Cours 420-203-RE ***Développement de programmes dans un environnement graphique*** Automne 2023 Cégep Limoilou Département d'informatique Professeur : Martin Simoneau	Tp3 (Version 0) **Il peut y avoir de changements surveiller le canal Teams** Simulateur de gravité Développement d'une application graphique complète (20 %)

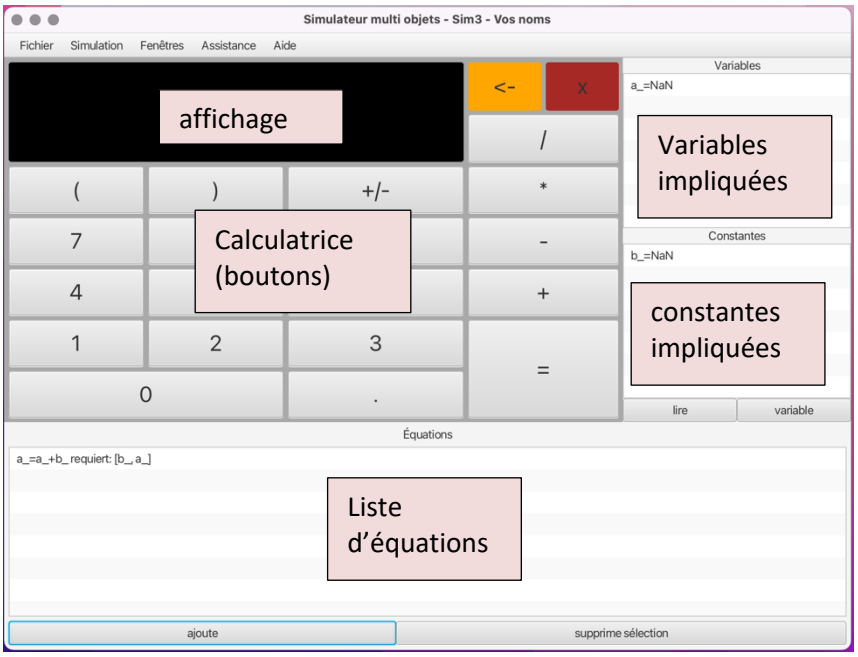
CONTEXTE DE RÉALISATION DE L'ACTIVITÉ :

- Durée : 4 semaines
- Ce travail peut être réalisé en équipe de 2 ou 3 personnes
- Suivre les consignes additionnelles sur le canal *Questions générales* de l'équipe Teams du cours.

OBJECTIFS

- Comprendre et utiliser les événements *javaFX*
- Programmer une application multifenêtre.
- Programmer des animations et des services.
- Prog
- Lier les fonctionnalités à l'interface graphique.
- Créer un code clair et concis.
- Communiquer et partager le travail à faire.

À partir du début d'interface graphique fournie, vous allez devoir réaliser le «simulateur temps-réel» suivant :



- Il s'agit d'un simulateur de gravité multivariable. Le simulateur possède 3 fonctionnalités principales :
  - Saisir des modèles physiques variés (incluant l'enregistrement et le chargement de ces derniers sur le disque). Ici, modèle signifie l'ensemble des équations simulées.
  - Simuler en temps réel le modèle en cours.
  - Afficher les résultats sous forme de tableau ou de graphique au fur et à mesure qu'ils sont générés.
- Le TP3 réutilise la calculatrice multivariable qui a été réalisée dans le TP2, mais en y apportant quelques modifications importantes :

- Toutes les variables peuvent se terminer par un « _ » . On utilisera uniquement les noms de variables terminées par un chiffre pour l'associer à un objet simulé particulier. Par exemple, la variable *a_* n'appartient à aucun objet particulier alors que *a_1* est une variable liée à l'objet simulé numéroté 1. *v_ =a_1+b_2* signifie que la valeur de *v_* est la somme de la valeur de *a_* sur l'objet 1 avec la valeur de *b_* sur l'objet 2.
- Le simulateur fait la distinction entre variables et constantes. Les constantes n'ont pas d'équations qui les modifient pendant la simulation alors que les variables sont modifiées par l'équation qui porte le même nom. Par exemple, avec *a_=b_+1 , a_* est une variable et *b_* est une constante. Simplement, il doit exister une variable pour chaque équation saisie avec le modèle. Les variables du TP2 deviennent les constantes du TP3. Vous remarquerez que la partie gauche de l'interface graphique fait maintenant la distinction entre les constantes et les variables.
- Le simulateur utilise implicitement 3 constantes/variables importantes :
  - ***t_*** le temps total simulé (qui devrait être la somme des d*t_*)
  - ***dt_*** l'intervalle de temps qui s'est écoulé depuis le dernier calcul.
  - ***Stop_*** une valeur qui indique que la simulation doit se terminer.
- *Tous les modèles doivent commencer avec une équation équation sim_ qui indique ce qui doit être simulé et qui force la création des constantes/variables t_, dt_ et stop_ :*
  - ***sim_=d_t&t_&stop_& « autres-variables-à-simuler »***

Activités à réaliser pour la partie 1 du TP3

Modification de l'UI :

- La partie gauche de votre UI doit maintenant contenir une liste de variables et une liste de constantes
- Il faut ajouter les 2 menus suivants :

Programmation des fonctionnalités:

- ***Gestions des fichiers***
  - Il est très important de gérer les fichiers tôt dans ce projet, car les modèles physiques seront relativement longs à saisir. Vous n'aurez certainement pas le goût de passer 30 minutes à resaisir le modèle chaque fois que vous allez ouvrir l'application pour corriger un bogue ! Nous allons utiliser un fichier texte pour saisir le modèle parce que ce dernier peut facilement être modifié et consulter. On n'utilisera pas la sérialisation objet parce que cette dernière rend les fichiers incompatibles lorsqu'on change le code et parce que les fichiers sont très difficiles à consulter et pratiquement impossibles à modifier.
  - Le menu ***Enregistrer*** doit faire sortir un dialogue qui demande à l'utilisateur de choisir l'emplacement et le nom du fichier. Il enregistre ensuite les données de la simulation dans un fichier avec le format suivant :

```
Équations:
b_=c_*3
a_=a_+b_
Variables:
b_=3.0
a_=5.0
Constantes:
c_=2.0
```

    - Les lignes de texte en jaune sont des marqueurs de section et elles doivent se retrouver intégralement dans tous vos fichiers. Elles seront utiles lorsque vous allez relire le fichier. Vous obtiendrez les équations constantes et variables par le moteur de calcul de votre simulateur.

- Le menu ***Charger*** demande à l'utilisateur quel fichier doit être chargé à l'aide d'un dialogue. Il retrouve ensuite le contenu de chaque section puis le transmet au moteur de calcul. Vous pourrez ajuster les méthodes du moteur de calcul pour y arriver.

- ***Moteur de calcul***
  - Le moteur de calcul doit avoir une structure de données pour contenir toutes les équations. On vous suggère fortement d'utiliser des maps si vous ne l'avez pas déjà fait dans le TP2. La clé sera le nom de la variable et la valeur sera la valeur de la variable.
  - Il doit aussi avoir un attribut pour retenir le pas de temps en cours (type Long).
  - Dans le TP3 on doit faire la distinction entre les constantes et les variables. Il faudra donc une map pour retenir les constantes (qui étaient les variables dans le TP2) et une autre pour les variables (une pour chaque équation saisie).
    - Comme la simulation avance dans le temps en calculant toujours des nouveaux pas de temps (étape de calcul) à partir du pas de temps précédant, il va falloir faire la distinction entre les variables qui appartiennent à l'ancien pas de temps et celles qui sont calculées pour le nouveau pas de temps. Vous aurez donc besoin de 2 maps en tout temps. Celle qui tient les anciennes valeurs et celle qui tient les nouvelles.

      Régle : Toutes les valeurs utilisées pour faire les calculs proviennent des anciennes valeurs, toutes les valeurs produites par les équations sont placées dans les nouvelles valeurs.
  - De plus, on ajoutera un historique qui est une map de map dans laquelle on conservera tous les pas de temps qui ne sont plus utilisés (avant le pas de temps précédant);
- Ajouts et modifications de méthode :
  - `public long avancePasDeTemps()` : Cette méthode sert à indiquer au moteur de calcul qu'un pas de temps est terminé. Elle doit :
    - Ajouter les anciennes données simulées dans l'historique.
    - Faire avancer le pas de temps.
    - Écraser les valeurs de la map des anciennes valeurs avec les nouvelles valeurs. N'écrasez pas simplement l'ancienne map avec la nouvelle, car vous perdrez alors les données qui n'ont pas été simulées dans le nouveau pas de temps.
    - Réinitialiser la map des nouvelles données. Elle doit être complètement vide.
    - Elle retourne la valeur du nouveau pas de temps actuel.
  - `public boolean ajouteEquation(String nouvelleEquation)` : Modifier la méthode existante pour qu'elle ajoute également une variable dans la map des variables (les anciennes) pour retenir les valeurs calculées par cette équation.
  - `public void effaceEquation(String nomEquation)` : modifiez cette méthode pour qu'elle efface également la variable associée à l'équation (l'ancienne et la nouvelle valeur).
  - `public double calcule(Equation equation)` : Modifiez cette équation pour que :
    - Elle enregistre la nouvelle valeur calculée dans la nouvelle variable qui a le même nom que l'équation.
    - Dans le calcul des équations à chaque pas de temps, on voudra pouvoir réutiliser les valeurs qui ont été produites à l'ancien pas de temps. Par exemple pour changer la position d'un objet, on prend la position calculée durant l'ancien pas de temps et on lui ajoute ou retire un delta position.
      - *x_ =x_+vx_ *dt_*
    - Pour indiquer d'utiliser l'ancienne valeur, on réutilise le même nom que l'équation (*x_* dans l'exemple). On aurait aussi pu utiliser *x0_*, mais c'est plus simple d'utiliser le même nom. Donc, si une équation requiert une variable qui a le même nom qu'elle, on ne doit pas lancer un appel récursif infini, mais plutôt utiliser l'ancienne valeur qui a le même nom. C'est très important !
  - `public void reset()` : cette méthode :
    - remet le pas de temps à 1

        - Vide les map de valeurs des constantes et variables.
        - Replace les valeurs dans l'ancienne *map* par la première valeur saisie dans l'historique.
        - Vide l'historique
    - Autres méthodes nécessaires pour gérer l'accès aux variables (setter et getter pour les variables constantes, équations). À vous de les déterminer ce dont vous aurez besoin. Vous pourrez les ajuster au fur et à mesure que vous avancerez dans le travail.
    - `public long avancePasDeTemps()` : cette méthode permet au moteur de calcul de se préparer pour exécuter un autre pas de temps en faisant dans l'ordre :
        - ajoutant les anciennes variables dans l'historique;
        - incrémenter le numéro du pas de temps;
        - remplacer les valeurs des anciennes variables par celles des nouvelles;
        - vider la map des nouvelles variables.
- Tests unitaires
    - On vous fournit de nouveaux tests unitaires pour valider votre moteur de calcul. Ils ne sont pas complets, ils vous aideront simplement à atteindre un niveau de stabilité minimal. Votre code doit satisfaire tous les tests. N'hésitez pas à ajouter d'autres tests si vous en sentez le besoin.
    - On vous

Activités à réaliser pour la partie 2 du TP3 - Simulation

Programmation des fonctionnalités:

- ***SimulationService***
    - Il s'agit ici de faire un service JavaFX qui va faire tourner la simulation en temps réel. C'est-à-dire qui va lancer un calcul d'un pas de temps à chaque intervalle de temps. Pour fonctionner, il aura besoin des informations suivantes :
        - La période d'attente entre 2 pas de temps successifs (le nombre de millisecondes à laisser passer entre deux calculs de pas de temps successifs).
        - L'échelle temporelle, le nombre de secondes simulées par seconde réelle. Si le nombre est supérieur à 1, la simulation est ralentie. Si le nombre est inférieur à 1 la simulation est accélérée.
        - Une référence vers le moteur de calcul qui contient le modèle à simuler.
    - Lorsque le service est lancé, il doit se répéter à chaque « intervalle de temps ». Bien entendu, le système d'exploitation n'est pas très précis et notre service ne sera pas appelé avec une grande précision. Le système d'exploitation, avec *SlowHelper* ou avec un *ScheduledService*, ne lancera jamais votre thread avant le temps, mais il pourra être significativement en retard s'il est lourdement chargé. Pour cette raison, vous devez mesurer le temps qui s'est déroulé entre 2 appels pour connaître le *dt* qui doit être utilisé pour le calcul. Un intervalle de temps peut être mesuré en faisant la différence entre 2 lectures d'horloge (*System.currentTimeMillis()).* N'oubliez pas que la simulation est en seconde alors que l'horloge est en millisecondes. Il faut faire la conversion.
    - À chaque pas de temps, le service doit lancer un pas de calcul du moteur de calcul. Le moteur de calcul ne connaît pas le temps ni le pas de temps. Pour lui, ces 2 données sont des constantes. *SimulateurService* devra donc injecter lui-même les valeurs appropriées pour les constantes (*t_* et *dt_*) avant chaque étape de calcul. N'oubliez pas de tenir compte du réglage d'échelle temporelle en réglant ces valeurs. Notez que pour vous aider à déboguer votre service vous pouvez ***temporairement*** replacer le *dt_* mesuré par une valeur fixe. Ainsi, la simulation ne sera pas perturbée par une séance de débogue très longue.
    - Pour lancer le calcul d'un pas de temps, il faut simplement demander au moteur de calcul de calculer l'équation *sim_*.

- Après chaque étape de calcul, il ne faut pas non plus oublier d'appeler la méthode *avancePasDeTemps* du moteur de calcul pour que ce dernier puisse se préparer au prochain pas de temps.
- Il faut maintenant penser à implémenter un mécanisme d'arrêt automatique. Le simulateur doit arrêter de simuler lorsque la variable stop_ passer à vrai. On pourra donc l'utiliser dans nos modèles pour gérer l'arrêt de simulation de différentes façons :
  - Après un certain (exemple : stop_=t_>tempsLimite_)
  - Lorsqu'un objet sort du cadre (exemple : stop_=x_<0)
- Lorsque la simulation se termine automatiquement, il est important de réinitialiser les variables impliquées et le moteur de calcul pour que le tout puisse repartir directement avec les valeurs de départ.
- La simulation peut également être ***interrompue*** ou ***pauser*** par l'usager. Bien entendu, le service *JavaFX* ne possède pas de mécanisme de pause. Il faudra donc en simuler un. Si l'utilisateur arrête la simulation, il suffit de repartir le tout à zéro (comme avec une fin de simulation automatique). Par contre, si l'utilisateur pause la simulation, il ne faut pas réinitialiser les données afin que la simulation puisse continuer lorsque le service sera relancé (Noter que l'objet *Task* sera recréé!). La pause ajoute une difficulté supplémentaire, le temps qui s'écoule pendant la pause ne fait pas partir de la simulation et il faut en tenir compte lorsqu'on règle les prochaines valeurs de *t_* et *dt_*.

Interface graphique:

Pour le TP3 vous devez concevoir vous-même votre interface graphique. Dans cette section on ne décrit donc pas. Les équipes de 2 devront obligatoirement concevoir 3 fenêtres connectées et les équipes de 3 devront concevoir 4 fenêtres connectées. Toutes les fenêtres doivent être lancées et connectées dès le lancement de l'application. Vous n'êtes cependant pas obligé de montrer toutes les fenêtres dès l'ouverture. Notez que par défaut, si un utilisateur ferme une fenêtre, cette dernière sera masquée et non détruite. On peut simplement rappeler la méthode show pour la faire réafficher de nouveau.
Commencez le projet par faire des fenêtres intuitives qui fonctionnent. Attendez à la fin pour faire améliorer l'apparence et le formatage.

- Modélisation :
  - Vous pouvez farder l'interface de la calculatrice du TP2 pour faire l'interface de modélisation, vous pouvez la modifier à votre guise ou même la refaire à votre goût.
  - Nouvelles fonctionnalités :
    - Il faut ajouter un menu pour indiquer si le moteur de calcul doit avancer le pas de temps lorsqu'il fait un calcul à partir de la calculatrice.
    - Il faut ajouter un bouton ou un menu pour que l'utilisateur puisse modifier l'une des équations saisies (sans la recréée).
  - Enregistrement et récupération du modèle simulés.
  - Faites en sorte que votre application charge un modèle par défaut dès le lancement. Cette fonctionnalité va vous sauver beaucoup de temps pendant le développement de votre application.
- Simulation :
  (c'est probablement l'interface qui devrait utiliser votre service de simulation)
  - Fonctionnalités requises :
    - Lancer la simulation
    - Arrêter la simulation
    - Mettre la simulation en pause
    - Saisir le dt souhaité (celui que le service tentera d'atteindre)
    - Régler l'échelle temporelle (le nombre de secondes simulées par seconde réelle)
- Présentation des résultats
  - Fonctionnalités requises :

- Afficher un résultat choisi dans un *LineChart*.
  - L'axe des **x** est le temps et l'Axe des **y** la valeur.
  - L'utilisateur doit pouvoir choisir la donnée qu'il veut tracer parmi les données du modèle utilisé. Les données possibles doivent donc être déterminées à partir du scénario.
  - Les différentes courbes se superposent d'une simulation à l'autre.
  - L'utilisateur doit pouvoir effacer l'ensemble des courbes sur le graphique.
  - (équipe de 3 seulement) Ajoutez un *tooltip* contenant la coordonnée (t,v) sur chaque valeur affichée dans la courbe. Pour créer des *Tooltip* il faut que chaque donnée ait un nœud qui supporte le *tooltip*. Il faut donc créer de nouveaux nœuds (*Circle*) et les placer sur chaque *XYChart.Data* avec la méthode *setNode().* Pour ajouter le *tooltip* sur le nœud il faut utiliser la méthode statique *Tooltip.install(…)* sur le nouveau nœud.
  - Bonus pouvoir sélectionner plusieurs données à afficher simultanément (assez difficile, car cela demande de la débrouillardise pour gérer des notions non-vues en classe)
- Afficher toutes les données reçues dans une table
  - Toutes les variables disponibles doivent être affichées dans la table
  - Il faut penser à ajouter 2 colonnes pour le temps et le dt
  - (équipe de 3 seulement) l'utilisateur doit pouvoir
  - L'utilisateur doit pouvoir effacer l'ensemble des données de la table
- Afficher le mouvement de tous les corps simulés en temps réel
  - Il faut pouvoir afficher la simulation en temps réel. Notez que ce sont surtout les coordonnées x et y de chaque corps céleste qui sont importantes pour cet afficheur.
  - Vous devez identifier les objets à simuler en fonction du no de variable.
  - Idéalement, retrouvez et déplacez l'objet existant plutôt que de tout effacer et recréer la scène à chaque fois. Vous devriez utiliser une *map* pour accéder facilement aux composants que vous avez déjà créés.
  - (équipe de 3 seulement) faire en sorte que chaque corps céleste puisse avoir sa propre image et sa propre taille. Les équipes de 2 peuvent utiliser un composant *Circle* de taille fixe. Les corps célestes des équipes de 3 devront également laisser une traînée avec un nombre de segments configurable par le UI (utiliser un composant *JavaFX Line*).
  - L'utilisateur doit pouvoir :
    - Démarrer la simulation
    - Mettre la simulation en pause
    - Arrêter la simulation
    - Régler l'échelle spatiale de la simulation (mètres par pixel)
    - Choisir une image

- Animations

  N.B. Toutes les animations doivent être programmées dans une classe du package ***vue***.
  - La personne qui fait le service de simulation n'a pas besoin de faire d'animation supplémentaire.
  - Grande animation des fenêtres. Positionne toutes les fenêtres de l'application pour utiliser un écran au maximum :
    - Les fenêtres n'empiètent pas les unes sur les autres;
    - Toutes la surface de l'écran est utilisée.
  - Animation de l'interface

- (équipe de 3) La personne qui ne fait ni le service de simulation ni l'animation des fenêtres principales devra faire 2 petites animations qui modifient l'UI.
    - Chacune doit impliquer au moins 2 phases ou 2 propriétés ou 2 composants animés.
    - Assurez0vous que les animations apportent quelque chose à votre application. Elles doivent être subtiles, non agressantes.
    - Elles doivent améliorer l'usage de l'UI, rendre l'application plus intuitive ou aider à mieux présenter l'information et/ou les contrôles.
    - Évitez les animations longues et pompeuses.
    - Elles doivent être faites avec des services *JavaFX.*

- Présentation de l'UI
  Chaque membre de l'équipe doit programmer au moins une cellule personnalisée (liste ou table). N.B. Toutes les cellules personnalisées doivent être programmées dans une classe du package *vue*.

    - Cellule personnalisée pour les équations (elle retire les « _ » de l'affichage)
    - Cellule personnalisée pour les constantes et variable (elle retire les « _ » de l'affichage)
    - (équipe de 3 seulement) la table doit pouvoir afficher avec un fond coloré les valeurs qui dépassent un certain seuil.

- Modèles
    - Chaque personne dans l'équipe doit remettre un modèle qui respecte les critères suivants
        - Il doit contenir soit :
            - 3 corps dynamiques avec un arrêt en cas de collision.
            - 3 corps statiques et un corps dynamique avec gestion de la collision (les corps doivent « rebondir »)
            - 1 corps en gravitation simple avec frottement de l'air, gestion des collisions, cible à atteindre et vent constant configurable pendant la simulation.
            - Autres suggestions à faire valider par le professeur.

Répartition du travail d'équipe suggéré
Equipe de 3

semaine	Équipe de 2	Équipe de 3
1	P1 : Moteur de calcul (passe les tests unitaires) P2 : Enregistreur + UI et controlleur de simulation	P1 : Moteur de calcul (passe les testa unitaires) P2 : Enregistreur + UI sim et controleur P3 : Simulation Service
2	P1 : Affichage sim + sim controller + cellule personnalisée P2 : Simulation Service	P1 : Affichage table + Cellule personnalisée P2 : Affichage Sim + sim Controller P3 : Affichage graphique+ Cellule personnalisée
3	P1 : Affichage table + animation des fenêtres P2 : affichage graphique + cellule personnalisée	P1 : 2 Animations supplémentaires P2 : Cellule personnalisée P3 : animation des fenêtres
4	P1 et P2 débogage, ajustements de l'UI et modèle personnel	P1, P2 et P3 débogage, ajustements de l'UI et modèle personnels

Informations utiles

- ListView
  - Pour consulter les éléments d'un ListView
    - *List.getItems()*
  - Pour connaître l'élément sélectionné
    - *Liste.getSelectionModel().getSelectedItem();*
- *Pour séparer une chaîne de caractères*
  - Méthode *split* de la class *String*

Barème D'évaluation :

- Le code est propre et bien formaté
- Aucune méthode ne dépasse 30 lignes (incluant javadoc et commentaires)
- Une méthode ne doit pas prendre plus de 10 seconde pour qu'un de vos collègues puisse le comprendre.
- 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*)
- La présentation de L'UI **(25%)**
- Le code de l'application **(40%)**
  - Qualité du code
    - Respect de l'architecture imposée.
    - Gestion des ressources
    - Séparation du code en méthodes, en classes et en package
    - Commentaires (*javadoc* et commentaires)
  - Normes à respecter:
    - Nomenclature java dont:
      - Nom de méthode commence toujours par un verbe et une lettre minuscule.
    - Ne pas utiliser les instructions break ou continue (sauf break dans une switch).
    - Une seule instruction retour par méthode.
    - Les javadoc :
      - commence par un verbe;
      - tous les paramètres doivent être expliqués ainsi que les contraintes qui se rapportent à chacun d'eux.
    - Une méthode ne doit pas dépasser 30 lignes (incluant la documentation)
    - Évitez les détours inutiles et le code inutilement compliqué.
    - Utiliser des noms **significatifs complets**.
    - Un programmeur de votre niveau devrait comprendre votre code en moins de 10 secondes.
    - Utilisez des constantes ou des propriétés,
      - pas de numéros autres que 0 dans le code.
      - Pas de chaîne de textes hard codées.
- Fonctionnement **(30%)**
- Utilisation adéquate de git (5**%)**

- **Le code appartient à celui qui le commit. Si vous travailler ensemble sur une portion de code comitter les 2 noms. Vous perdrez les 10% si vous ne respectez pas cette règle.**
- **Les messages de commits doivent clair, concis et pertinents. Ce n'est pas un chat!**

À REMETTRE :

- Il est à remettre à la date indiquée sur Léa.
- Remettez votre projet complet dans une archive .**zip** sur Léa.
- Invitez le professeur à faire partie de votre équipe git.