Cégep André-Laurendeau

420-235-AL (Hiver 2023)

Travail pratique 2

Classes, objets, constructeurs, copies, etc.

Échéance : 31 mars à minuit, ou tel qu'indiqué sur Léa

David Giasson; Michel Généreux

Objectif

Appliquer les techniques de programmation orientée objet pour implémenter le code et les structures de données répondants aux besoins décrits dans cet énoncé. Vous devrez également utiliser le logiciel de gestion de versions Git tout au long de ce travail.

Directives

- Ce travail peut être fait **seul ou en équipe de deux :**
  - Si le travail est fait en équipe, indiquez le nom des deux coéquipiers en commentaire en haut de chacun des fichiers de code source.
- Une fois terminé, le travail doit être remis sur Léa :
  - Compressez le répertoire contenant le projet IntelliJ pour votre code et votre référentiel git
  - Déposez ce fichier .zip sur Léa (une seule remise par équipe)
  - Pour vérifier que la remise a bien fonctionné :
    - Téléchargez votre fichier .zip depuis Léa
    - Décompressez le fichier .zip
    - Ouvrez le projet dans IntelliJ et assurez-vous que tout fonctionne correctement
- Le travail doit être remis au plus tard à la date officielle indiquée sur Léa. Après cette date, 10% de pénalité sera enlevé par jour de retard.
- Toute forme de plagiat entrainera automatiquement la note zéro (0) ainsi qu'un rapport officiel à votre dossier. N'oubliez-pas d'indiquer vos références si vous utilisez des extraits de code trouvés sur internet ou provenant d'une autre personne.
- La qualité du français est également importante. Jusqu'à 10% de la note finale de votre travail pourrait être retirée pour cause d'un mauvais usage du français.

Critères d'évaluation

- Fonctionnement correct du programme
- Utilisation judicieuse des structures de données
- Qualité du code (encapsulation, noms des méthodes et variables, formatage, lisibilité, etc.)
- Utilisation appropriée de git et IntelliJ

Mise en contexte

Vous avez appris que les *String* Java sont basées sur de simples tableaux de caractères, et que cette structure de donnée primitive est peu efficace lorsqu'il faut effectuer plusieurs opérations d'insertion ou de concaténation. Puisque votre équipe est chargée de développer un éditeur de texte, vous décidez d'implémenter vos propres classes pour avoir une alternative plus performante aux *String*.

IMPORTANT : VOUS NE POUVEZ PAS UTILISER DE VARIABLES OU DE MÉTHODES DES CLASSES JAVA *String* OU *StringBuilder* NULLE PART DANS VOTRE CODE, EXCEPTÉ DANS LES MÉTHODES *toString()*

Travail à réaliser

Préalable : Utilisation de git (5%)

Tout au long de ce travail, vous devrez utiliser le gestionnaire de code source *git* pour faire le suivi de vos modifications. Vous n'êtes pas obligés d'utiliser GitHub pour héberger votre « répo » en ligne, mais vous devez au moins utiliser git localement (ie. faire régulièrement des « *commit* » dans votre dossier).

Classe 1 : Mot (environ 60 lignes de code -- 30%)

La classe *Mot* servira à représenter un seul mot au sein d'un texte. Puisque chaque mot sera relativement court, mais qu'ils seront très nombreux dans le texte, vous décider d'utiliser une structure de données de type « **Vecteur** » pour implémenter efficacement cette classe.

Vous aurez donc minimalement besoin :

- D'un tableau de caractères au moins assez grand pour contenir toutes les lettres du mot (mais sans non plus allouer trop de mémoire inutilement – vous agrandirez plutôt le tableau seulement lorsque ce sera utile);
- D'un entier pour conserver le nombre réel de caractères dans le mot.

NB. Un constructeur par défaut et un constructeur prenant une String en paramètre vous est fourni avec cet énoncé, mais vous êtes libre de modifier ce code selon vos besoins.

Il n'y a pas d'exigences fermes quant au contenu de la classe *Mot*, mais nous suggérons d'implémenter au moins les méthodes suivantes :

- `public String toString()` => Convertit le mot en *String* pour affichage.
- `public int getLongueur()` => Retourne le nombre de lettres du mot.
- `public char getLettre(int index)` => Retourne le caractère à l'index spécifié.
- `public void ajouter(char lettre)` => Ajoute une lettre à la fin du mot.

    - Le vecteur doit être agrandi automatiquement s'il n'y a pas assez d'espace libre pour accueillir la nouvelle lettre. La nouvelle capacité du vecteur devrait être `(taille_actuelle * 2 + 1)`
- `public boolean inserer(char lettre, int index)` => Si l'index est valide (entre 0 et la longueur du mot inclusivement), ajoute la lettre passée en paramètre à l'index spécifié du vecteur. Sinon, retourne *false*.
    - Le vecteur doit être agrandi automatiquement s'il n'y a pas assez d'espace libre.

Critères de correction	Points
Méthodes *getLongueur(), getLettre()* et *toString()*	5
Méthode *ajouter()* et *insérer()*	10
Augmentation automatique de la capacité du vecteur	10
Qualité du code	5

Classe 2 : Phrase (environ 120 lignes de code -- 65%)

Les mots doivent ensuite être regroupés en phrases. Les phrases étant régulièrement modifiées (ajout ou insertion de mots, etc.), vous décidez d'utiliser une structure de donnée de type « **Liste** » (simplement ou doublement chainée, selon votre préférence) pour lier les mots afin de les assembler en phrases. En fait, tant qu'à implémenter vos propres structures de données, vous décidez même d'utiliser la classe *Mot* créée précédemment pour représenter les nœuds de votre liste. Ainsi, même pas besoin d'ajouter une classe *Nœud* additionnelle!

Il vous faudra au moins :

- Un pointeur vers le premier (et optionnellement le dernier) mot de la phrase;
- Un entier pour obtenir rapidement le nombre de mots de la phrase;
- Et bien sûr, il faut ajouter un pointeur vers le mot suivant dans la classe *Mot*.

Une première ébauche de la classe *Phrase* vous est fournie et contient un constructeur vide, un constructeur acceptant une *String*, ainsi que deux autres méthodes acceptant des *String* (mais elles ne font que passer les appels vers vos propres méthodes, que vous devrez ajouter). Vous êtes libres de modifier ce code comme bon vous semble, mais vous ne devez pas utiliser de variables de type String nulle part ailleurs dans votre code (sauf dans la méthode *toString*()).

Afin de répondre aux attentes de la classe *Main*, votre classe Phrase devra posséder au moins les méthodes suivantes :

- `public String toString()` => Retourne l'ensemble de la phrase convertie en String pour en permettre l'affichage.

- **Important : Il faut ~~insérer~~ afficher un espace entre chaque mot!** Cela peut être fait par la classe *Phrase* ou dans la classe *Mot* directement, soit dès l'ajout d'un nouveau mot ou lors de son affichage seulement, à votre choix. Il peut également y avoir un espace « inutile » à la fin de la phrase.
- L'affichage de vos phrases doit être identique à celui de l'exemple fourni en annexe.

- `public int getNbMots()` => Retourne le nombre de mots de la phrase.
- `public int getLongueur()` => Retourne le nombre total de lettres dans la phrase, incluant les espaces. Cette valeur est donc égale à « *maPhrase.toString().length()* ».

- `public Mot getMot(int indexMot)` => Retourne le mot à l'index spécifié.
- `public char getLettre(int indexMot, int indexLettre)` => Retourne le caractère à l'index *indexLettre* provenant du mot à la position *indexMot* dans la phrase.
- `public char getLettre(int indexLettre)` => Retourne le caractère à l'index équivalent si la phrase était représentée par une ~~simple~~ chaine de caractères *String* normale.

- `Public void ajouter(char c)` => Ajoute le caractère à la fin du dernier mot de la chaine. Si le caractère est un espace, ajoute plutôt un nouveau mot (vide) à la chaine.
- `Public void ajouter(Mot mot)` => Ajoute le mot après le dernier mot de la chaine. ~~-~~ NB. Le constructeur *Phrase(String str)* utilise cette méthode pour initialiser la phrase.
- `Public void ajouter(Phrase autre)` => Ajoute la totalité de l'autre phrase à la fin de la chaine actuelle (en conservant ses mots).

a mis en forme : Police :Italique

- `Public void inserer(char c, int indexMot, int indexLettre)` => Insère le caractère dans le mot à l'index *indexMot* de la phrase, à la position *indexLettre* dans ce mot. Si l'un ou l'autre des index n'est pas valide, retourne plutôt *false* sans effectuer de modifications.
- `Public void inserer(Mot mot, int indexMot)` => Insère le mot à la position *indexMot* dans la phrase. Cet index peut aller de 0 jusqu'au nombre de mots de la phrase inclusivement. Pour tout autre index, retourne *false* sans effectuer de modifications.
- `Public void inserer(Phrase autre, int indexMot)` => Si l'index est valide, insère l'ensemble de l'autre phrase dans la phrase actuelle.

Vous êtes bien sûr libres d'ajouter toute autre méthode publique ou privée qui vous semble utile.

Critères de correction	Points
Méthodes *getNbMots(), getLongueur()* et *toString()*	10
Méthodes *getMot()* et *getLettre(indexMot et indexLettre) – 2 variantes*	15
Méthodes *ajouter(char, Mot, Phrase) – 3 variantes*	10
Méthodes *insérer(char, Mot, Phrase) – 3 variantes*	20
Qualité du code (efficacité, lisibilité, simplicité, etc.)	10

Remise

Une fois terminé, remettez votre travail sur Léa dans un fichier zip contenant :

- Dossier TP2_VotreNom (racine du projet)
  - Dossier caché « .git » contenant votre **référentiel git**
  - Fichier TP2_VotreNomChaineRapide.iml (ou portant nom similaire) pour votredu **projet IntelliJ**
  - Dossier « **src** » contenant vos classes *Mot* et *Phrase* ainsi que toutes autres classes que vous avez utilisé.
    La classe « *Main* » ne devrait pas avoir été modifiée (sauf pour y ajouter votre ou vos noms en haut) car lors de la correction elle sera remplacée par la version de départ.
  - Les autres dossiers ne sont pas nécessaires mais peuvent être remis quand même.

Annexe : Résultat attendu

```
=== Test #1 : Ajout de mots et de lettres ===
=== Test #1 : Ajout de mots et de lettres ===
Merci princesse
2 mots, 16 caractères

=== Test #2 : Insertion de mots et de lettres ===
est dans un
3 mots, 12 caractères

=== Test #3 : Insertions mixtes ===
Merci autre princesse, mais Mario est où?
7 mots, 42 caractères

=== Test #4 : Validation de getMot() et getLettre() ===
Merci princesse, est
a t e
M p ?

=== Test #5 : Test des insertions invalides ===
false false château
false false false
Merci autre princesse, mais Mario est où?
7 mots, 42 caractères

=== Test #6 : Validation intégratice ===
Merci Mario, mais la princesse est dans un autre château!
10 mots, 58 caractères
Merci princesse
2 mots, 16 caractères
```

a mis en forme : Retrait : Gauche : 0 cm, Première ligne : 0 cm

```
=== Test #2 : Insertion de mots et de lettres ===
est dans un
3 mots, 12 caractères

=== Test #3 : Insertions mixtes ===
Merci autre princesse, mais Mario
5 mots, 34 caractères

=== Test #4 : Validation de getMot() et getLettre() ===
Merci princesse, Mario
a t e
M p o

=== Test #5 : Test des insertions invalides ===
false false chateau
false false false
Merci autre princesse, mais Mario
5 mots, 34 caractères

=== Test #6 : Validation intégratice ===
Merci Mario, mais la princesse est dans un autre chateau!
10 mots, 58 caractères

Process finished with exit code 0
```

a mis en forme : Normal