correct pour la méthode `toString`;
- Un test correct et un test incorrect pour la méthode `equals` de `Equation`
- Un test correct et un test incorrect pour la méthode `equals` de `SystemeEquation`

Exemple. Pour le système d'équations suivant sous forme de `String`:

$$\begin{aligned} 3x + 5y - 3z &= 15 \\ 7x + 10y + 1z &= 2 \\ -3x + 2y - 5z &= 6 \end{aligned}$$

On devrait obtenir, après une application de la méthode Gauss, l'affichage suivant:

$$\begin{aligned} x &= 4.15625 \\ y &= -2.25 \\ z &= -4.59375 \end{aligned}$$

Notez bien.

- Il faut sérieusement considérer remplacer les tableaux de double par autre chose dans les classe `Vecteur` et `Matrice` pour réaliser cette modification;
- La méthode `coefficientAtableau` est un symptôme d'une mauvaise organisation du code, vous devriez oeuvrer à la faire disparaître;
- Tous vos anciens tests doivent encore fonctionner après que vous ayez réaménagé le code.