| Cours 420-266-LI<br>Programmation orientée objet II<br>Hiver 2025<br>Cégep Limoilou<br>Département d'informatique<br><br>**Professeur :** **Martin Simoneau**<br>**Martin Couture** | **TP3 (14 %)**<br>• Fichiers<br>• Exceptions<br>• Algorithmes |
|---|---|

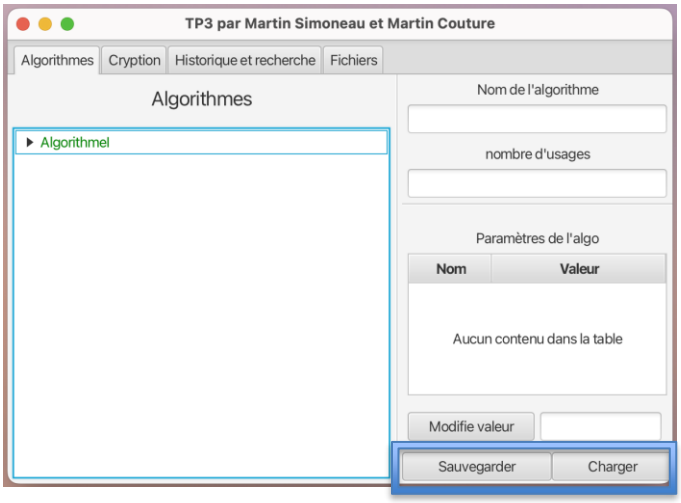
# Objectifs

- Placer l'information importante dans des fichiers
  - Texte
  - XML/json
  - binaire
- Programmation OO
  - *Algorithmes avec collection*
  - Connexions d'objets
  - Réutilisation
  - Documentation, assertions et commentaires

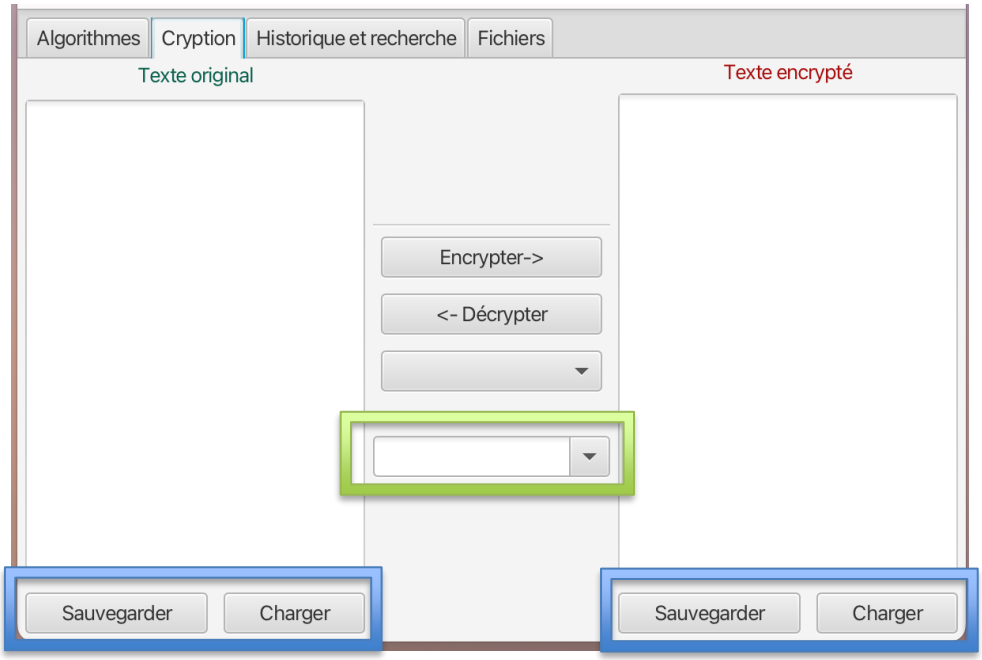
**Contexte :**

- Remettre vos projets sur *Léa* à la date indiquée
  Le travail se fait en équipe de 2, la copie est interdite :
  Il est interdit de donner son travail. Ne laissez pas traîner votre code, si quelqu'un d'autre le trouve, vous serez aussi en situation de triche.
  Il est interdit d'utiliser le travail d'un autre.
- Vous ne pouvez modifier le code que dans le package *etu*. Il est interdit de changer le code des packages *echange* et *fx*.

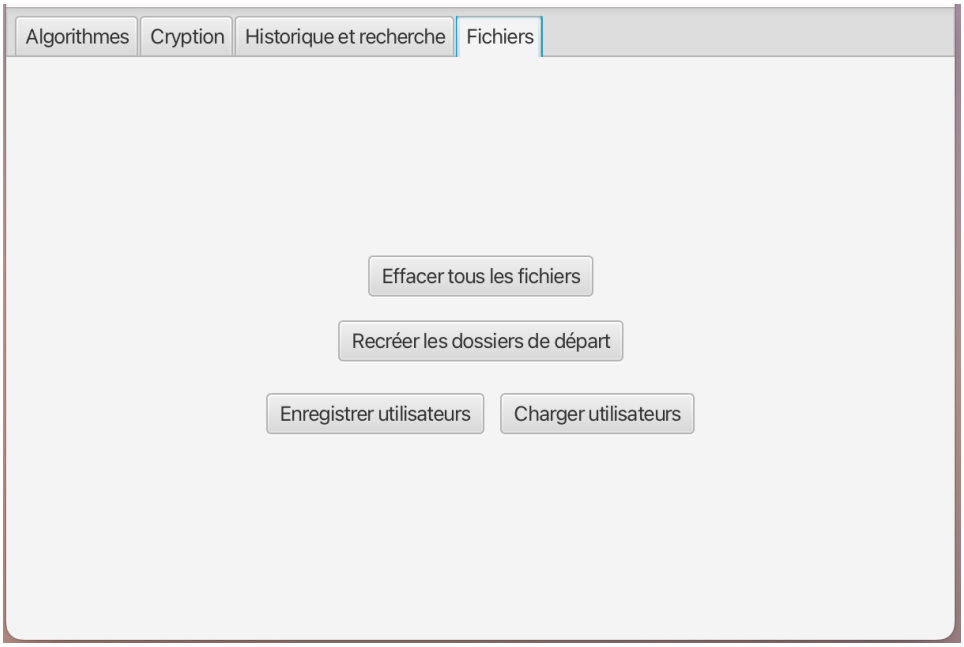
**Présentation de l'application :**




- Dans la page principale, on peut maintenant enregistrer et charger une version des paramètres dans un fichier




- Il est possible de charger et de sauvegarder les fichiers de texte originaux et les encryptions directement dans l'interface.
- Le menu encadré en vert permet de changer l'utilisateur actuel. Un admin, un invité et un utilisateur ayant votre nom y figurent dès le lancement de l'application. Vous pouvez ajouter de nouveaux utilisateurs simplement en y saisissant un nom et en appuyant sur « enter ». Pour simplifier le travail, on n'utilise pas de mot de passe.




- Il est possible d'effacer tous les fichiers produits par l'application
- Il est possible d'enregistrer tous les utilisateurs et de les recharger.

## Installation

### Installation

- Pour faire plus simple, on vous fournit un projet complet pour le tp3. Vous allez devoir:
  - Y remettre :
    - Votre package *algo*
    - Votre package *recherche*
  - Remplacer le *Modele* qui s'y trouve par le vôtre (celui du TP2)

## Fichier texte pour encryption et decryption

NB Les Streams ont tous un constructeur surchargé qui reçoit un File au lieu d'une String. Voir API

io.***TexteIO*** :

*Cette classe s'occupe des fichiers textes. Elle doit accomplir les tâches suivantes :*

- `void enregistreTexte(File fichier, String texte)`
  - Enregistre un fichier texte ligne par ligne en respectant le format de ligne utilisé par le système d'exploitation.
  - La méthode reçoit en paramètres :
    - Le fichier qui doit être mis sur le disque
    - Le texte qui doit aller dans le fichier
  - Dans la section gestion des exceptions, on vous indiquera comment gérer les exceptions pour cette méthode
- `String chargeText(File fichier)`
  Charge un fichier texte standard complet dans une chaîne de caractères. Attention à bien gérer les fins de ligne !
  - La méthode reçoit en paramètres :
    - Le fichier qui contient le texte à charger.
  - Dans la section gestion des exceptions, on vous indiquera comment gérer les exceptions pour cette méthode.
- `Map<String, String> chargeRessource()`
  - Charge le fichier de ressources de l'application. Le fichier se nomme ***configuration.txt*** et il vous a été fourni à la racine du projet. Vous devez le placer au bon endroit, c'est une ressource importante.
  - La méthode charge chacune des propriétés dans une *Map* qu'elle retournera à la fin.
    - Chaque ligne du fichier de configuration correspond à une propriété.
    - Le nom de la propriété est séparé de sa valeur par un « : »

      `Exemple : format-fichiers-parametres:json`
  - Dans la section gestion des exceptions, on vous indiquera comment gérer les exceptions pour cette méthode.

## Fichier paramètre des algos

io.*IO* :

*Cette classe s'occupe des fichiers json et xml. Elle doit accomplir les tâches suivantes :*

- `public void sauvegardeAlgo(AlgorithmeI algo)`
  - Reçoit un algo en paramètre et doit mettre ses paramètres sur le disque. Éviter de mettre des éléments inutiles sur le disque.
  - On a ajouté l'annotation *@JsonTypeInfo* sur *AlgorithmeI*, ceci dernier permet à Jackson de savoir quel algorithme a servi à produire les données et par conséquent à l'utiliser lors de la lecture.

- L'algorithme doit être écrit dans un fichier qui porte le nom de l'algorithme avec l'extension qui correspond à ce qui a été retrouvé dans les propriétés du fichier de configuration par le gestionnaire de fichiers (voir plus bas). Exemple : *Rotation paramètre.xml.*
- Les fichiers produits par cette méthode se retrouvent dans le dossier *fichiers/parametres*.
- Chaque algorithme ne peut être sauvegardé que dans un seul fichier. Il y a une seule sauvegarde par algorithme.

- `AlgorithmeI chargeAlgo(String nomAlgoRecherche)`
  - La méthode reçoit un nom d'algorithme pour lequel elle doit retourner la version du même algo qui a été mise sur le disque. Seuls les paramètres importent pour l'algo retrouvé dans le fichier, les autres données ne sont pas importantes, on va garder les données de l'algo courant.

## • Fichier binaire pour enregistrer les utilisateurs

*Io.**UtilisateurIO :***

- Cette classe est responsable d'écrire et lire sur le disque, l'utilisateur qui est sélectionné dans l'onglet d'encryptage.
- Présentement, l'attribut nombreFichierGenere de l'administrateur et l'attribut nombreUsages de l'utilisateur ne sont pas géré. Pour bien distinguer les différents utilisateurs sur le disque vous pouvez

*Cette classe s'occupe des fichiers json et xml. Elle doit accomplir les tâches suivantes :*

- `void sauvegardeUtilisateur(File fichier, AbstractUtilisateur abstractUtilisateur)`
  - La méthode reçoit un utilisateur et un fichier en paramètre. Elle doit écrire cet utilisateur sur le fichier en format binaire (utilisez l'extension .uti)
  - Un seul utilisateur est emmagasiné dans un fichier nommé ***utilisateur.uti*** . Le fichier est écrasé à toutes les nouvelles sauvegardes.
  - Si vous implémenter le code en respectant les principes SOLID, vous aurez un bonus. Vous avez le droit de modifier les classes utilisateurs pour y arriver
- `AbstractUtilisateur chargeUtilisateur(File fichier)`
  - La méthode charge l'utilisateur dans le fichier reçu en paramètre.
  - La difficulté de cet exercice tient au fait qu'il y a une hiérarchie de classe Utilisateur. Il est relativement facile de le l'écrire parce qu'au moment de l'écrire toute l'information est connue. Par contre, au moment de le lire, on ne sait pas quelle classe utiliser… Il faudra trouver une stratégie pour y arriver.
  - Si vous implémenter le code en respectant les principes SOLID, vous aurez un bonus. Vous avez le droit de modifier les classes utilisateurs pour y arriver

## Gestionnaire de fichier

io.***GestionnaireFichiers*** :

- Vous devez créer une classe qui implémente *GestionnaireFichierI*. Elle sera votre gestionnaire de fichier
- Le gestionnaire de fichiers doit être créé dans la méthode d'initialisation de votre classe qui implémente *ApplicationModelI*. Comme le moteur de recherche, votre *GestionnaireFichierI* doit également être transmis au UI.
- Dès la création du gestionnaire de fichiers, ce dernier doit charger les propriétés du fichier de ressources en s'aidant de la classe *TexteIO*. Il doit également conserver les propriétés pour usage ultérieur dans une map.
- Trouver un moyen pour que la classe *IO* écrive soit des fichier *xml* soit des fichiers *json* en fonction de la propriété ***format-fichiers-parametres***.
- La méthode prepareDossier sera lancée au démarrage de l'application pour s'assurer que tous les dossiers nécessaires soient toujours présent
- Notez que l'objet *ApplicationUI* reçu peut être important pour votre gestionnaire de fichiers.

- *La classe doit implémenter l'interface GestionnaireFichierI. Elle devra donc interagir avec le UI par les méthodes suivantes :*
    - `public void viderDossiersFichiers()`
        - efface le contenu de tous les sous-dossiers du dossier de travail fichier (*parametres*, *encryption* et *texte-originaux*)
    - `void prepareDossiersRequis()`
        - Cette méthode est appelée lorsque l'application est lancée pour s'assurer que tous les dossiers nécessaires sont présents. La méthode doit donc s'assurer que les 3 sous-dossiers de travail sont présents.
    - `void enregistreTexte(File fichier, String texte)`
        - enregistre le texte reçu en paramètre dans le fichier reçu en paramètre à l'aide de la classe *TexteIO*.
    - `String chargeTexte(File fichier)`
        - Charge le fichier demandé en paramètre et retourne le texte à l'aide de la classe *TexteIO*.
    - `File getDossier(Dossiers dossier)`
        - Retourne un objet *File* qui correspond à l'enum reçu.
    - `void sauvegarderParametreSelectionne(AlgorithmeI algo)`
        - Enregistre les paramètres de l'algorithme reçu en paramètre à l'aide de la classe *IO.*
    - `void chargeParametresDansAlgo(AlgorithmeI algoRecherche)`
        - Charge l'algorithme demandé en paramètre à l'aide de la classe *IO*.
        - La méthode doit également remettre les paramètres obtenus de IO dans l'algorithme reçu en paramètre.
        - Les algorithmes qui n'ont pas de paramètres ne doivent pas être traités.
    - `void sauvegardeUtilisateur(AbstractUtilisateur abstractUtilisateur)`
        - Enregistre l'utilisateur reçu sur le disque
        - Doit utiliser *UtilisateurIO* pour y arriver.
        - Le fichier doit se nommer u*tilisateur.uti* .
    - `public AbstractUtilisateur chargeUtilisateur()`
        - Charge l'utilisateur qui est dans le fichier *utilisateur.uti* et le retourne.

## Gestion des exceptions

- Les bases des classes d'utilisateur vous sont fournies dans le package ***etu.utilisateur***.
    - Il y a 3 types d'utilisateur concret : Invite Admin et Utilisateur
    - Il y a un seul objet/instance d'invite et son nom est toujours « Invite »
    - Il y a un seul admin
    - Vous pouvez obtenir l'utilisateur actuel par l'objet *ApplicationUI* reçu lors de l'initialisation de votre *ApplicationModelI*.
- Le projet possède déjà une exception maison *TP3Exception* qui est capable de conserver un *AbstractUtilisateur*. Vous devez créer dans le package *io.exception*, l'exception ***TP3FichierException*** qui
    - est une sorte d'exception *TP3Exception* ;
    - conserve le *fichier* impliqué en plus de l'utilisateur;
    - est une exception contrôlée.
- Toutes les méthodes des classes *IO*, *TexteIO* et *UtilisateurIO* doivent lancer des exceptions de type ***TP3FichierException*** lorsque survient un problème
    - Essayer d'être le plus précis possible;
    - Envoyer des messages pertinent et complet;

- Ajouter l'utilisateur et le fichier à l'exception.

- Le gestionnaire de fichiers doit attraper toutes les *exceptions* de fichier *TP3FichierException* et relancer une simple exception *TP3Exception* avec le message suivant (construit à l'aide des informations disponibles dans *TP3FichierException*):

  Cher(e) ***<nom utilisateur>, <message de l'Exception>*** avec le fichier ***<nom du fichier>.***

  **Par exemple :** *Cher(e) Invite, il s'est produit une erreur d'écriture avec le fichier rotation_fixe.json.*

## • Répartition suggérée du travail

Tâches :

a) Fichier texte + ressources + parties correspondantes du gestionnaire de fichiers + adaptations nécessaires du code +.

b) Fichier XML/json + gestion des dossiers + parties correspondantes du gestionnaire de fichiers + adaptations nécessaires du code.

c) Fichier utilisateur en binaire SOLID + parties correspondantes du gestionnaire de fichiers + adaptations nécessaires du code.

Chaque développeur doit faire l'une des 3 tâches. Une tâche ne peut être faite que par un seul développeur. L'ensemble de l'équipe doit gérer les exceptions.

Un **bonus** sera accordé si vous gérer correctement le nombre d'usage de l'utilisateur et le nombre de fichier généré par l'administrateur.

## Critères d'évaluation et exigences:

**Exigences**

- Mettre vos noms en commentaires dans l'en-tête de chacune des classes dans lequel vous avez travaillé.
- Par refactoring, ajouter vos initiales au nom du projet. Exemple : *tp1* devient ***tp1_dev1_dev2***

  Critères d'évaluation :

**Qualité du code**

- Commentaires pertinents et suffisants
- Les noms des méthodes et des attributs sont pertinents et **complets**.
- Il n'y a pas de code inutile ou commenté.
- Formatage effectué dans chaque classe.
- Algorithmes simples, sans détour et efficaces.
- Un développeur de votre calibre ne devrait pas prendre plus de 10 secondes pour comprendre entièrement une méthode de votre projet.
- Le code fonctionne sans erreurs.
- Toutes les tâches demandées ont été accomplies.
- Il y a des **assertions** partout où s'est nécessaire.

- Classification et polymorphisme (au moins 30% de la note)
  - Les principes **SOL**ID sont toujours respectés.
  - La réutilisation et la compatibilité sont sans failles.

- Les noms de classes et des méthodes sont pertinents.
- L'annotation ***@Override*** est toujours utilisée lorsque c'est possible.
- L'abstraction est toujours utilisée lorsqu'elle est pertinente.

---

FIN