# Feedback labo 5

NOVEMBRE 2022

R. RENTSCH

# Rendus des laboratoires



Sous la forme d'un rapport pdf, contenant :

- Page de titre
- Diagramme de classes UML
- Liste des tests réalisés
- Listings formatés correctement
  - Mots-clés en gras
  - /!\ lignes de code trop longues !
  - Documents disponibles sur Cyberlearn
    - Installation et configuration de Notepad++.pdf
    - Créer un document pdf incluant des listings.pdf

# Documentation du code



- Munir chaque classe / méthode / champ d'un en-tête **Javadoc**
- Commenter le code des méthodes

# Documentation du code



```
/**
 * Cette classe ...
 * @author R. Rentsch, ...
 * @version 1.0
 */
public class Rectangle {

   /**
    * Constructeur
    * @param longueur la longueur du rectangle
    * @param largeur la largeur du rectangle
    * @throws IllegalArgumentException lev&eacute;e si longueur ou largeur &leq; 0
    */
   public Rectangle(double longueur, double largeur) { // pas de throws ici
      if (longueur <= 0 || largeur <= 0) {
         throw new IllegalArgumentException("Parametres incorrects !");
      }
      this.longueur = longueur;
      this.largeur = largeur;
   }

   /**
    * longueur du rectangle
    */
   private final double longueur;

   /**
    * largeur du rectangle
    */
   private final double largeur;
}
```

Javadoc

# Respect de la donnée



- *"Ecrire un programme de test prenant **en argument** les tailles N1, M1, N2, M2 de deux matrices ainsi qu'un modulo n et effectuant les opérations sur une matrice N1 × M1 et N2 × M2 de manière à produire un résultat semblable à:"*

- **=> paramètres passés en ligne de commande !**

# Packages



- Ne pas mettre la classe contenant le programme principal (*main*) dans le même package que les autres classes

  ... sinon il y a risque de ne pas détecter des erreurs d'accessibilité latentes 

# Tests / JUnit



- Séparer les tests du *main* (2 fichiers distincts)
- Inutile de faire des "tonnes de tests" (parfois redondants)

- … mais faire des **tests pertinents** couvrant les différents cas pouvant se présenter
  - Exemples :
    matrice vide, matrice irrégulière, matrice contenant des valeurs non conformes au modulo, matrices avec modulo différents, ...

- Si vous faites des tests unitaires
  - Choisir des noms parlants pour les divers tests
  - Indiquer quelle **version de JUnit** est utilisée
    N.B. Pas nécessaire si projet Maven car info intégrée dans le fichier pom.xml

# Violation potentielle de l'encapsulation



- Exemple

# Terminologie utilisée



- Attention à utiliser une terminologie adéquate 
  - nbLignes / nbColonnes d'une matrice et pas hauteur / largeur d'une matrice

# Tirage nombres aléatoires modulo n (càd dans l'intervalle [0; n -1])



- En utilisant la classe **java.lang.Math**
  - tab2d[i][j] = (**int**)(Math.*random*() * modulo);

- En utilisant la classe **java.util.Random**
  - tab2d[i][j] = *RAND*.nextInt(modulo);
  - /!\ RAND ne doit pas être déclaré dans le constructeur mais déclaré en tant que champ static final de la classe
    ... sinon générateur aléatoire recréer à chaque nouvel appel du constructeur

    **private static final** Random *RAND* = **new** Random();

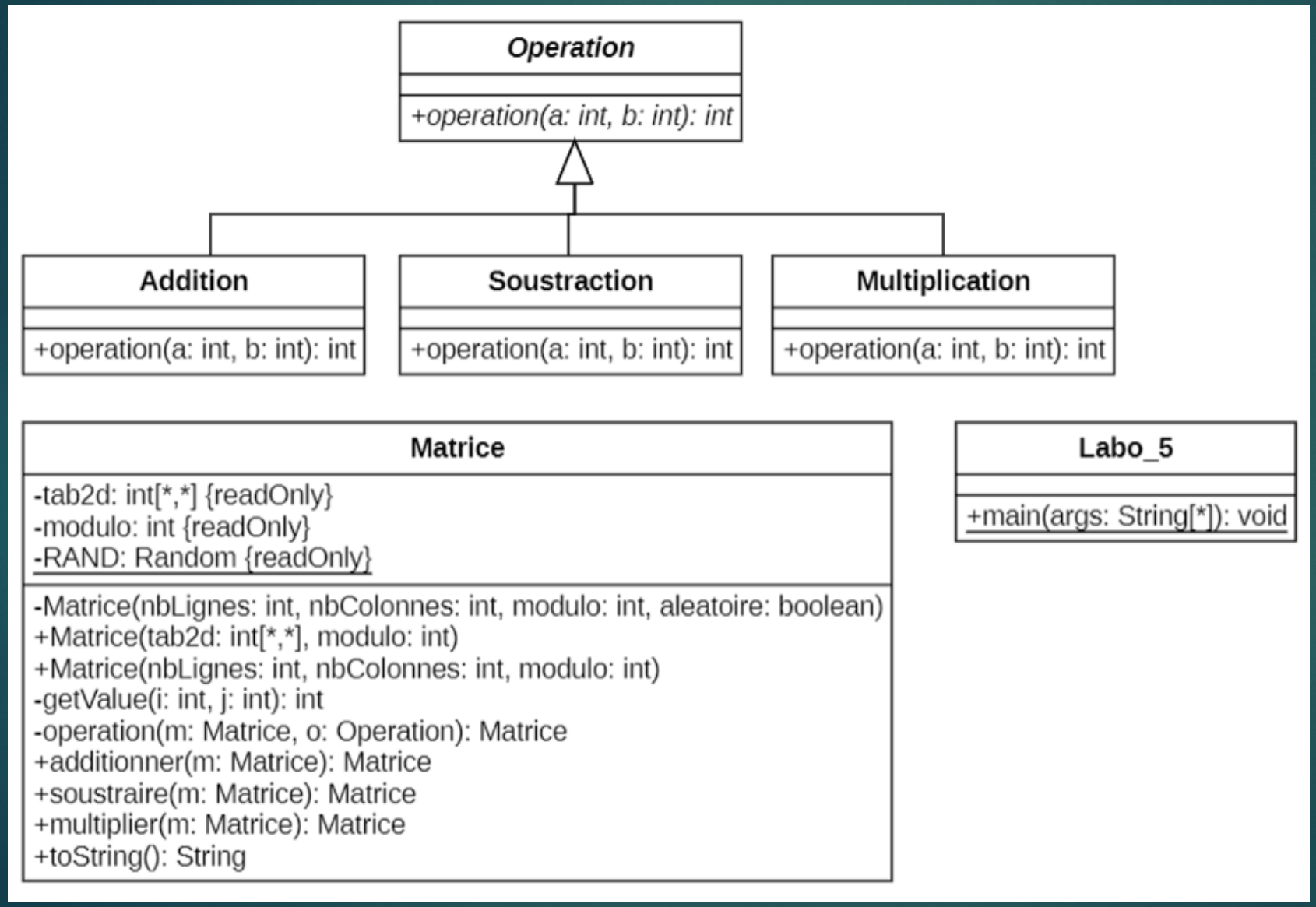
# Factorisation des constructeurs



```
private Matrice(int nbLignes, int nbColonnes, int modulo, boolean aleatoire) {

   // check de validité de modulo. Si pas ok, lever une RuntimeException
   // check de validité de nbLignes et de nbColonnes. Si pas ok, lever une RuntimeException

   this.modulo = modulo;
   tab2d = new int[nbLignes][nbColonnes];

   if (aleatoire) {
      // remplir this.tab2d avec des valeurs aléatoires comprise entre 0 et modulo - 1
   }
}

public Matrice(int nbLignes, int nbColonnes, int modulo) {
   this(nbLignes, nbColonnes, modulo, true);
}

public Matrice(int[][] tab2d, int modulo) {

   this(tab2d.length, tab2d.length > 0 ? tab2d[0].length : 0, modulo, false);

   // parcourir le tableau passé en paramètre et vérifier
   // 1) que toutes les lignes du tableau ont la même taille
   // 2) que tous les éléments du tableau admettent une valeur comprise entre 0 et modulo-1
   // Si pas ok, lever une RuntimeException, sinon faire this.tab2d[i][j] = tab2d[i][j];
}
```

# Factorisation des opérations (+, - , *)





# Factorisation des opérations (+, - , *)



- A relever :
  - Addition renvoie a + b
  - Soustraction renvoie a - b
  - Multiplication renvoie a * b

- La présence, dans la classe Matrice, des méthodes (non static !)
  - -operation(m : Matrice, o : Operation) : Matrice
  - +additionner(m : Matrice) : Matrice
  - +soustraire(m : Matrice) : Matrice
  - +multiplier(m : Matrice) : Matrice

# Factorisation des opérations (+, - , *)



- private Matrice operation(Matrice m, Operation o) {
  // check mêmes modulo. Si pas ok, lever RuntimeException
  Matrice résultat = ...
  Pour chaque élément de résultat
  résultat.tab2[i][j] =
  **Math.floorMod**(o.operation(this.get(i, j), m.get(i, j)), modulo);
  return résultat;
  }

- public Matrice additionner (Matrice m) {
  return operation(m, new Addition());
  }

- Permettra à celui qui écrit le *main*, d'écrire, par ex:

  m1.additionner(m2);

  ... ce qui est beaucoup plus naturel que :

  Matrice.operation(m1, m2, new Addition());