Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2016-2017**

TD - Séance n°9

Interfaces, polymorphisme

Le but de ce TD est de réaliser un formateur de texte fonctionnant sur le même principe que le formateur de texte `fmt` sous Unix. Un tel programme peut prendre un texte très mal formaté comme celui-ci :

```
Lorem ipsum dolor         sit amet, consectetur     adipiscing elit,
sed do       eiusmod tempor incididunt       ut labore et dolore
            magna     aliqua.
  Ut
                               enim ad           minim
veniam, quis nostrud        exercitation ullamco
laboris nisi ut       aliquip ex ea   commodo       consequat.
```

... et le transformer en :

```
Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore
et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat.
```

Certains éditeurs de texte proposent cette fonctionnalité par défaut. Par exemple, dans Emacs il suffit de tapper `Alt + Q` depuis l'intérieur d'un paragraphe.

Dans notre implémentation, nous allons utiliser un objet de la classe `Scanner` pour lire les mots un à un. Un deuxième objet, le formateur, accumule ces mots en listes de lignes dont chacune contient des mots séparés par des espaces. Le formateur imprime ensuite la liste de ces lignes. Ainsi, le programme supprime les espaces redondants entre deux mots, les lignes vides redondantes entre deux paragraphes, ou les passages à la ligne non nécessaires. Dans la partie facultative, vous pourrez aussi ajouter une option au formateur permettant de justifier le texte.

On modélise le problème de la façon suivante : on introduit le concept de Boîte qui représente les objets composant le texte formaté.

- une *Boîte* est un élément du texte formaté qui a une taille et peut être affiché.
- une *Boîte espace* est un élément du texte formaté qui séparera les mots. Ce sera un seul espace dans le cas ou le texte n'est pas justifié, et potentiellement plusieurs espaces sinon.
- une *Boîte mot* est un élément du texte formaté qui représentera un mot.
- une *Boîte composite* représentera une ligne de texte.

1 Partie obligatoire

Les objets implémentant l'interface `Boite` représenteront une unité de texte d'au plus une ligne, soit un mot, un espace, ou une ligne. Trois classes implémenteront cette interface :

- `BoiteEspace` : un objet de cette classe représente un espace ;
- `BoiteMot` : un objet de cette classe représente un mot ;
- `BoiteComposite` : un objet de cette classe représente une suite horizontale de boîtes ; nous nous en servirons pour représenter les lignes.

Exercice 1
Écrivez la déclaration de l'interface `Boite`, qui contient les déclarations de deux méthodes publiques : `length()`, de type entier, `toString()`, de type String.

Exercice 2
Écrivez ensuite les définitions de deux classes qui implémentent cette interface : `BoiteEspace` et `BoiteMot`. Une `BoiteEspace` a une longueur de 1, et se convertit en la chaîne réduite à un espace `" "` (à ne pas confondre avec la chaîne vide !). Une `BoiteMot` représente une chaîne arbitraire, sa méthode `toString` retourne cette chaîne, et la méthode `length()` retourne sa longueur.

Exercice 3
Écrivez maintenant une nouvelle classe `BoiteComposite` qui implémente l'interface `Boite`. Une boîte composite contient une suite de boîtes ; sa largeur est la somme des largeurs des boîtes qu'elle contient, et sa représentation sous forme de chaîne est la concaténation des représentations des boîtes qu'elle contient.

Remarque : Les objets de la classe `BoiteComposite` *stockeront des objets qui sont soit des mots, soit des espaces.*

En plus des méthodes de l'interface `Boite`, la classe `BoiteComposite` implémente une méthode publique `isEmpty` qui détermine si une boîte composite est vide, et une méthode publique `addBoite` qui ajoute une boîte à la fin d'une boîte composite.

Exercice 4
Écrivez maintenant une classe `Formateur` qui accumule la suite des mots contenus dans un fichier texte. Le constructeur de cette classe prendra un nom de fichier sur lequel il construira un objet de la classe `Scanner`, le *parseur*.
Décrivez les deux méthodes publiques : `read`, qui lit la suite des mots retournés par le parseur et la range dans un vecteur de `Boite`, et `print`, qui imprime cette suite de mots. Le vecteur de `Boite` est organisé comme suit. Tous les mots d'un paragraphe sont accumulés dans une `BoiteComposite`. Le formateur commence la lecture avec une boîte composite vide. A chaque fois qu'il lit un nouveau mot, il y ajoute un espace et la boîte représentant ce mot. Enfin, lors d'une fin de paragraphe, c'est-à-dire quand il trouve une ligne vide, il ajoute la boîte courante

au vecteur de `Boite` et recommence avec une nouvelle boîte composite vide. (Il sera peut-être utile d'utiliser une méthode privée pour la fin d'un paragraphe.)

Pour pouvoir reconnaître le fin d'un paragraphe, la méthode `read` pourrait utiliser un autre objet de la classe `Scanner`. (La classe `Scanner` possède plusieurs constructeurs. Un de ces constructeurs est `Scanner(String source)`.)

Vous ferez attention à n'ajouter d'espaces ni au début ni à la fin d'un paragraphe.

Exercice 5
Modifiez le programme précédent pour qu'il coupe les lignes. Le formateur passe maintenant à une nouvelle boîte composite dès lors qu'ajouter le nouveau mot à la boîte courante lui ferait dépasser la largeur de la page (fixée arbitrairement, par exemple à 75). Cependant, on ne passe jamais à une nouvelle boîte si la boîte courante est vide (pourquoi ?).

Pour des raisons esthétiques, le programme devra insérer une ligne blanche entre deux paragraphes.

2 Si vous avez du temps ...

Le texte produit dans le formateur réalisé n'est pas justifié puisque la marge droite n'est pas alignée. Pour justifier le texte, nous introduisons une nouvelle interface `BoiteEtirable` qui étend l'interface `Boite`. Les objets de l'interface `BoiteEtirable` pourront être convertis en chaînes de longueur arbitraire, ce qui nous permettra de justifier les lignes.

Afin de produire du texte justifié, on modifie la méthode d'impression de la classe `Formateur` pour qu'elle imprime des espaces de largeur variable.

Exercice 6
Commencez par ajouter à l'interface `Boite` une nouvelle méthode booléenne `isEtirable`, et écrivez le code de cette méthode pour toutes les classes qui implémentent `Boite`. Pour le moment, cette méthode retourne `false` pour tous les objets.

Exercice 7
Définissez maintenant une nouvelle interface `BoiteEtirable` qui étend `Boite` en lui ajoutant une méthode `toString(int n)` de type `String`. Dans le reste de cette partie, vous implémenterez cette nouvelle méthode qui doit convertir une boîte en une chaîne, mais en ajoutant `n` espaces supplémentaires aux endroits où cela peut se faire.

Exercice 8
Modifiez maintenant la définition de la classe `BoiteEspace` pour qu'elle implémente l'interface `BoiteEtirable`. Toutes les `BoiteEspace`s sont étirables (la méthode `isEtirable` retourne toujours `true`), et `toString(n)` retourne simplement une chaîne de `n+1` espaces (l'espace d'origine, et `n` espaces ajoutés).

Exercice 9
Le cas d'une boîte composite est un peu plus compliqué. Une boîte composite peut-être étirée dès qu'une des boîtes qu'elle contient peut l'être : la méthode `isEtirable` devra donc vérifier si c'est le cas. La méthode `toString(n)` devra ajouter un certain nombre d'espaces à chaque boîte étirable contenue. Malheureusement, ce nombre n'est pas toujours constant : si une boîte composite contient deux boîtes étirables et qu'il faut distribuer trois espaces, il faudra ajouter deux espaces à la première mais un seul à la seconde. Supposons qu'une boîte composite contienne e boîtes étirables et qu'on veuille l'étirer de n espaces, le nombre exact d'espaces à ajouter à chaque boîte étirable contenue est alors $n_{\text{esp}} = \frac{n}{e}$. Cependant, n peut très bien ne pas être divisible par e ; on calcule donc $n_{\min} = [n_{\text{esp}}]$, la partie entière de n_{esp}, qui est le nombre minimal d'espaces à rajouter à une boîte étirable. Le nombre d'espaces qui nous restent est alors $n_{\text{suppl}} = n - e \times n_{\min}$. La méthode `toString(n)` de la classe `BoiteComposite` devra donc calculer les entiers $n_{\min}$ et n_{suppl} comme ci-dessus, et ensuite retourner la concaténation de ses éléments ; les n_{suppl} premiers éléments étirables devront être étirés de $n_{\min} + 1$ espaces, tandis que les autres devront l'être de $n_{\min}$ espaces seulement.

Exercice 10
Écrivez la méthode `print` de la classe `Formateur` pour qu'elle étire les lignes étirables [1] afin d'arriver à une largeur uniforme de 75 caractères.

Exercice 11
Le programme précédent a un défaut flagrant : il justifie toutes les lignes, même celles qui sont à la fin d'un paragraphe. Il va donc falloir le modifier pour inhiber la justification de ces dernières.

On pourrait penser à définir une nouvelle classe qui ressemble à `BoiteComposite` mais dont les objets ne sont jamais étirables. Cependant, Java n'offre pas de facilités pour changer la classe d'un objet après sa création, donc on n'aurait aucun moyen de changer la classe de la boîte courante au moment d'arriver à la fin du paragraphe.

La solution retenue consiste à inclure dans la classe `BoiteComposite` un nouveau champ booléen `inhibe` qui sert à inhiber l'étirage. Ajoutez une nouvelle méthode `inhibeEtirage()` qui fixe ce champ à `true` et une méthode `setEtirable()` qui affecte `false` à ce champ. Modifiez également la méthode `isEtirable` pour prendre en compte le champ `inhibe`.

Exercice 12
Implémentez la gestion des lignes en fin de paragraphe dans la classe `Formateur`.

1. Pourquoi certaines lignes risquent-elles de ne pas être étirables ?