Examen final

Info 211a — Licence 2 MI

13 décembre 2018

> Aucun document de cours/TP/TD n'est autorisé. Le barème (sur 23 points !) est donné à titre indicatif. La notation tiendra compte du respect des conventions de code Java *(1,5 point)* ainsi que de la rédaction et du soin *(1,5 points)*.
> Le sujet comporte 9 questions (introduites par le symbole ⋄) réparties en 6 exercices.

L'objectif de ce sujet est de développer un système de mise en page très simple inspiré de l'≪ algorithme des boîtes ≫ de TEX. La mise en page d'un document consiste à assembler les différents éléments composant ce document (lettres, mots, paragraphes, images, ...) de manière à en faciliter la lecture, par exemple en homogénéisant les espaces entre mots ou entre lignes.

Dans TEX, toutes les entités intervenant dans un document sont représentées par des boîtes. Ces boîtes sont définies par :

— leur contenu (une lettre ou une image par exemple) ;
— leur taille actuelle (la place qu'elles occupent dans la page en train d'être composée) ;
— leur taille maximale et minimale.

Il est possible de définir certaines boîtes comme une ≪ composition ≫ d'autres boîtes. Par exemple, la boîte représentant un mot est modélisée par la succession des boîtes représentant les lettres de ce mot. L'ensemble des propriétés d'une *boîte composite* se déduit des propriétés des boîtes qu'elle contient. La mise en page du document consiste alors à modifier la taille des différentes boîtes le composant afin de respecter au mieux les règles typographiques.

Dans ce sujet, nous nous intéresserons uniquement à la modélisation et à la manipulation des boîtes ; nous n'aborderons pas le problème de leur création.

1 Exercice 1 : Les boîtes, exemple de la classe `Character`

Le concept de ≪ boîte ≫ joue un rôle central dans l'algorithme que nous souhaitons utiliser. Il est représenté par l'interface `Boxable` donnée à la fin du sujet. Toutes les boîtes implémentent l'interface `Boxable` et possèdent trois attributs décrivant :

— la largeur de la boîte ;

— sa largeur *maximale* ;
— si cette boîte est redimensionnable ou non.

La classe `Character` permet de représenter des *caractères* qui sont des boîtes non redimensionnables, de largeur 1, dont la représentation textuelle est une chaîne de caractères (`String`) de longueur 1, passée au constructeur.

◇ *(1 point)* Donnez l'implémentation complète de la classe `Character`.

Exercice 2 : Boîtes composites

Les boîtes composites permettent de définir une boîte comme étant une composition de sous-boîtes. Elles sont modélisées par la classe `CompositeBox` qui implémente (partiellement) l'interface `Boxable`.

Une `CompositeBox` contient une liste de sous-boîtes. Cette classe définit un seul constructeur sans paramètres, une méthode `addBox` qui ajoute une sous-boîte à la boîte composite et lève l'exception `NotAddable` si ce n'est pas possible (à cause d'une erreur dans l'ajout). Elle possède les propriétés suivantes :

— `getText()` est définie comme la concaténation des contenus des sous-boîtes ;
— une boîte composite est redimensionnable si au moins l'une de ses sous-boîtes l'est ;
— sa largeur maximale correspond à la somme des largeurs maximales de ses sous-boîtes redimensionnables (ou est indéfiniment extensible si l'une des sous-boîtes l'est).
— la méthode `extend` renverra une exception indiquant que la méthode ne peut pas être appelée, en utilisant, par exemple, le code suivant :

```
@Override
public int extend(int width) {
  throw new RuntimeException("not implemented");
}
```

◇ *(0,5 points)* Donner (et justifier !) la signature de la méthode `addBox` ;
◇ *(2,5 points)* Donner une implantation complète de la classe `CompositeBox`.

2 Représentation des textes

Exercice 3 : mots

On définit maintenant une classe permettant de représenter les *mots* (classe `Word`) qui sont des boîtes composites non redimensionnables composées uniquement d'un ensemble de `Character`. Le constructeur de `Word` prend en paramètre une `String s`. Le mot est construit en convertissant chaque caractère de `s` en une boîte `Character` et en ajoutant celle-ci au mot. Il n'est pas possible de modifier un mot : après la création de celui-ci, tout appel à `addBox` lèvera une exception (cf. code fournit dans la question précédente).

◇ *(1,5 points)* Définir la classe `Word` à partir des informations précédentes.

Exercice 4 : Séparateurs

Les séparateurs sont des boîtes modélisant la séparation (ou, plus concrètement, la distance) entre deux boîtes. Ils permettent par exemple de décrire l'espacement entre les lettres d'un mot, les mots d'une ligne, ...

Tous les séparateurs héritent de la classe `Separator` donnée à la fin du sujet.

On souhaite définir deux types de séparateurs :

— les ressorts `Spring` qui sont redimensionnables (avec une largeur initiale donnée ou égale par défaut à 0) ;

— les colles `Glue` qui ont une largeur fixée et ne sont pas redimensionnables.

◇ *(0,5 points)* Donner le commentaire (en précisant le rôle de tous les paramètres et de la valeur retournée) correspondant à la méthode `getText` de la classe `Separator` ;

◇ *(2 points)* Donner le code des classes `Glue` et `Spring`.

Exercice 5 : Lignes

Les lignes (classe `Line`) modélisent une ligne du document. Ce sont des boîtes composites redimensionnables. Bien qu'une ligne puisse comporter des éléments de différentes natures (textes, images, équations, ...), nous nous intéresserons uniquement, dans la suite du sujet, aux lignes « textuelles ». Une ligne textuelle est représentée par une boîte composite composée des sous-boîtes représentant les mots ; l'espace entre deux mots est représenté par un ressort (espace extensible) sauf si l'une des boîtes est un signe de ponctuation auquel cas le séparateur est de taille fixe. Ainsi la phrase « `Hello World !` » est représentée par le code :

```
Line c = new Line();
c.addBox(new Word("Hello"));
c.addBox(new Spring(" ")); // espace entre deux mots
c.addBox(new Word("World"));
c.addBox(new Glue(" ", 1)); // espace devant une ponctuation
c.addBox(new Word("!"));
```

Lors d'un agrandissement d'une ligne (appel à la méthode `extend`) il est nécessaire, pour obtenir un résultat « joli », d'agrandir équitalement l'ensemble des sous-boîtes tout en respectant les contraintes de largeur des boîtes. Ce problème d'*optimisation sous contraintes* peut s'avérer très compliqué à résoudre de manière exacte. C'est pourquoi nous adoptons une méthode de résolution approchée, composée de deux étapes :

— dans une première étape, on essaye d'agrandir toutes les boîtes redimensionnables de manière uniforme ; si la division n'est pas entière on répartira les espaces supplémentaires sur les premiers éléments qui l'autorisent ;

— si les agrandissements demandés dans la première étape n'ont pas pu être réalisés (parce que certaines boîtes ont atteint leur largeur maximale), on répartit, dans une deuxième étape, la largeur manquante entre les boîtes qui peuvent encore être agrandies, en espérant que la ligne ait alors la largeur désirée.

◇ *(4 points)* Donner le code complet de la classe `Line`

Exercice 6 : Mise en page

La modélisation « en boîtes » développée dans ce sujet permet de réaliser simplement divers opérations de mise en page. Par exemple, pour justifier un texte, il suffit d'agrandir toutes les lignes pour qu'elles aient toutes la même largeur [1].

On souhaite créer un programme permettant d'illustrer les capacités de la modélisation développée dans ce sujet. Ce programme devra composer un texte de 3 lignes (modélisée chacune par une instance de `Line`, l'afficher, avant d'afficher les trois lignes de manière justifiée. Plus concrètement, sa sortie sera :

```
Un chasseur sachant
chasser sans son chien
est un bon chasseur

Un                  chasseur                  sachant
chasser          sans          son          chien
est           un           bon           chasseur
```

Le premier paragraphe correspond au texte « brut », le second au texte justifié sur 55 colonnes.

◇ *(1 point)* Donner le code d'une classe de test permettant d'afficher les deux sorties précédentes.

3 Exercice 7 : Le mot de la fin (3 points)

◇ Écrivez une fonction qui détermine le mot le plus fréquent dans une chaîne de caractères. On appelle « mots » toute séquence de caractères séparés par des espaces ; si plusieurs mots ont le même nombre d'occurrences, on en renverra un de manière arbitraire.

◇ On rappelle que les classes `ArrayList<E>` et `LinkedList<E>` de la bibliothèque java standard implémentent toutes deux l'interface `List<E>`. On souhaite écrire une fonction de tri. Quels sont les avantages et les inconvénients de lui donner comme signature :
— `public void sort(List t)`
— `public void sort(LinkedList t)`

1. Par soucis de simplification, on supposera que les lignes sont toujours plus petites que la largeur demandée.

4 Annexes

4.1 Interface `Boxable`

```
public interface Boxable {

    /**
     * Retourne la largeur de la boîte
     *
     * @return la largeur exprimée en nombre de caractères
     **/
    public int getWidth();

    /**
     * Retourne la largeur maximale de cette boîte
     *
     * @return la largeur maximale exprimée en nombre de caractères
     **/
    public int getMaxWidth();

    /**
     * Retourne le contenu textuel de la boîte.
     *
     * @return la chaîne de caractères représentant le contenu textuel.
     **/
    public String getText();

    /**
     * Détermine si une boîte peut être agrandie.
     *
     * Une boîte peut être agrandie si elle est redimensionnable et si
     * sa largeur actuelle est (strictement) plus petite que sa
     * largeur maximale.
     *
     * @return true s'il est possible d'agrandir la boîte.
     **/
    public boolean isResizable();

    /**
     * Agrandit la boîte d'une largeur donnée en respectant la
     * contrainte de largeur maximale.
     *
     * L'élargissement se fait en agrandissant le contenu de la boîte
     * (par exemple en agrandissant l'espace entre les lettres).
```

```
     *
     * Si la boîte ne peut pas être élargie (this.isResizable() est
     * faux), cette méthode ne modifie pas la largeur de la boite et
     * renvoie 0. Sinon la boîte est élargie autant que possible (tout
     * en respectant la contrainte de largeur maximale) et la fonction
     * renvoie la largeur ajoutée (c.-à-d. le nombre de caractères
     * dont la largeur de la boîte a été augmentée).
     *
     * @param width l'agrandissement demandé
     * @return la largeur ajoutée à la boîte
     */
    public int extend(int width);
}
```

4.2 Classe Separator

```
public class Separator implements Boxable {

        private String sep;
        private int width;
        private int maxWidth;
        private boolean resizeable;

        protected Separator(boolean resizeable, int width, int maxWidth, String sep) {
                this.resizeable = resizeable;
                this.width = width;
                this.maxWidth = maxWidth;
                this.sep = sep;
        }

        public String getText() {
                String res = "";
                for (int i = 0; i < this.getWidth(); i++) {
                        res += sep;
                }
                return res;
        }

        public int getWidth() {
                return this.width;
        }

        public int getMaxWidth() {
                return this.maxWidth;
```

```
        }

        public boolean isResizable() {
                return this.resizeable;
        }

        @Override
        public int extend(int width) {
                throw new RuntimeException("not implemented");
        }

        public void increaseWidth(int width) {
                this.width += width;
        }

}
```