Cours 420-203-RE ***Développement de programmes dans un environnement graphique*** Automne 2022 Cégep Limoilou Département d'informatique	Tp3 Simulateur linéaire interactif JavaFX Service JavaFX Conception d'application (20 %)

CONTEXTE DE RÉALISATION DE L'ACTIVITÉ :

- Durée : 5 semaines (jusqu'à la fin de la semaine 14)
- Ce travail peut être réalisé en équipe de 2 maximum.
- Suivre les consignes additionnelles sur le canal *Questions générales* de l'équipe Teams du cours.

OBJECTIFS

- Concevoir une application JavaFX interactive complète
- Programmer les algorithmes requis en utilisant les structures de données appropriées
- Gérer le projet et les ressources convenablement

Remise

- **Changez l'artifactId et le nom du projet pour remplace votre-nom par vos véritables noms (sans accents sans espace)**
- **Remettre une archive en format zip contenant le projet complet.**

Éléments requis:

1. Pour ce TP on ne vous impose pas l'interface GUI. Vous être libres de la produire comme vous le désirez, mais en respectant les exigences minimales demandées.
2. Le UI doit comprendre
    a. Un éditeur de simulation
        i. L'éditeur de simulation doit comprendre au moins 3 *ListView*:
            1. la liste de toutes les simulations connues. Cette liste affiche:
                a. Le nom de la simulation;
                b. Le nombre de pas simulés.
            2. la liste de toutes les variables de la simulation qui est sélectionnée;
                a. le nom de la variable (celui qui est utilisé dans les équations);
                b. la valeur de la variable;
                c. Toutes les simulations doivent avoir au minimum les variables suivantes:
                    i. ***t*** : représentant le temps depuis le début de la simulation.
                    ii. ***dt*** : l'interval de temps en cours
                    iii. ***stop*** : une variable permettant de stopper la simulation en fonction d'une équation logique;
            3. la liste de toutes les équations qui modifient la variable sélectionnée de la simulation sélectionnée:
                a. Le nom de l'équation;
                b. L'expression mathématique de *MathXParser*. Voir *Equation* plus loin.
        ii. L'éditeur de simulation doit offrir un moyen (bouton, menu contextuel…) pour que l'utilisateur puisse:
            1. **Ajouter / modifier / effacer ->** une simulation
            2. **Ajouter / modifier / effacer / changer la valeur / dupliquer** -> une variable
            3. **Ajouter / modifier / effacer** -> une équation

iii. Lorsque l'utilisateur sélectionne une simulation dans la première liste, la seconde liste doit afficher ses variables. Lorsque l'utilisateur sélectionne une variable dans la seconde liste, la troisième liste devrait afficher les équations qui lui sont associées. L'utilisateur ne doit pas pouvoir ajouter une équation si une variable n'est pas sélectionnée et pareillement pour les variables et les listes. Évidemment, en plus de mettre les différents objets correctement dans les listes, il faut les imbriquer correctement.
    1. Simulation possède un historique qui est une liste de tous les états successifs qui ont été calculés. Un état est l'ensemble des variables d'une simulation. L'état 0 est important, car il constitue les valeurs initiales desquelles repart toujours la simulation.
iv. L'éditeur de simulation doit également proposer une fonctionnalité pour **lire** et écrire les simulations complètes sur le disque (avec les variables et les équations).
    1. Vous pouvez le faire avec des boutons ou des menus contextuels. Les fichiers peuvent être écrits en utilisant la sérialisation objet, mais attention les fichiers deviendront illisibles au fur et à mesure que vous modifierez le code.
    2. L'utilisateur doit pouvoir choisir le nom et l'emplacement du fichier lui-même. Utilisez un *FileChooser* (*showOpenDialog* et *showCloseDialog*). Les fichiers auront l'extension "***.sim***" (exemple gravite**.sim**).

b. Un simulateur
    i. Le simulateur doit pouvoir connaître l'équation qui est sélectionnée dans l'éditeur d'équation. C'est cette dernière qui sera simulée.
    ii. Le simulateur doit permettre de:
        1. **modifier** la longueur des pas de temps (période)
        2. **démarrer** la simulation;
        3. **arrêter** la simulation;
        4. mettre la simulation en **pause**. Cette fonctionnalité est plus complexe, car il faut retirer le temps de pause du temps de la simulation. Vous devriez la faire à la fin.
    iii. Le simulateur affiche un label qui indique le temps où est rendue la simulation.
    iv. Le simulateur présente également une liste contenant toutes les variables de la simulation sélectionnée. Vous pouvez la mettre à jour avec un bouton, mais il est préférable qu'elle se mette à jour automatiquement avec un écouteur sur la liste de l'éditeur de simulations.
        1. L'utilisateur doit pouvoir choisir une variable et pendant la simulation cette variable sera mise à jour en continu dans un *LineChart*.
        2. Comme toutes les simulations sont temporelles, le *LineChart* présente toujours le temps en abscisse et la valeur de la donnée choisie en ordonnée.
        3. Pour un extra vous pouvez permettre d'afficher plusieurs variables sélectionnées en même temps, mais c'est significativement plus difficile. N'essayez de la faire qu'à la toute fin.

c. Un calculateur
    i. Vous pouvez réutiliser le calculateur qui a été développé dans le TP2. On va le modifier légèrement:
        1. Au lieu d'avoir une liste de fonction personnalisée, on utilisera les équations du simulateur. Les équations
        2. Il faut ajouter un moyen (bouton, menu contextuel…) pour permettre à l'utilisateur de transférer la dernière valeur calculée vers la variable qui est sélectionnée dans la liste de l'éditeur de simulation.

d. (pour les équipes de 2 seulement) Une animation qui redimensionne et déplace chaque fenêtre à une position prédéterminée sur l'écran de sorte que l'utilisateur puisse bien voir l'ensemble de

toute l'interface. La fonctionnalité est déclenchée par un menu ou un bouton qui doit être évident et bien placé.

3. Exigences minimales
    a. Personne seule:
        i. Au moins 2 fenêtres (en plus du à propos et les dialogues). Les fenêtres doivent contenir:
            1. Simulateur, calculateur et Éditeur de simulation
        ii. L'animation de repositionnement des fenêtres
        iii. 1 scénario de simulation
    b. En plus pour les équipes de 2:
        i. Au moins 3 fenêtres (en plus du à propos et des dialogues).
        ii. L'animation de repositionnement des fenêtres
        iii. Un afficheur spécialisé
        iv. 2 scénarios de simulations dans 2 fichiers

Fonctionnalités requises

v. Sauvegarde des fichiers de simulation.

Architecture, démarche et code fourni

1. Pour aider à vaincre le syndrome de la page blanche, on vous fournit quelques classes de départs et on vous propose une démarche d'avancement. Cela devrait vous permettre d'avancer sans faire d'erreurs trop importantes .
    a. Tout le code qui vous est fourni peut être modifié librement. On vous le donne pour vous simplifier la tâche et pour vous sauver du temps. Attention, si vous vous en écartez trop, ce sera à vous d'en assumer les conséquences. Il sera plus difficile de vous aider à avancer.
        i. ***Simulation***:
            1. Cette classe est responsable de produire un nouvel Etat à partir de l'ancien état lorsqu'on appelle sa méthode *simulateStep*. Attention, la boucle de simulation principale devrait plutôt être dans *SimulationService* parce que ce dernier gère naturellement le temps, le lancement et l'arrêt du simulateur.
            2. La classe accumule également tous les résultats de simulation dans un attribut nommé *historique.* Le premier état de l'historique contient les valeurs initiales qui proviennent de l'éditeur de simulation (liste de variables).
            3. Pour calculer un nouvel état, il faut:
                a. À chaque pas de temps, il faut mettre à jour les valeurs de ***t*** et ***dt*** avant de calculer les valeurs des autres variables avec les équations.
                b. À chaque pas de temps il faut recalculer chaque variable qui possède des équations. **IMPORTANT** pour chacune, il faut additionner la contribution de chaque équation à la valeur actuelle (au début du pas de temps) de la variable.
                c. Les résultats sont placés dans un nouvel état qui sera retourné par la méthode
        ii. ***SimulationService***:
            1. Cette classe est un service JavaFX et elle est responsable de gérer le temps de la simulation. Elle doit appeler la méthode *simuleStep* de la simulation sélectionnée en lui envoyant le temps actuel et l'intervalle de temps. Elle doit également envoyer l'état actuel :
                a. Lorsqu'on redémarre la simulation on doit repartir de l'état 0, si on redémarre après une pause il faut repartir du dernier état simulé.
            2. Le simulateur devrait pouvoir s'arrêter par lui-même en fonction des états présents (par exemple lorsqu'un objet sort du cadre). Pour cela, il faut surveiller

la variable nommée *stop*. La simulation s'arrête lorsque stop est vrai ou lorsqu'elle est interrompue par l'utilisateur.

iii. Etat

1. C'est principalement une classe de données. Elle contient toutes les variables de la simulation pour un pas de temps.
2. Cette classe possède un constructeur par recopie qui déclenche une copie profonde (deep copy) de toutes les variables et équations contenues. Le nouvel objet et tous ces constituants sont donc complètement distincts de l'objet original.

iv. Variables

1. Possède un nom: qui peut être utilisé dans les équations MathXParser (pas d'espace, de caractères spéciaux ou d'accents).
2. Possède une valeur numérique de type double
3. Elle possède aussi un constructeur par recopie
4. Elle contient la liste de toutes les équations responsables de la modifier.
5. *MathXParser* ne reconnaît pas les variables du simulateur c'est pourquoi on a laissé des méthodes de conversion dans la classe *Simulation*.

v. Equations

1. Possède un nom: qui n'est pas utilisé par *MathParser*. C'est pour vous aider à l'identifier.
    a. Possède une expression mathématique de la forme:
        i. f(var1,var2,var3,var4)= …
        ii. Exemple *f(dt,ay)=ay*dt*
    b. Il est essentiel que:
        i. toutes les variables utilisées soient définies dans la simulation;
        ii. toutes les variables utilisées soient dans la déclaration de la fonction.

Contre exemple: *f(x,t)=x+t+y* ne fonctionnera pas parce que *y* n'est pas dans *f(x,t)*. Il faudrait écrire ***f(x,y,t)=x+y+t***

4. Démarche proposée

a. Moteur de calcul

i. Commencez par faire ***Simulation***. Modifiez les *Etat*, *Variable* et *Equation* au besoin. Validez vos résultats avec un jeu d'équations simples. Allez-y progressivement:
    1. Une seule équation *f(x)=x+1*
    2. Une équation à 2 variables pour x: *f(x,y)=x+y*, et pour y: *f(x,y) = x-y*
    3. Une équation impliquant le pas de temps *f(t,x) = x+t*.
    4. Assurez-vous que l'historique possède tous les états et qu'ils sont bien indépendants les uns des autres. Vous pouvez vérifier en utilisant le débogueur.

ii. Programmez ensuite *SimulationService* et assurez-vous que le temps et le pas de temps sont bien mis à jour. Au début, faites simplement l'affichage en console. Gérer d'abord le démarrage et l'arrêt complet de la simulation.

iii. Créez un moteur pour enregistrer et charger la simulation dans un fichier. Saisir une simulation est assez long, vous allez rapidement apprécier de pouvoir les conserver.

iv. Gérer la pause et la variable stop qui permet d'interrompre la simulation en fonction de ce qui s'y déroule.

b. UI

i. Décidez de l'organisation de votre application (faites les fichiers FXML). Restez simple, c'est plus facile d'augmenter la complexité que de la simplifier !

ii. Commencez par faire l'éditeur d'équation, il est essentiel pour pouvoir tester le simulateur.
iii. Faites ensuite le simulateur. Vous pourrez alors tester des simulations plus complexes (gravité ou autre)
iv. Faites l'affichage dans le *LineChart et du TableView*
v. *(pour les équipes de 2).* Créez les affichages spécialisés
vi. *(pour les équipes de 2)* Créez l'animation du positionnement des fenêtres.
vii. Améliorez l'apparence en utilisant des cellules maison dans les *ListView*

Informations utiles

- ListView
  - Pour consulter les éléments d'un *ListView*
    - *List.getItems()*
  - Pour connaître l'élément sélectionné
    - *Liste.getSelectionModel().getSelectedItem();*
  - Pour réagir à un changement de sélection
    - *Liste.getSelectionModel().getSelectedItem().addListener(callback);*

Barème D'évaluation :

- L'interface est propre, bien documentée et **intuitive** (utilisez les tooltips au besoin).
- La simulation personnalisée est originale et suffisamment complexe (pour les équipes de 2).
- La classe *DialoguesUtils* est la seule qui peut utiliser les *dialogues JavaFX.*
- Le code est propre et bien formaté.
- Toutes les fonctionnalités ont été réalisées.
- Toutes les exigences ont été rencontrées.
- Aucune méthode ne dépasse 50 lignes (incluant javadoc et commentaires)
- Il n'y a pas de @ID ou méthode inutile dans le code.
- Toutes les consignes ont bien été suivies et tous les comportements fonctionnent sans problème.
- Les méthodes que vous avez programmées ou modifiées ont une javadoc conforme et des commentaires pertinents.
- Les fonctions sont bien découpées. Les noms des méthodes sont clairs et significatifs. Il n'y a pas de code dupliqué ou redondant.
- Il n'y a pas de *StackTrace* dans la console *IntelliJ.*
- Des solutions efficaces ont été utilisées pour résoudre les problèmes (ex : classe *Function* sans méthode *toString*)

À REMETTRE :

- Il est à remettre à la date indiquée sur Léa.
- Remettez votre projet complet dans une archive .**zip** sur Léa.
- Votre projet doit comprendre les fichiers des simulations que vous avez développées.