Université Paris Cité **Programmation Orientée Objet**
L2 **Année 2022-2023**

TP n° 5

Héritage, Récurrence

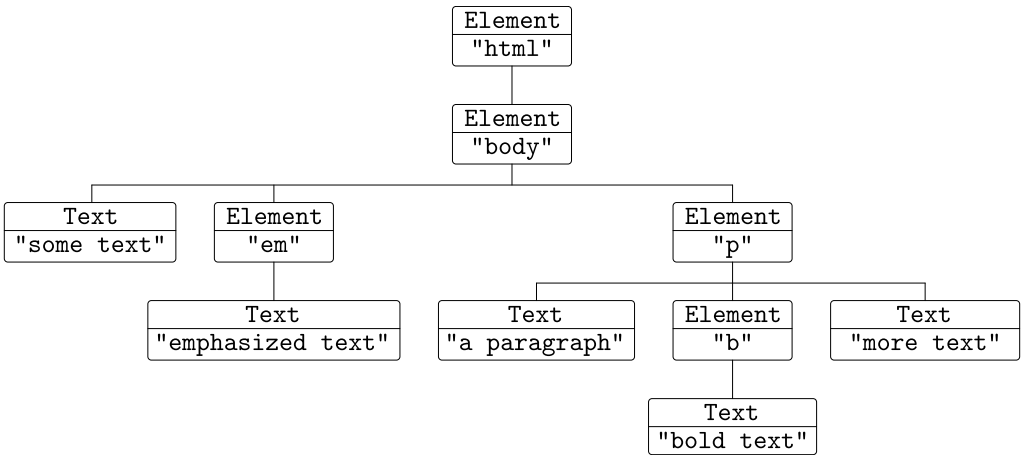
Le contenu d'un document XML (*e.g.* une page HTML) peut être vu comme une structure arborescente dont les nœuds peuvent être de deux sortes :

— Un nœud *élément* encapsule un nom de balise sous forme de chaîne de caractères (`"body"`, `"div"`, ...) ainsi qu'une suite de nœuds (les *fils* de ce nœud).

— Un nœud *texte* encapsule une simple chaîne de caractères.

Le document lui-même, s'il est non vide, encapsule un lien vers le nœud racine de cette arborescence. Voici un exemple de page web, et la manière dont son contenu peut être vu comme un arbre :

```
<html>
    <body>
        some text
        <em>
            emphasized text
        </em>
        <p>
            paragraph
            <b>
               bold text
            </b>
            more text
        </p>
    </body>
</html>
```



Le but de la première partie de ce TP est de définir un ensemble de classes permettant de représenter en mémoire un document XML. Dans la seconde, nous nous servirons de la classe `Scanner` pour convertir en HTML un fichier texte spécifiant, dans un format minimaliste, un assemblage de listes imbriquées.

1 XML

Nous allons d'abord définir les trois classes `Node`, `Text` et `Element`, permettant de représenter les nœuds d'un contenu XML, puis la classe `Document` représentant un document XML.

Une réimplémentation de la méthode `toString()` de `Document` permettra de produire le code source XML d'un document avec son indentation correcte : elle invoquera une méthode définie dans `Node` (`toXML(int indent)`) et implémentée par ses classes héritières, dont l'argument spécifiera l'indentation du nœud courant lors de cette conversion.

Sauf indication contraire, les champs des classes ci-dessous seront privés, et les constructeurs et méthodes publiques.

Exercice 1 Définir une classe `Node`. Cette classe sera l'ancêtre des deux classes représentant les deux sortes de nœuds d'une arborescence XML. Munir cette classe de trois méthodes :

- `static String tabs(int indent)` renvoyant une chaîne formée de `indent` × 4 espaces. Cette méthode peut être écrite par récurrence.
- `ArrayList<Node> childs()` renvoyant une liste vide définie comme un champ statique et privé de la classe `Node`.
- `String toXML(int indent)` renvoyant `null`.

Remarques. Si vous connaissez les classes abstraites, vous pouvez définir la classe `Node` ainsi que la méthode `toXML` comme abstraites. Sinon, retenez que dans toute la suite, il ne sera jamais nécessaire de créer une instance de `Node`, et que la méthode `toXML` sera réimplémentée dans les deux classes héritières.

Noter également que la constante 4 de `tabs`, comme toute constante numérique utilitaire, gagnerait à être définie comme un champ statique de `Node` plutôt que d'être écrite en dur dans le corps d'une méthode.

Exercice 2 Définir une classe `Text` héritière de `Node` munie d'un unique champ de type `String`. Cette classe représentera un nœud texte, son champ représenterant sa chaîne encapsulée.

- Ajouter un constructeur permettant d'initialiser ce champ.
- Réimplémenter la méthode `toXML`. Cette méthode devra renvoyer la chaîne encapsulée, précédée de `indent` × 4 espaces (servez-vous bien sûr de `tabs`), et suivie d'un retour à la ligne.

Exercice 3 Définir une classe `Element` héritière de `Node` munie de deux champs privés : un champ de type `String`, un autre de type `ArrayList<Node>`. Cette classe représentera un nœud élément, les deux champs représentant son nom de balise et la liste de ses fils.

Ajouter un constructeur `Element(String tag)` chargé d'initialiser le nom de balise et de créer la liste des fils (initialement vide).

Exercice 4 Java autorise la définition de constructeurs et méthodes à nombre variable d'arguments. La suite des arguments en nombre variable sera stockée dans un tableau, utilisable tel quel dans le corps du constructeur ou de la méthode. On peut par exemple écrire (noter les triples points après le type des arguments en nombre variable, sans espace) :

```
void m(int a, int b, int... args) {
    System.out.println("a : " + a + ", b : " + b);
    System.out.print("args [" + args.length + "] : ");
    for (int i : args) {
        System.out.print(i + " ");
    }
}
```

Une invocation de la forme `o.m(0, 1, 2, 3, 4)` produira :

```
a : 0, b : 1
args [3] : 2 3 4
```

Ajouter à la classe `Element` les éléments suivants :

- Une méthode `void add(Node... childs)` ajoutant à la liste des fils d'un nœud élément tous les éléments de `childs`.
- Un constructeur `Element(String tag, Node... childs)` invoquant le constructeur de l'exercice 3 pour initialiser la balise d'un nœud élément (`this(...)`), puis ajoutant les éléments de `child` à la liste de ses fils à l'aide de la méthode précédente.

Exercice 5 Implémenter la méthode `String toXML(int indent)` de la classe `Element`. Cette méthode devra construire une chaîne contenant successivement :

1. La balise du noeud élément, précédée de `indent` × 4 espaces encadrée par `"<"` et `">"` (*e.g.* `"<body>"`), suivi d'un retour à la ligne.
2. La concaténation des chaînes renvoyées par une suite d'appels récursifs de `toXML` sur chacun des fils du nœud, avec comme argument `indent + 1`.
3. La balise du noeud élément, précédée de `indent` × 4 espaces encadrée par `"</"` et `">"` (*e.g.* `"</body>"`), suivi d'un retour à la ligne.

Exercice 6 Définir enfin la classe `Document` contenant un unique champ de type `Node`. Cette classe représentera un document XML, son champ représentant son nœud racine, ou valant `null` si le document est vide.

- Ajouter un constructeur `Document(Node root)` permettant de créer un document à partir de son nœud racine.
- Réimplément la méthode `String toString()` en renvoyant une chaîne vide si le document est vide, et sinon, le résultat de `toXML(int indent)` invoquée sur le nœud racine avec une indentation nulle.

Le fragment de code ci-dessous (*c.f.* le fichier `Test.java`) devrait produire très exactement le code source XML présenté en début d'énoncé. Il exploite l'existence de constructeurs à nombre variable d'arguments pour construire un document complet en une seule instruction :

```
Document document =
    new Document(
        new Element("html",
            new Element("body",
                new Text("some text"),
                new Element("em", new Text("emphasized text")),
                new Element("p",
                    new Text("paragraph"),
                    new Element("b", new Text("bold text")
                    ),
                new Text("more text")
            )
        )
    )
  );

System.out.println(document.toXML());
```

Exercice 7 La *taille* d'un contenu est son nombre de nœuds. Par exemple, le contenu du document représenté en début d'énoncé est de taille 10 (5 nœuds éléments, 5 nœuds textes).

— Écrire dans `Node` une méthode (récursive) `int size()` qui, invoquée sur le nœud racine d'un document, renvoie la taille de son contenu.

— Écrire dans `Document` une méthode `int size()` renvoyant 0 si le document est vide, et sinon, la taille de son contenu.

Exercice 8 La *profondeur* d'un contenu vaut 1 + la longeur maximale d'un chemin menant de sa racine à l'un de ses nœuds sans descendants. Par exemple, le contenu du document représenté en début d'énoncé est de profondeur 5.

— Écrire dans `Node` une méthode (récursive) `int depth()` qui, invoquée sur le nœud racine d'un document, renvoie la profondeur de son contenu.

— Écrire dans `Document` une méthode `int depth()` revoyant 0 si le document est vide, et sinon, la profondeur de son contenu.

2 Scanner (optionnel)

Cette section est algorithmiquement moins simple que la précédente. La classe `Scanner` permet de lire le contenu d'un fichier ligne par ligne (elle permet bien d'autres choses, par exemple de séparer en tokens le contenu d'un texte), sur le modèle suivant :

```
// ouverture d'un scanner sur le fichier de nom fileName
Scanner s = null;
try {
    s = new Scanner(new File(fileName));
}
catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}

// tant qu'il reste des lignes,
while (s.hasNextLine()) {
    String line = s.nextLine();  // lire la ligne suivante
    // traitement quelconque de la ligne lue...
}
```

Le but de cette section est de convertir en un document HTML le contenu d'un fichier texte au format suivant : chaque ligne commence par un certain nombre de tabulations, l'*indentation* de cette ligne, suivies d'une certaine chaîne de caractères, le *texte* de cette ligne.

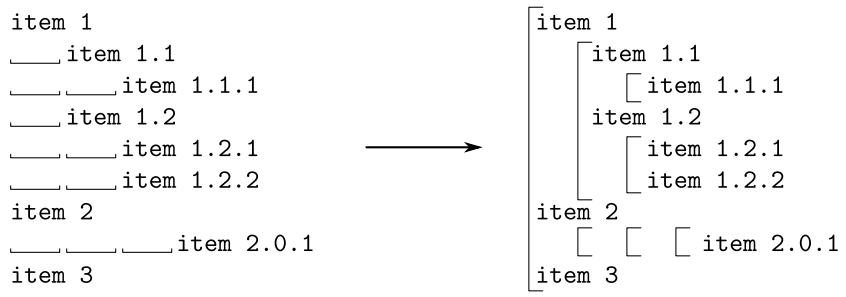
Les textes seront interprétés comme des items (`<li>`) de listes HTML sans numérotation (`<ul>`) imbriquées les unes dans les autres, dans un document respectant les conventions suivantes :

— L'élément `<body>` du document contient une unique liste. Le *niveau* de cette liste est 0. Le niveau d'une liste fille d'une liste de profondeur n vaut $n + 1$.

— Le texte d'une ligne du fichier d'indentation n est un item d'une liste de profondeur n.

— Si deux lignes quelconques du fichier ont la même indentation, et si aucune ligne intermédiaire n'a d'indentation strictement plus petite, alors les textes de ces lignes sont des items de la même liste.

Voici un exemple de la manière dont cet assemblage peut être produit à partir d'un fichier au format texte attendu (`list.txt`, transformé en un document de code source `list.html`) :



Noter le placement de l'item 2.0.1 : avec 3 tabulations de plus que l'item 2, il doit être placé dans une liste de niveau 3, elle-même dans une liste de niveau 2, elle-même dans une liste de niveau 1. Le retour à une tabulation nulle à la ligne suivante place l'item 3 dans la même liste que les items 1 et 2, c'est-à-dire dans l'unique liste de niveau 0.

2.1 Algorithme

L'algorithme que nous utiliserons pour construire la liste de profondeur nulle est le suivant :

1. On crée une pile de noeuds éléments.
2. On crée un noeud élément de balise `"ul"`, et l'on empile ce noeud.
3. Pour chaque ligne du fichier :
   (a) Soit n l'indentation de cette ligne, soit t son texte.
   (b) Tant que $n + 1$ est strictement supérieur à la taille de la pile :
   On crée un nœud élément de balise `"ul"`,
   le nœud créé est ajouté aux fils du noeud au sommet de la pile,
   on empile le nœud crée.
   (c) Tant que $n + 1$ est strictement inférieur à la taille de la pile :
   On dépile le sommet de la pile, en perdant ce sommet.
   (d) On crée un noeud élément de balise `"li"`,
   on crée un noeud texte encapsulant le texte t,
   le noeud texte est ajouté aux fils du nœud `"li"`,
   le noeud `"li"` est ajouté aux fils du nœud au sommet de la pile.
4. On renvoie le nœud crée à l'étape (2).

Le nœud créé à l'étape (2) et renvoyé à l'étape (4) est la liste principale de la construction, celle de profondeur nulle. À l'étape (3), une seule des deux boucles (2.b) et (2.c) est susceptible d'être exécutée. La boucle (2.b) ajoute à la pile de nouvelles listes imbriquées en faisant croître le niveau de la liste au sommet de la pile. La boucle (2.c) fait décroitre le niveau de la liste au sommet de la pile. À l'étape (2.d), le sommet de la pile est une liste de niveau exactement égal à n : c'est dans cette liste que l'on ajoute un nouvel item de texte t.

2.2 Implémentation

Créer une classe utilitaire `Parser`, contenant une unique méthode statique `Document parse(String fileName)`. Cette méthode doit construire le document contenant la traduction du fichier de nom `fileName` à l'aide de l'algorithme précédent. Pour la pile, servez-vous de la classe `Stack`, spécialisée à `Stack<Element>`. Pour extraire n et t, servez-vous des méthodes `charAt` et `substring` de la classe `String`.