INF-111

Travail pratique #1

Groupe : 3-4 étudiants (une seule remise par équipe).

Remise : 16 mars

Auteur : Frédéric Simard

Révision : Pierre Bélisle

---

1 - Introduction

1.1 - Contexte académique

Ce premier devoir vous amène dans le monde des systèmes de base de données relationnelles[1], mais il vise surtout à vous aider à maîtriser les sujets suivants:

- La programmation Orienté-Objet appliquée
- Les types de données abstraits (pile, file et liste)
- L'héritage et la composition
- Les collections
- Le tri et la fouille
- Les classes abstraites et les interfaces
- L'architecture d'un programme

Ce devoir est la première partie de deux. Cette première partie s'intéresse principalement au développement du système de base de données. La seconde partie, le devoir 2, s'attaque à l'interface utilisateur. En développement logiciel, ces deux segments sont appelés respectivement dorsal (*back-end*) et frontal (*front-end*).

Pour ce devoir, deux concepts tirés du génie logiciel ont été intégrés dans le devoir. Vous n'avez pas à maîtriser ces concepts à 100%, mais une compréhension de base sera requise pour compléter le travail. C'est deux concepts sont : l'indexation et le salage. Ils sont expliqués aux annexes A et B respectivement. L'énoncé vous dirige vers ceux-ci au moment opportun.

1.2 - Description du problème

On vous demande de développer la preuve de concept d'une application de gestions de comptes bancaires grand public. Les fonctionnalités utilisateurs demandées sont limitées, mais le programme développé devra permettre une extension de l'application, qui s'adapte aux besoins futurs.

---

[1] https://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es

Les éléments à réaliser:

- Une structure de base de données qui permet la recherche rapide d'éléments
- Un système de sauvegarde de mot de passe sécurisé
- Une interface de développement d'application (API) qui permettra à l'interface graphique de s'attacher à votre base de données
- Une validation extensive de votre programme

Au final, votre programme permettra la gestion des utilisateurs du système de gestion bancaire et des transactions bancaires effectuées entre ceux-ci. Dans le cadre de ce devoir, le système ne permettra que la simulation des fonctionnalités, mais le prochain devoir permettra d'opérer comme si vous étiez un utilisateur vous-même.

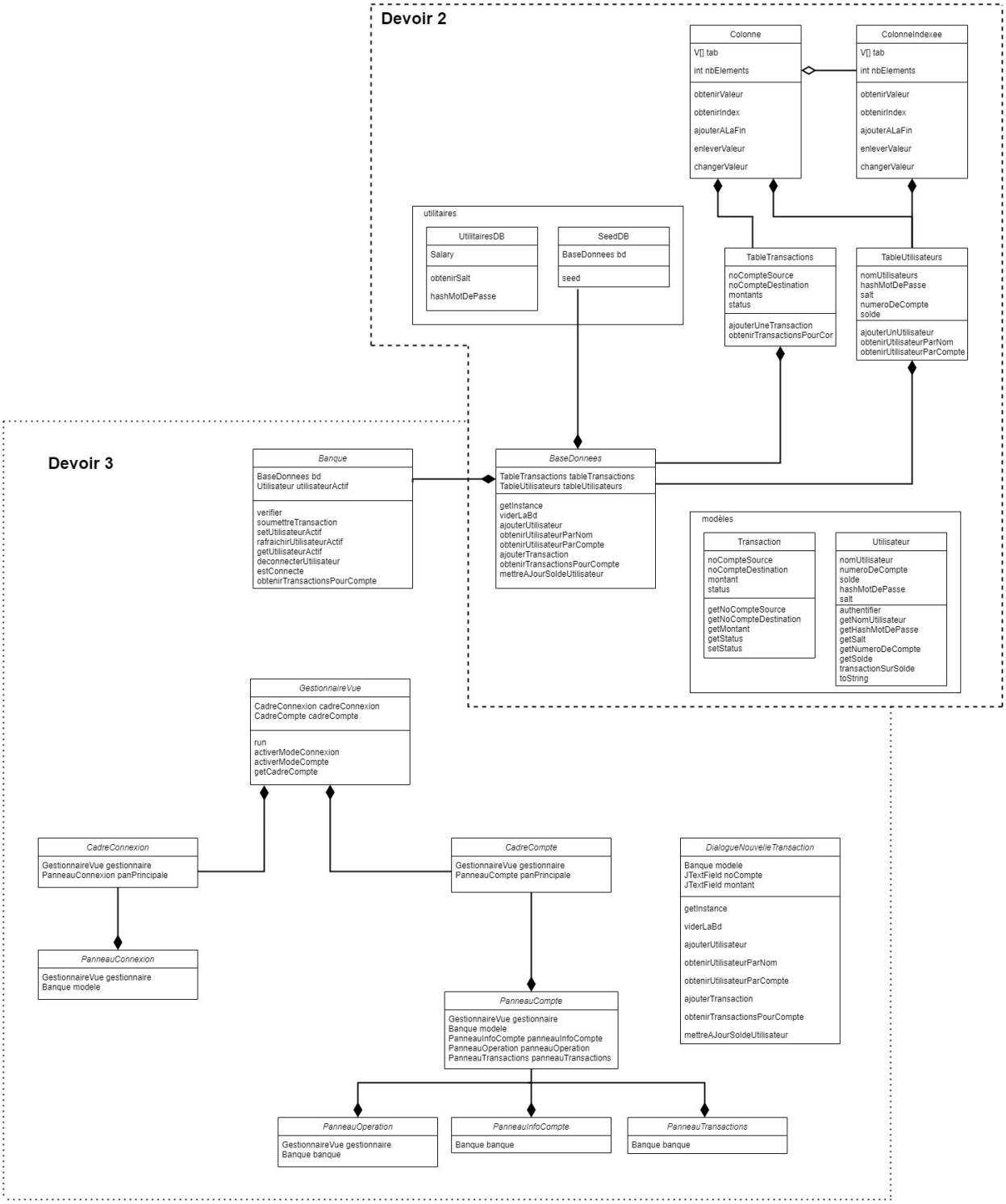
Prenez note qu'il s'agit d'un exemple de programme développé dans le cadre académique INF111. La manière dont la base de données est réalisée est inspirée des véritables systèmes de base de données, mais ne présente pas un cas réel d'implémentation.

1.3 - Réalisation du travail

Ce document vous guide dans le développement du projet d'une manière ascendante, les composantes de base sont développées puis assemblées. Chaque section à développer vient avec une série de tests à réaliser. Les tests sont évalués.

1.4 - Survol du projet

Le schéma qui suit vous présente le projet dans son ensemble. Bien que l'ensemble des éléments du programme ne puisse prendre tout son sens qu'au fil du développement, voici une description sommaire des éléments qui habitent ce programme.


Devoir 2
Colonne
V[] tab
int nbElements
obtenirValeur
obtenirIndex
ajouterALaFin
enleverValeur
changerValeur
ColonneIndexee
V[] tab
int nbElements
obtenirValeur
obtenirIndex
ajouterALaFin
enleverValeur
changerValeur
utilitaires
UtilitairesDB
Salary
obtenirSalt
hashMotDePasse
SeedDB
BaseDonnees bd
seed
TableTransactions
noCompteSource
noCompteDestination
montants
status
ajouterUneTransaction
obtenirTransactionsPourCor
TableUtilisateurs
nomUtilisateurs
hashMotDePasse
salt
numeroDeCompte
solde
ajouterUnUtilisateur
obtenirUtilisateurParNom
obtenirUtilisateurParCompte
Devoir 3
Banque
BaseDonnees bd
Utilisateur utilisateurActif
verifier
soumettreTransaction
setUtilisateurActif
rafraichirUtilisateurActif
getUtilisateurActif
deconnecterUtilisateur
estConnecte
obtenirTransactionsPourCompte
BaseDonnees
TableTransactions tableTransactions
TableUtilisateurs tableUtilisateurs
getInstance
viderLaBd
ajouterUtilisateur
obtenirUtilisateurParNom
obtenirUtilisateurParCompte
ajouterTransaction
obtenirTransactionsPourCompte
mettreAJourSoldeUtilisateur
modèles
Transaction
noCompteSource
noCompteDestination
montant
status
getNoCompteSource
getNoCompteDestination
getMontant
getStatus
setStatus
Utilisateur
nomUtilisateur
numeroDeCompte
solde
hashMotDePasse
salt
authentifier
getNomUtilisateur
getHashMotDePasse
getSalt
getNumeroDeCompte
getSolde
transactionSurSolde
toString
GestionnaireVue
CadreConnexion cadreConnexion
CadreCompte cadreCompte
run
activerModeConnexion
activerModeCompte
getCadreCompte
CadreConnexion
GestionnaireVue gestionnaire
PanneauConnexion panPrincipale
CadreCompte
GestionnaireVue gestionnaire
PanneauCompte panPrincipale
DialogueNouvelleTransaction
Banque modele
JTextField noCompte
JTextField montant
getInstance
viderLaBd
ajouterUtilisateur
obtenirUtilisateurParNom
obtenirUtilisateurParCompte
ajouterTransaction
obtenirTransactionsPourCompte
mettreAJourSoldeUtilisateur
PanneauConnexion
GestionnaireVue gestionnaire
Banque modele
PanneauCompte
GestionnaireVue gestionnaire
Banque modele
PanneauInfoCompte panneauInfoCompte
PanneauOperation panneauOperation
PanneauTransactions panneauTransactions
PanneauOperation
GestionnaireVue gestionnaire
Banque banque
PanneauInfoCompte
Banque banque
PanneauTransactions
Banque banque

Colonne (Section 2.2)

La colonne est l'élément de base dans la construction d'une base de données. Les colonnes sont assemblées en table et les tables forment une base de données.

Chaque entrée dans la base de données occupe une ligne (un enregistrement) et chaque élément de l'entrée est distribué au travers des colonnes (les attributs). Donc un attribut équivaut à une colonne et une instance à une ligne.

ColonneIndexee (Section 2.3)

La colonne indexée sert la même fonction que la colonne, mais ajoute un mécanisme d'index[2] qui permet d'effectuer des recherches de manière beaucoup plus rapide, au coût d'un processus d'insertion un peu plus complexe.

Modèles (Sections 3.1 et 3.2)

Les modèles **Transaction** et **Utilisateur** sont des objets qui représentent des entrées dans la base de données. Les modèles eux-mêmes ne se retrouvent pas dans la base de données, mais servent de contenant à données à l'extérieur de celle-ci.

Tables (Sections 4.1, 4.2 et 4.3)

Les tables gardent les entrées dans la base de données en mémoire. Dans notre projet, il y a deux tables: **TableUtilisateurs** et **TableTransactions**, qui stockent les données relatives aux utilisateurs et aux transactions respectivement.

BaseDonnees (Section 5.1 et 5.2)

La base de données regroupe finalement les tables et expose les accès requis par la banque.

Banque

La banque sera développée au devoir 2.

1.5 - Fichiers fournis

Des fichiers de départ vous sont fournis. Voici la liste et une description de la manière des utiliser.

Utilitaires

Ce fichier contient les méthodes publiques permettant :

- D'obtenir un *salt* (voir annexe B et section 3.1.2)
- D'obtenir le hash d'un mot de passe et d'un *salt* (voir annexe B et section 3.1.2)

[2] https://fr.wikipedia.org/wiki/Index_(base_de_donn%C3%A9es)

InterfaceColonne

Ce fichier contient une interface qui déclare les services publics d'une colonne, elle doit être utilisé pour développer la colonne simple.

ColonneIndexee

Le point de départ de la colonne indexée vous est fourni. Elle contient une classe interne ValIndexee dont l'utilité est décrite en Annexe A.

1.6 – Instruction supplémentaire

Voici une instruction supplémentaire importante.

ColonneIndexee

La colonne indexée est le défi du devoir. Elle vous demandera beaucoup d'effort.

Dans ce contexte, sachez que tout le travail peut se réaliser sans l'implémentation de la colonne indexée, à l'exception du test d'unicité à la section 4.1.2, qui doit être contournée. De plus, l'intégration de la colonne indexée *a posteriori* ne requiert que de changer les types des colonnes dans les tables, puisque les deux colonnes partagent la même interface publique, et d'ajouter le test d'unicité. En d'autres mots : intégrer la colonne indexée à la toute fin est facile (grâce au pouvoir du polymorphisme).

Cela veut aussi dire que l'implémentation de la colonne indexée n'est pas un élément bloquant dans la réalisation du devoir. Toutefois, Une partie importante des points est associée avec cette section.

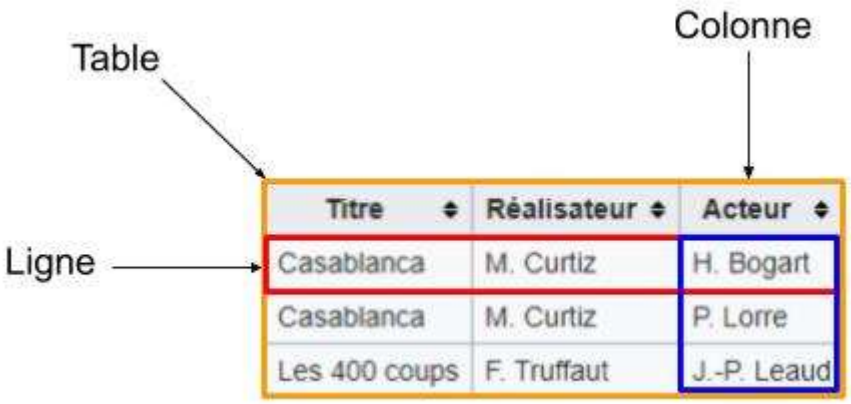
La stratégie de développement suggéré est donc la suivante :

- Suivez l'ordre du développement du devoir et tentez de réaliser la colonne indexée à la section 2.3.
- Si vous bloquez dans l'implémentation de la colonne indexée, mettez là de côté temporairement, puis continuez à développer les autres sections.
- Avant la remise, prévoyez le temps de terminer la colonne indexée et de l'intégrer.

Section 2 - Colonnes (semaine 1 et 2)

Section 2.1 – Description (semaine 1)

Dans une base de données relationnelle[3], les données sont classées par table et les tables sont composées de colonnes. Les attributs d'une entrée sont disposés en ligne, mais seules les colonnes sont des éléments structurant du programme de la table.



Pour notre projet, nous développerons deux types de colonnes: les colonnes et les colonnes indexées. La première rappellera les notions enseignées relatives aux listes statiques, mais l'implémentation sera adaptée à la situation actuelle. La seconde est développée avec un objectif bien précis, offrir une méthode de recherche rapide. Il s'agira de combiner les notions de liste, tri et fouille en un seul objet.

Au-delà des tests usuels des fonctionnalités développés, cette section demande le développement d'un test d'évaluation des performances des deux colonnes pour la recherche d'éléments.

Section 2.2 – Colonne (semaine 1)

Description

La colonne est un type de données abstrait implémentée selon l'approche statique. L'implémentation que vous faites doit utiliser une classe générique (voir section Generic Classes au bas de la page: https://www.tutorialspoint.com/java/java_generics.htm), la description utilise V, pour référencer la classe générique.

[3] https://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es_relationnelle

Services

Elle offre les services suivants:

- ajouterValeur(**V** valeur), cette méthode ajoute une valeur à la fin de la colonne.
- obtenirValeur(int index), cette méthode retourne la valeur à l'index reçu.
- obtenirIndex(**V** valeur), cette méthode retourne l'index de la première occurrence de la valeur reçue (valeur égale (equals), pas référence égale (==)).
- changerValeur(int index, **V** valeur), cette méthode remplace la valeur stockée à l'index par celle reçue.
- getNbElements(), retourne le nombre d'éléments contenus dans la colonne.
- afficherContenu(), cette méthode affiche le contenu de la colonne. Cette méthode est utilisée pour le déverminage.

Contraintes

De plus,

- La colonne doit ajuster automatiquement la taille du tableau sous-jacent, si le nombre d'éléments insérés dépasse la capacité initiale.
- Toutes les méthodes qui reçoivent un index doivent vérifier que l'index est valide et retourner une exception si ce n'est pas le cas.

Validation

Écrivez des tests qui démontrent que chacune des méthodes décrites plus haut fonctionne bien. Pour simplifier la validation, nous vous suggérons d'utiliser la classe **Integer** comme classe générique **V**. Cela devrait vous simplifier la vie.

Section 2.3 – ColonneIndexee (semaine 1 et 2)

Description

La colonne indexée est une classe dérivée de la Colonne, mais elle intègre un mécanisme d'indexation, expliqué en Annexe A, qui permet d'extraire des valeurs plus rapidement.

Services

Elle redéfinit (override) les services suivants:

- ajouterValeur(**V** valeur)
- obtenirIndex(**V** valeur)
- estUnique(**V** valeur), cette méthode détermine si la valeur reçue est déjà dans le tableau. Retourne *false*, si elle est présente, et *true*, si absente, parce qu'elle répond à la question: est-ce que la valeur est unique?
- changerValeur(int index, **V** valeur), cette méthode n'est pas supportée pour la colonneIndexee, redéfinir avec une définition vide et qui retourne une exception lors de l'appel.

- afficherContenu(), implémentation suggérée, redéfinir pour afficher les paires index et valeurs, de même que les valeurs de la colonne normale.

Contraintes

De plus,

- La colonne doit ajuster automatiquement la taille du tableau sous-jacent, si le nombre d'éléments insérés dépasse la capacité initiale.
- Toutes les méthodes qui reçoivent un index doivent vérifier que l'index est valide et retourner une exception si ce n'est pas le cas.

Validation

Écrivez des tests qui démontrent que chacune des méthodes décrites plus haut fonctionnent bien. Pour simplifier la validation, nous vous suggérons d'utiliser la classe **Integer** en place de la classe générique. Cela devrait vous simplifier la vie.

Profilage des performances

Écrivez une procédure de test qui évaluent les performances de votre colonne indexées versus la colonne simple. Voici quelques éléments pour vous aider dans la conception :

- Inspirez-vous de la procédure de test de l'Annexe A.
- Mesurez un grand nombre d'essais, puis calculez la moyenne (ex : 10000).
- À chaque essai, remplissez la colonne d'éléments aléatoires et positionnez l'élément témoin aléatoirement.
- Insérez un grand nombre d'élément dans la colonne (ex : 10000)
- Ne mesurez que le temps requis pour trouver l'élément. En d'autres mots, n'incluez pas le temps requis pour remplir la colonne.
- Appelez **obtenirIndex**, plusieurs fois dans le même test (5 fois). Ce n'est pas grave si elle retourne toujours le même résultat, puisque la recherche doit quand même se faire.
- Voici comment mesurer un différentiel de temps :

```
long debut = System.currentTimeMillis();
…
tempsExect[i] = (System.currentTimeMillis() - debut);
```

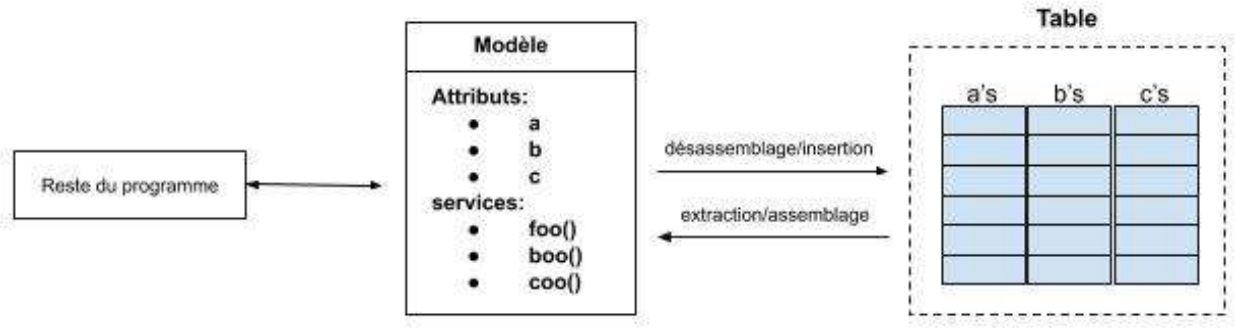
La colonne indexée devrait dépasser les performances de la colonne simple d'une manière très importante. Si vous n'y arrivez pas, réviser votre algorithme de fouille et tenter d'identifier le problème. Sinon, vous pouvez demander de l'aide, pour valider votre implémentation.

Section 3 – Modèles (semaine 2)

Comme nous l'avons vu dans la section 2, les attributs des entrées dans la base de données sont chacun stockés dans leur propre colonne. Bien que cette mécanique offre un grand nombre d'avantages pour le fonctionnement, la maintenance et les performances de la base de données, elle brise l'association directe entre les attributs.

NOTE : il existe d'autres types de base de données qui ne brisent pas les éléments lors de leur entrée, mais elles comportent leurs propres désavantages. Il s'agit toujours d'une question de compromis.

C'est pour cela que les logiciels qui utilisent des bases de données relationnelles implémentent généralement une interface pour le traitement de « modèles » de données. Pour les besoins de notre discussion, les modèles sont tout simplement des classes qui contiennent les attributs et offrent des services autour de ceux-ci.



Notre programme utilise deux modèles : les Utilisateurs et les Transactions. Voici leur description.

3.1 Modèle : Utilisateur

La classe **Utilisateur** représente un des utilisateurs de notre système bancaire. Voici une description des éléments de la classe. Le fonctionnement du *salt* et du *hash* du mot de passe sont décris à l'Annexe B.

3.1.1 Attributs

- Nom de l'utilisateur (String)
- Numéro de compte (String)
- Solde (double)
- *Salt* (byte[])
- *Hash* du mot de passe (byte[])

3.1.2 Services

- **Utilisateur(String nomUtilisateur, String motDePasse,**

String numeroDeCompte, double solde)

Ce constructeur initialise les attributs avec les valeurs reçues à l'exception du *salt* et du *hash* du mot de passe. Pour initialiser ces derniers, utiliser les méthodes correspondantes dans les utilitaires (lisez Annexe B, si ce n'est pas déjà fait, pour comprendre le Salage des mots de passes).

- **Utilisateur(String nomUtilisateur, byte[] hashMotDePasse,**
  byte[] salt, String numeroDeCompte, double solde)

Ce constructeur initialise les attributs avec les valeurs reçues.

- **authentifier(String nomUtilisateur, String motDePasse)**

Cette méthode retourne un booléen indiquant si le nom d'utilisateur et le mot de passe reçus en paramètres correspondent bien à celui du modèle. Le nom de l'utilisateur peut être comparé directement, mais le mot de passe doit d'abord passer par le salage, avant d'être comparable (Annexe B). Vous devez ensuite comparer le contenu des deux tableaux de bytes. Nous vous suggérons une recherche Google (ou ChatGPT) sur le sujet, puisqu'une solution toute simple existe.

- **Accesseurs informateur pour :**
  - Nom de l'utilisateur
  - Numéro de compte
  - Solde
  - *Hash* du mot de passe
  - *Salt*

- **transactionSurSolde(double differentiel)**

Ajoute le différentiel au solde.

- **toString()**

Retourne une représentation de l'utilisateur sous forme de chaîne de caractères. Cette méthode est principalement utilisée pour le déverminage.

3.2 Modèle : Transaction

La classe **Transaction** représente une transaction bancaire entre deux utilisateurs. Voici une description des éléments de la classe.

3.2.1 Constantes publiques

String ACCEPTE = "Accepte"
String REFUSE = "Refuse"
String A_DETERMINER = "A determiner"

3.2.2 Attributs

- Numéro de compte source (String)
- Numéro de compte destination (String)
- Montant (double)
- Statut (String)

3.2.3 Services

- **Transaction(String noCompteSource, String noCompteDestination, double montant, String statut)**

Ce constructeur initialise les attributs avec les valeurs reçues.

- **Transaction(String noCompteSource, String noCompteDestination, double montant)**

Ce constructeur initialise les attributs avec les valeurs reçues, à l'exception de statut qui est initialisé à l'aide de la constante A_DETERMINER.

- **Accesseurs informateur pour :**
  - Numéro du compte source
  - Numéro du compte destination
  - Montant
  - Statut

- **Accesseurs mutateur pour :**
  - Statut

3.3 Validation

Vérifier que vos modèles peuvent être instanciés et que tous les services fonctionnent bien. Vous devez prendre le temps de vérifier que vos constructeurs sont compatibles avec l'utilisation faite dans le fichier **baseDonnees.utils::SeedDB**.

Section 4 – Tables (semaine 3)

Les tables rassemblent les colonnes en une unité capable de stocker et rappeler un modèle. On retrouve donc une table par modèle : TableUtilisateurs et TableTransactions.

4.1 TableUtilisateurs

4.1.1 Attributs

La table contient une colonne pour chacun des attributs de Utilisateur (Section 3.1).

Les colonnes contenant le nom de l'utilisateurs et le numéro de compte sont des colonnes indexées. Les autres sont des colonnes simples.

4.1.2 Services

- **TableUtilisateurs()**

Ce constructeur initialise les colonnes.

- **ajouterUnUtilisateur (Utilisateur utilisateur)**

Cette méthode ajoute un utilisateur à la base de données, mais seulement si cet utilisateur est unique. Voici les étapes :

  - Vérifie que le nom et le numéro de compte de l'utilisateur sont uniques (colonnes indexées seulement)
    - Sinon, retourne *false*
  - Ajoute chacun des attributs de l'instance utilisateur à la table correspondante.
  - Retourne *true*

- **obtenirUtilisateurParNom (String nomUtilisateur)**

Cette méthode retrouve un utilisateur à partir de son nom. Voici la procédure :

  - Utilise la méthode permettant d'obtenir l'index de la ligne de cet utilisateur
  - Utilise l'index pour retrouver les valeurs des attributs de **Utilisateur**
  - Construit puis retourne une instance **Utilisateur**, initialisée avec les valeurs récupérées

Note :

  - Retourne *null*, si nomUtilisateur non-présent ou si une exception est levée.

- **obtenirUtilisateurParCompte (String numeroDeCompte)**

Cette méthode retrouve un utilisateur à partir de son numéro de compte. Se comporte comme la méthode précédente, mais trouve l'index avec le numeroDeCompte.

- **mettreAJourSoldeUtilisateur(Utilisateur utilisateur, double solde)**

Cette méthode retrouve un utilisateur puis change son solde. À vous de déterminer comment l'implémenter.

4.1.3 Validation

Écrivez des tests qui valident votre interface.

4.2 TableTransactions

4.2.1 Attributs

La table contient une colonne pour chacun des attributs de Transaction.

Toutes les colonnes sont des colonnes simples.

4.2.2 Services

- **TableTransactions()**

Ce constructeur initialise les colonnes.

- **ajouterUneTransaction(Transaction transaction)**

Cette méthode insère chacun des attributs de la transaction dans la base de données.

- **obtenirTransactionsPourCompte(String numeroDeCompte)**

Cette méthode récupère toutes les transactions associées avec le numéro de compte. Elle récupère à la fois les transactions dont le compte est source ou destination.

Toutes les transactions sont ajoutées à une collection java de votre choix, puis retournées.

S'il n'y a aucune transaction, une collection vide est retournée.

4.2.3 Validation

Écrivez des tests qui valident votre interface.

Section 5 – BaseDonnees (semaine 3)

La base de données est une composition de tables. Elle expose une interface de programmation. Voici les méthodes à présenter :

- **BaseDonnees (),** Initialise les tables.
- **viderLaBd (),** Réinitialise les tables.
- **ajouterUtilisateur (Utilisateur nouvelUtilisateur)**
- **obtenirUtilisateurParNom(String nomUtilisateur)**
- **obtenirUtilisateurPourCompte(String numeroDeCompte)**
- **mettreAJourSoldeUtilisateur(Utilisateur utilisateur, double solde)**
- **ajouterTransaction(Transaction transaction)**
- **obtenirTransactionsPourCompte(String numeroDeCompte)**

Annexe A - Mécanisme d'indexation

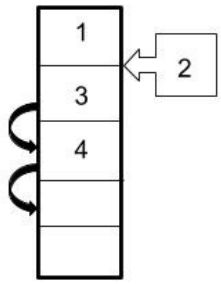
Problème

Le mécanisme d'indexation vise à répondre à un problème simple: accéder le plus rapidement possible à une donnée sur demande.

Comme nous l'avons vu en classe, une manière d'accélérer la recherche est d'utiliser la fouille dichotomique/binaire (ou par interpolation linéaire, mais pour le devoir, nous nous concentrerons sur la fouille dichotomique/binaire), mais il faut pour cela que les données soient ordonnées.

Illustration du problème

Si on n'a qu'un tableau, aucun problème, il suffit de maintenir l'ordre au moment de l'ajout. Par exemple, on détermine où la valeur 2 doit être insérée, puis on la met en place.



Dans les bases de données relationnelles, cependant, un problème survient. Puisqu'il y a plusieurs colonnes, en garder une en ordre implique que l'ordre des autres ne pourra pas être maintenu.

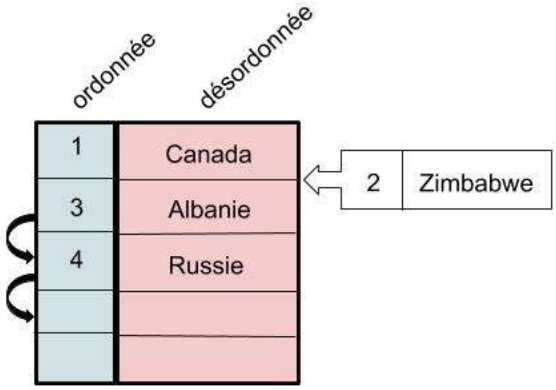
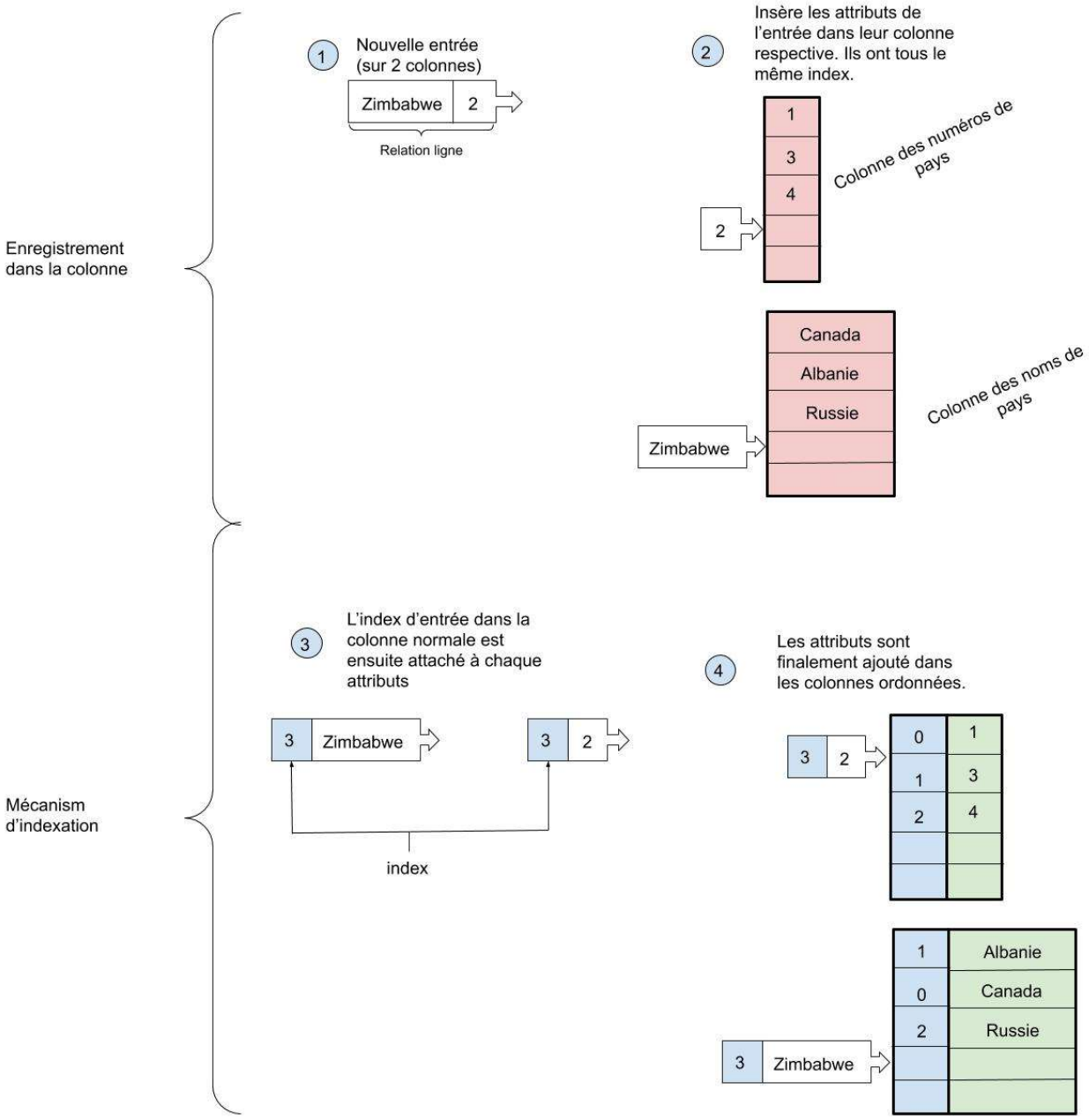


Illustration de la solution

L'indexation répond à ce problème en construisant un registre séparé pour mémoriser l'ordre de chaque colonne. Les registres contiennent à la fois les données et leur relation avec la ligne.

La figure suivante illustre le processus:



Tout d'abord, prenez bien en note qu'il y a ici 2 colonnes séparées. Mais dans votre code de test, vous pouvez démontrer le mécanisme avec une seule colonne indexée.

Dans l'exemple

1) La nouvelle donnée (ligne) contient 2 attributs : un nom de pays et un chiffre.

2) Les deux valeurs sont placées au même index, à la fin, dans leur colonne “normale” respective. Notez que dans l’exemple, ils sont à l’index 3.
3) L’index d’insertion, le 3 dans ce cas-ci, est joint aux attributs. (on a alors des paires index :attribut)
4) Chaque attribut est ensuite inséré dans la colonne indexée par rapport à leur ordre logique (numérique ou lexicographique). L’index reste associé avec la valeur.

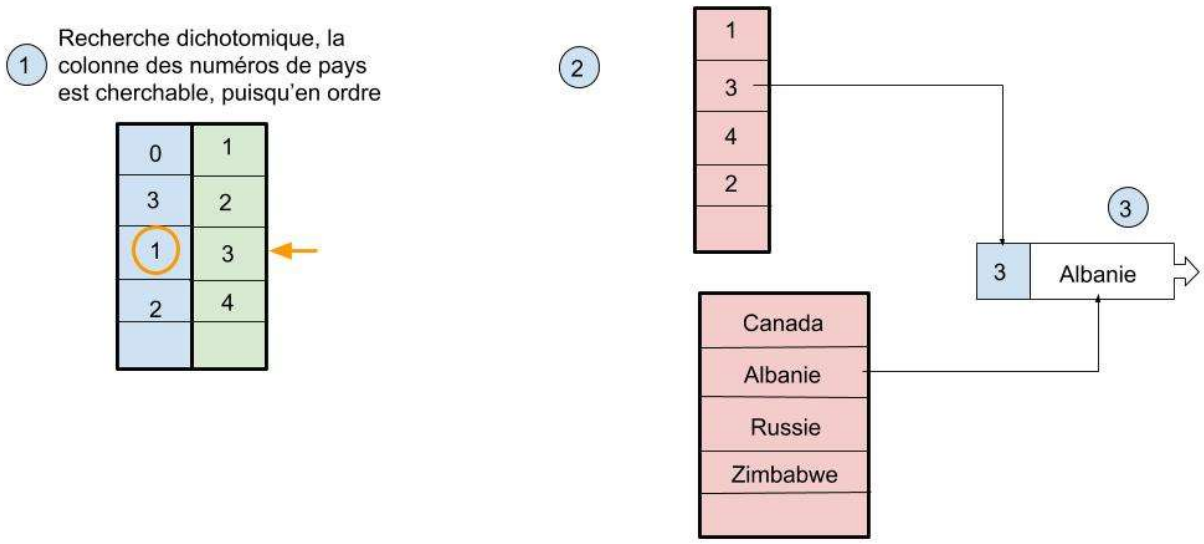
Explication de la solution

Voyons maintenant comment l’indexation peut être utilisée pour accélérer l’accès aux données. Tout d’abord, rappelez-vous bien que la recherche dichotomique est beaucoup plus rapide que la recherche linéaire, c’est un fait établi.

Cas 1

On désire obtenir la ligne du pays numéro 3.

1. Lancer une recherche dichotomique dans la colonne indexée des numéros de pays, trouve l’index de la ligne de l’entrée du pays numéro 3.
2. Utiliser l’index de la ligne pour aller chercher les informations dans les colonnes simples.
3. Réassembler la ligne (en groupant les attributs), puis retourner la réponse.

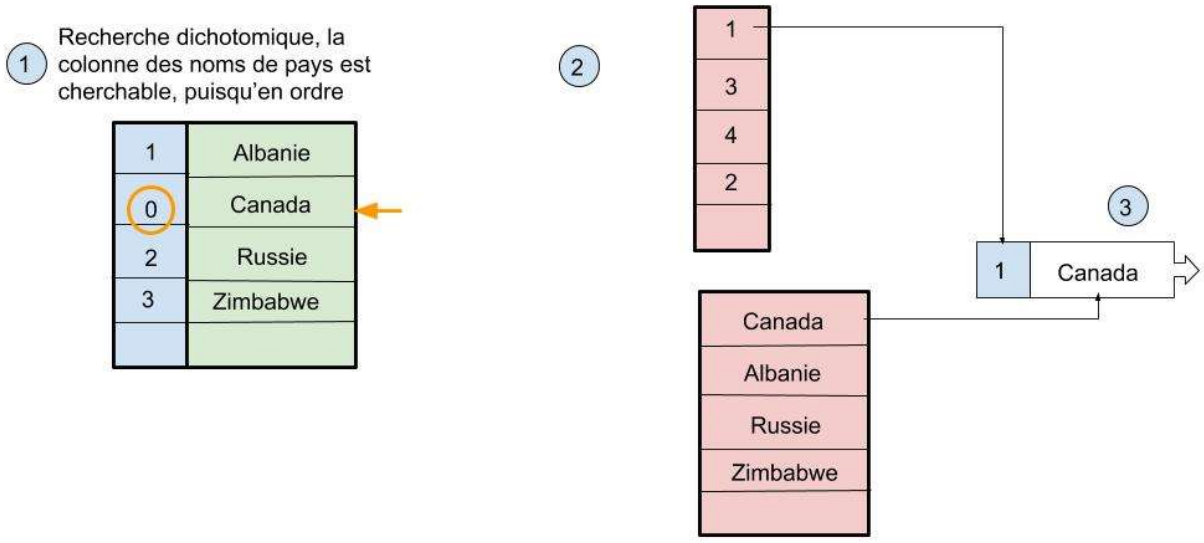


Cas 2

On désire obtenir la ligne du Canada.

1. Lancer une recherche dichotomique dans la colonne indexée des noms de pays, trouve l’index de la ligne de l’entrée du pays Canada.
2. Utiliser l’index de la ligne pour aller chercher les informations dans les colonnes normales.

3. Réassembler la ligne, puis retourner la réponse.



Le mécanisme d'indexation permet donc de trouver l'index d'une ligne à partir du contenu recherché.

Implémentation

Dans les illustrations précédentes, la colonne simple est en rouge. Elle garde les données dans l'ordre chronologique d'entrée, puisque toutes les entrées sont ajoutées à la fin.

La colonne indexée, une classe dérivée de Colonne, ajoute un nouveau tableau qui contient les index et les valeurs. Comme point de départ, un fichier vous est fourni avec le tableau et la classe interne qui regroupe les index et valeurs. Ils vous restent à implémenter la logique d'insertion et d'extraction des données.

Voici quelques explications supplémentaires sur chacune des méthodes à implémenter:

ajouterValeur(V valeur)

- Dans la colonne normale, l'insertion se fait à la fin. L'index de l'élément dans la colonne normale est donc facile à déterminer.
- Ensuite, il faut créer la paire index-valeur en instanciant un objet de type ValIndexee (fournis).
- Avant de pouvoir insérer l'élément dans la table *valIndexee*, il faut d'abord ajuster la taille du tableau au besoin (agrandir, si on est à la taille max).
- Ensuite il faut trouver l'index où insérer dans la table *valIndexee*. Pour le devoir, cette étape doit être réalisée avec la fouille linéaire. Prenez note, par contre, que l'on pourrait utiliser la fouille binaire pour accélérer l'insertion.

NOTE : tous les types de valeurs supportés doivent implémenter l'interface Comparable, qui impose la définition de *compareTo()* (référez-vous à la documentation Java). Pour le devoir le seul type qui sera indexée est la **String**, cependant nous vous recommandons d'effectuer vos tests avec des **Integer**. Les deux types implémentent déjà compareTo().

- Une fois qu'on a trouvé l'index, procéder à une insertion avec décalage dans le tableau *valIndexee*.

obtenirIndex(V valeur)

C'est cette méthode qui doit inclure la recherche binaire.

- Vous recevez la valeur à chercher, vous devez la trouver dans le tableau *valIndexee.*
- Quand vous l'avez trouvé, vous devez retourner l'index contenu dans l'instance de **ValIndexee**.

Validation

Voici une procédure de validation et le résultat qui a été obtenu avec la solution.

```
private static boolean test_colonneIndexee() {

	// résultat tu test
	boolean resultat = true;

	// instanciation d'une colonne indexée d'Integer
	ColonneIndexee<Integer> colonneIndexee = new
                                        ColonneIndexee<Integer>();

	// prépare la valeur témoin et sa position aléatoire
	int valeurTest=0;
	int posValeurTest=rand.nextInt(5)+5;
	int i=0;

	// entre des valeurs aléatoires
	for(;i<posValeurTest;i++) {
		colonneIndexee.ajouterValeur(rand.nextInt());
	}
	// insère la valeur témoin
	colonneIndexee.ajouterValeur(valeurTest);

	// complète avec d'autres valeurs
	for(;i<10;i++) {
		colonneIndexee.ajouterValeur(rand.nextInt());
	}

	// vérifie que la valeur témoin est dans le tableau à
	// l'index trouvé
	int index = colonneIndexee.obtenirIndex(valeurTest);
	try {
		if(colonneIndexee.obtenirValeur(index)!=valeurTest) {
			resultat = false;
		}
	} catch (Exception e) {
		resultat = false;
	}

	// affiche le contenu de la colonne indexée
	colonneIndexee.afficherContenu();

	return resultat;
}
```

```
colonne indexée
index:valeur
5:-2090608677
0:-1894863579
3:-1347664915
6:-301694983
2:-277723390
9:0
7:294643832
4:333156737
10:865462456
8:1928425627
1:2077032500

colonne
-1894863579
2077032500
-277723390
-1347664915
333156737
-2090608677
-301694983
294643832
1928425627
0
865462456
```

Annexe B – Salage

Nous ne vous demandons pas d'implémenter les détails du salage[4], mais vous devez utiliser les méthodes utilitaires qui s'y rattachent. Une bonne compréhension du mécanisme vous aidera lors de l'implémentation.

Le salage est utilisé dans le cadre d'un problème tout simple : dans les systèmes à mot de passe, comment sauvegardons-nous le mot de passe des utilisateurs d'une manière sécuritaire?

La première approche au stockage de mots de passe est le stockage en texte lisible (plain text). C'est-à-dire copier le mot de passe tel quel dans la base de données. Un problème évident survient, dès qu'un pirate informatique gagne accès à la base de données, il gagne automatiquement accès aux mots de passe. Cette approche est très déficiente et ne devrait jamais être utilisée.

Nom utilisateur	Mot de passe
admin	admin
username	123456
charles	qwerty
...	...

La seconde approche est un peu mieux. Plutôt que de sauvegarder le mot de passe, on utilise une transformation mathématique difficilement réversible que l'on appelle *hash*[5]. L'intérêt est que même si un pirate informatique compromet la base de données, il doit effectuer une quantité de travail importante avant de retrouver le mot de passe. Mais considérant qu'il s'agit d'un problème fréquent pour les pirates informatiques, ceux-ci ont développé une parade : les tables arc-en-ciel[6] (*rainbow table*). Ces tables contiennent les résultats précalculés d'un grand nombre de *hash*. Si la *rainbow table* de votre algorithme de hachage est publié, le hachage est aussi faible que l'encodage en texte lisible.

Nom utilisateur	hash
admin	dum7z
username	bOTwZ
charles	0VeCb
...	...

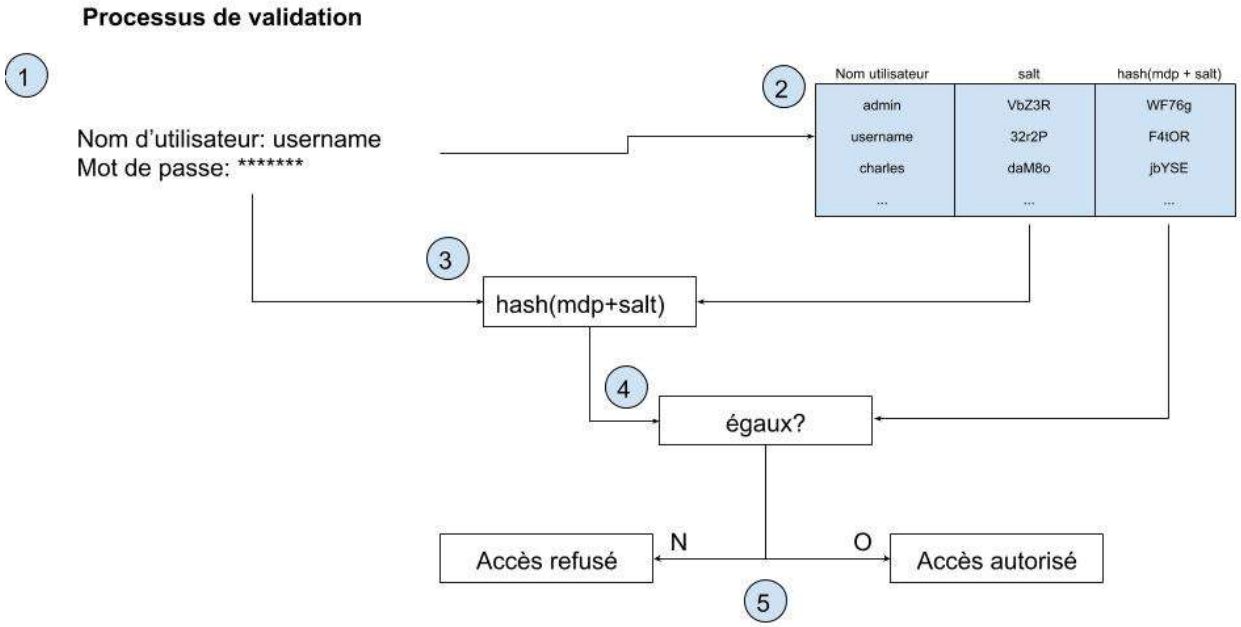
La troisième est meilleure approche. Elle consiste à mélanger votre mot de passe avec un *salt* (figurativement, l'ajout d'une pincée de sel) avant de la hacher. Les *rainbow table* ne peuvent

[4] https://fr.wikipedia.org/wiki/Salage_(cryptographie)
[5] https://en.wikipedia.org/wiki/Cryptographic_hash_function
[6] https://en.wikipedia.org/wiki/Rainbow_table

contenir qu'un certain nombre de mots de passe, généralement dérivés des statistiques connues sur l'utilisation des mots de passe (approche dictionnaire). Le *salt* augmente l'entropie de votre mot de passe avec un nombre aléatoire. Cet ajout nullifie pratiquement la possibilité d'utiliser une *rainbow table*[7]. Il y a un détail important à cette approche par contre. Puisque le *salt* entre dans l'opération du hachage, il faut avoir accès au *salt* à chaque fois que l'on veut valider le mot de passe. Par conséquent, il faut stocker le *salt* dans la base de données. Cela peut paraître contre-productif, mais sachez que même si le pirate a accès au *salt*, il doit quand même calculer la *rainbow table* pour trouver votre mot de passe. La quantité de travail est telle qu'il cherchera plutôt à compromettre une autre base de données.

Nom utilisateur	salt	hash(mdp + salt)
admin	VbZ3R	WF76g
username	32r2P	F4tOR
charles	daM8o	jbYSE
...	...	...

Dans le cadre du devoir, vous aurez à implémenter le processus de validation suivant. Sa réalisation implique plusieurs objets du programme.



1) L'utilisateur entre son nom et son mot de passe.
2) Le nom de l'utilisateur est utilisé pour accéder au *salt* et au *hash* dans la base de données.

[7] https://stackoverflow.com/questions/420843/how-does-password-salt-help-against-a-rainbow-table-attack

3) Le hash du mot de passe entrée par l'utilisateur est calculé à l'aide du *salt*.
4) Le résultat est comparé au hash stocké dans la base de données.
5) S'il y a correspondance, le mot de passe est le bon et l'accès est autorisé, sinon l'accès est refusé.

Instructions de remise

La remise se fera par moodle. Vous devez vous enregistrer comme une équipe, à l'aide de l'interface disponible.

Vous avez jusqu'à la date/heure limite pour soumettre votre travail, tous les travaux remis plus de quelques secondes après l'heure de la remise seront pénalisés.

Votre remise ne doit contenir que le code source, soit les fichiers .java et le dossier src/. Tout autre fichier présent sera considéré comme un manquement aux instructions.

Tous les membres de l'équipe qui ont travaillé sur un fichier doivent être identifiés, au minimum, dans les commentaires d'en-tête du fichier en question.