Conservatoire National des Arts et Métiers
École SITI - Département Informatique

Année 2015–2016

UE NFP 136 (VARI 2)

Sujet du projet 2015-2016 : Gestion d'un système de fichiers simplifié

Le but de ce projet est d'implémenter en JAVA un système de fichiers simplifié, permettant de stocker des dossiers (répertoires) et fichiers au sein d'une structure arborescente. L'objectif est de faire en sorte que, grâce à l'utilisation de quelques commandes reproduisant des commandes système usuelles, un utilisateur puisse modifier, consulter et parcourir cette arborescence de fichiers. On se limitera ici aux fonctionnalités essentielles : on ne gèrera donc qu'un nombre très restreint de commandes, et on n'associera pas de droits d'accès aux dossiers/fichiers. On souhaite également rendre possible la création d'arborescences de fichiers à partir d'exemples pré-définis, qui seront fournis sous la forme de fichiers texte à récupérer, qui respecteront un certain format. Ce document a pour objectif de vous guider dans les différentes étapes de la réalisation de ce projet, et est à lire attentivement.

Présentation du projet

On s'intéresse dans ce projet à un système de fichiers dit "arborescent" (c'est-à-dire dans lequel les dossiers sont organisés au sein d'une arborescence) de type *Linux* (par exemple). L'utilisation d'une structure arborescente pour stocker des fichiers permet de représenter un ensemble de dossiers contenant des fichiers, certains de ces dossiers étant inclus dans d'autres. Dans une telle structure, il existe un dossier "origine", appelé *racine* (*root* en anglais), dont découlent tous les autres dossiers. (Par convention, le nom de ce dossier sera toujours "" (chaîne de caractères vide).)

En d'autres termes, tout dossier est soit inclus dans ce dossier racine, soit inclus dans un dossier qui est lui-même inclus dans un autre dossier, lui-même inclus dans un autre dossier, etc., lui-même inclus dans le dossier racine. La relation d'inclusion (d'un dossier ou d'un fichier dans un autre dossier) peut se modéliser à l'aide d'une relation "père-fils" dans un arbre (arborescence), dont chaque nœud représente un dossier ou un fichier. Cet arbre est enraciné en le nœud représentant le dossier *racine*, et le seul nœud sans père est donc cette racine.

La première étape consiste à implémenter les opérations liées à la gestion d'une telle arborescence, permettant de stocker des fichiers. La seconde étape consiste à permettre la construction d'un système de fichiers pré-défini

à partir des informations contenues dans un fichier texte respectant un certain format, qui sera détaillé en temps voulu. Enfin, la troisième étape consiste à permettre à un utilisateur d'interagir, par l'intermédiaire de lignes de commandes, avec ce système de fichiers (comme s'il utilisait un terminal pour saisir des commandes Linux), à l'aide d'instructions usuelles (comme `ls`, `cd`, `less`, `mkdir`, `pwd`, `rm`, etc.). Il est à noter que, pour ne pas trop compliquer la gestion de ce système de fichiers, on n'associera pas de droits d'accès (en lecture, en écriture ou en exécution) aux différents dossiers/fichiers.

Étape 1 : la classe `ArbreFichiers`

On se propose dans un premier temps d'écrire une classe `ArbreFichiers`, qui est destinée à représenter une arborescence de fichiers (ou bien un nœud d'une telle arborescence). On utilisera pour cela une représentation des arbres à l'aide de références (ou "chaînages") proche de celle vue en cours, mais en ayant recours à plus d'informations que dans cette représentation, pour permettre un parcours plus aisé de l'ensemble des dossiers/fichiers.

Ainsi, chaque instance/objet de cette classe représentera un nœud de l'arborescence, et contiendra les 8 variables d'instance suivantes (on rappelle que le bon usage en JAVA est de déclarer ces variables comme `private`) :

1. Quatre variables de type `ArbreFichiers`, correspondant aux nœuds suivants de l'arborescence : le père de ce nœud, son premier fils (fils le plus à gauche) et ses frères gauche et droit. Il convient de noter que les dossiers/fichiers représentés par les fils d'un nœud seront stockés par ordre alphabétique, de gauche à droite.

2. Une variable de type `String` contenant **le nom du dossier/fichier** représenté par ce nœud. On supposera, pour éviter toute complication inutile, que le nom d'un dossier/fichier ne peut contenir aucun espace.

3. Une variable booléenne indiquant si ce nœud représente un fichier (si elle vaut `true`) ou un dossier (si elle vaut `false`).

4. Une variable de type `String` dans laquelle sera stocké le contenu associé à ce nœud, si ce dernier représente un fichier. Dans le cas où ce nœud représente un dossier, cette variable sera égale à `null`.

5. Une variable entière dont la valeur correspond à **la taille** (en octets) du dossier/fichier correspondant à ce nœud. Naturellement, la taille d'un fichier est égale au nombre de caractères de son contenu (obtenu via la méthode `length` de la classe `String`), et la taille d'un dossier est la somme des tailles des éléments (dossiers et fichiers) qu'il contient.

Cette classe contiendra également les accesseurs en lecture qui seront nécessaires (notamment pour accéder au nœud père, à la nature d'un nœud -dossier ou fichier-, et au contenu d'un fichier), et un (ou des) constructeur(s).

Enfin, cette classe contiendra (au moins) les 5 méthodes suivantes, qui permettront de modifier et parcourir l'arborescence de fichiers considérée :

- Méthode 1 : elle permet d'ajouter comme fils d'un nœud donné un autre nœud (ou un morceau d'arborescence) passé en paramètre.
- Méthode 2 : elle permet de supprimer un nœud de l'arborescence.
- Méthode 3 : elle renvoie, sous la forme d'une variable de type `String`, l'ensemble des informations (une lettre égale à 'd' comme dossier ou 'f' comme fichier, ainsi que le nom et la taille en octets) concernant les dossiers et fichiers inclus dans le dossier représenté par un nœud.
- Méthode 4 : elle renvoie, sous la forme d'une variable de type `String`, les noms des nœuds situés sur la branche reliant la racine à un nœud.
- Méthode 5 : elle renvoie une référence vers le nœud dont le nom (chemin relatif) est passé en paramètre via une variable de type `String`.

Voici quelques indications pour implémenter ces méthodes :

- **Méthode 1**. D'abord, il faut faire du nœud n_1 sur lequel la méthode est appelée le père du nœud n_2 passé en paramètre.

  Il faut ensuite parcourir les fils du nœud n_1 : s'il n'a aucun fils, alors le nœud n_2 devient son unique fils, mais dans le cas contraire il faut comparer le nom de chaque fils avec le nom du nœud n_2 à ajouter, pour trouver l'emplacement correct pour le nœud n_2.

  On peut utiliser pour cela la méthode `compareToIgnoreCase` de la classe `String`, qui prend en paramètre une variable de type `String`, et renvoie un entier > 0 si la variable de type `String` sur laquelle la méthode est appelée se situe après (dans l'ordre alphabétique) celle passée en paramètre, et un entier ≤ 0 sinon.

  Une fois le bon emplacement déterminé, il suffira d'y insérer n_2 (en mettant à jour, si besoin, le premier fils de n_1, et les frères gauche et/ou droit de n_2 et de ses nouveaux frères gauche et droit, s'ils existent).

  Enfin, il ne faut pas oublier de mettre à jour la taille du nœud n_1 (qui a été modifiée suite à l'ajout du nœud n_2), puis de son père, puis du père de son père, etc., jusqu'à atteindre la racine.

- **Méthode 2**. Il faut d'abord différencier le cas où le nœud n_1 à supprimer (c'est-à-dire le nœud sur lequel la méthode est appelée) est le

premier fils de son père (il faut alors mettre à jour cette information, puisque son frère droit devient le premier fils) de l'autre cas (où il suffit de mettre à jour le frère droit du frère gauche de n_1).

Dans les deux cas, il faut également mettre à jour le frère gauche du frère droit de n_1 (si ce dernier existe). Enfin, comme dans la méthode précédente, il ne faut pas oublier de mettre à jour la taille du père du nœud n_1 (qui a été modifiée suite à la suppression du nœud n_1), puis du père de son père, etc., jusqu'à atteindre la racine.

- **Méthode 3**. Il suffit d'examiner, de gauche à droite, l'ensemble des fils du nœud sur laquelle la méthode est appelée, à l'aide d'un parcours de liste. Pour chaque fils, on mettra à jour la variable de type `String` à renvoyer en concaténant ses informations avec celles des fils précédents. Il faut noter que la chaîne ainsi construite servira à l'implémentation de la commande `ls` (voir l'étape 3 pour plus de détails).

- **Méthode 4**. Il faut parcourir les nœuds se trouvant entre la racine et le nœud sur lequel la méthode est appelée, en commençant par ce dernier, et en remontant ensuite vers son père, puis le père de son père, etc., jusqu'à atteindre la racine. Comme dans la méthode précédente, en chaque nœud parcouru, on mettra à jour la variable de type `String` à renvoyer en concaténant son nom avec ceux des nœuds précédents.

  Il faut noter que la chaîne ainsi construite servira à l'implémentation de la commande `pwd` (voir l'étape 3 pour plus de détails).

- **Méthode 5**. La variable de type `String` passée en paramètre est soit ".." (qui signifie "remonter vers le dossier père"), soit le nom d'un des dossiers inclus dans le dossier représenté par le nœud sur lequel la méthode est appelée.

  Si sa valeur est "..", alors on renvoie une référence vers le père du nœud sur lequel la méthode est appelée. Sinon, il faut parcourir les fils du nœud sur lequel la méthode est appelée, jusqu'à trouver celui dont le nom correspond au nom passé en paramètre : on renvoie alors une référence vers ce fils. Par ailleurs, il faut être vigilant pour prendre en compte le cas où le nom spécifié ne correspond pas à un dossier existant, qui peut générer des erreurs lors de l'exécution.

Étape 2 : construction d'une arborescence de fichiers initiale à partir d'un fichier texte

La méthode décrite ici peut être intégrée à votre classe principale. Son objectif est de lire les informations contenues dans le fichier texte fourni en paramètre, et de les utiliser pour créer l'arborescence de fichiers initiale

avec laquelle l'utilisateur va pouvoir interagir. Voici quelques indications concernant le format du fichier à lire :

- Le fichier débute par le mot-clé **racine** et se termine par le mot-clé **fin**. Au début, le dossier "courant" est le dossier "racine".
- Certaines lignes contiennent des commentaires à la fin, séparés du reste de la ligne par un espace puis un caractère "%".
- Toute ligne, sauf la première et la dernière, qui ne correspond pas au contenu d'un fichier, commence par un bloc de un ou plusieurs caractères "*" (sans espace), et correspond alors soit à la description d'un nouveau dossier/fichier, soit à la fermeture d'un dossier.
- Une ligne qui contient la description d'un nouveau dossier de nom "un_nom" a le format suivant (sans tenir compte des "*") :

  un_nom d %commentaires éventuels

  Ce dossier est inclus dans le dossier "courant" actuel. En outre, il devient le nouveau dossier "courant" jusqu'à nouvel ordre (c'est-à-dire jusqu'à ce qu'il soit refermé, ou qu'une autre ligne contenant la description d'un nouveau dossier soit rencontrée).
- Une ligne qui contient la description d'un nouveau fichier de nom "un_nom" a le format suivant (sans tenir compte des "*") :

  un_nom f %commentaires éventuels

  Dans ce cas, la ligne qui suit immédiatement cette ligne représente à chaque fois le texte contenu dans ce fichier. En outre, ce fichier est inclus dans le dossier courant.
- Une ligne correspondant à la fermeture d'un dossier contient un seul mot (sans tenir compte des "*" ou des commentaires) : le mot `fin`.

  Quand une telle ligne est rencontrée, le dossier courant est refermé, et le dossier représenté par son père (c'est-à-dire le dossier qui le contient) devient le nouveau dossier courant.

Voici un exemple d'un tel fichier fourni (appelons-le "toto.txt") :

```
racine
* un_fichier f
ceci constitue le contenu de ce fichier
* sd1 d %ceci est un commentaire : le dossier sd1 a pour pere la racine
** un_autre_fichier f %cet autre fichier est inclus dans sd1
ceci constitue le contenu de cet autre fichier
```

```
** sd2 d %le dossier sd2 est inclus dans sd1
*** un_3e_fichier f %ce fichier est inclus dans sd2
ceci constitue le contenu de ce 3e fichier
** fin %commentaire : on ferme sd2, et on revient dans sd1
** un_dernier_fichier f %ce fichier est lui aussi inclus dans sd1
ceci constitue le contenu de ___ ce dernier fichier
* fin %on ferme sd1, et on revient a la racine
* sd3 d %ce dossier a pour pere la racine, et ne contient aucun fichier
* fin %on ferme sd3, et on revient a la racine
fin
```

Ce fichier correspond à une arborescence de fichiers dans laquelle :

- La racine contient un fichier (`un_fichier`) et 2 dossiers (`sd1` et `sd3`).
- Le dossier `sd1` contient 2 fichiers (`un_autre_fichier` et `un_dernier_fichier`) et un dossier (`sd2`).
- Enfin, le dossier `sd2` contient un fichier (`un_3e_fichier`).

Pour lire des données dans un fichier, on peut utiliser une variable de la classe `BufferedReader`. Voici comment initialiser une variable `lecteur` de cette classe, pour accéder au contenu du fichier de nom "toto.txt" :

```
BufferedReader lecteur = new BufferedReader(new FileReader("toto.txt"));
```

En appelant la méthode `readLine` (qui renvoie une valeur de type `String`) sur cette variable, on lira le contenu de la ligne suivante du fichier de nom "toto.txt" (la lecture d'un fichier étant séquentielle). Ainsi, dans le cas du fichier décrit ci-dessus, après le premier appel à cette méthode, la variable `ligne` contiendra la valeur `"racine"`. Après un deuxième appel à cette méthode, la variable `ligne` contiendra la valeur `"* un_fichier f"` :

```
String ligne = lecteur.readLine(); //ligne contient "racine"
ligne = lecteur.readLine(); //ligne contient "* un_fichier f"
```

La construction d'une variable de type `BufferedReader` est susceptible de générer une exception (si le fichier de nom "toto.txt" n'existe pas, par exemple), et il serait donc préférable de gérer toutes ces instructions dans un bloc `try{...} catch(Exception e){...}`. Enfin, après utilisation, cette variable `lecteur` doit être fermée, à l'aide de l'instruction `lecteur.close();`.

Pour lire le fichier "toto.txt", il est également possible d'utiliser la classe `Scanner`. On lit alors chaque ligne à l'aide de la méthode `nextLine()` :

```
Scanner lecteur2 = new Scanner(new File("toto.txt")); //création
String ligne = lecteur2.nextLine(); //premier appel = première ligne lue
ligne = lecteur2.nextLine(); //deuxième appel = deuxième ligne lue
```

Une fois les informations d'une ligne associée à un dossier/fichier récupérées, il faudra les stocker dans une variable de la classe `ArbreFichiers`. Pour cela, on vous rappelle l'existence de :

- La méthode `split()` de la classe `String`, qui permet de "découper" la variable de type `String` sur laquelle la méthode est appelée en fonction du séparateur passé en paramètre (par exemple `" "`), et qui stocke les variables de type `String` résultant de ce découpage dans un tableau.
- La méthode `charAt()` de la classe `String`, qui renvoie le caractère qui se trouve, dans la variable de type `String` sur laquelle la méthode est appelée, à l'indice (entier) passé en paramètre.

(Pour plus d'information sur ces méthodes, se référer à l'énoncé du TP 3.)

Étape 3 : implémentation de commandes système basiques

Cette étape peut être réalisée dans une méthode appelée dans le `main`, ou bien directement dans le `main`. L'objectif est ici de faire saisir par l'utilisateur des commandes système de type Linux, puis de les exécuter, ce qui consiste essentiellement à appeler les méthodes définies à l'étape 1. Les 8 commandes (9 en réalité) à considérer dans le cadre de ce projet sont les suivantes :

1. `ls` : cette commande n'attend aucun argument, et a pour effet de lister l'ensemble des dossiers et fichiers (noms et tailles) inclus dans le dossier courant (par ordre alphabétique, puisqu'on rappelle qu'ils sont stockés dans cet ordre). Les dossiers apparaîtront précédés de la lettre 'd' (comme dossier). Ainsi, si le dossier courant est le dossier `sd1` de l'exemple donné à l'étape 2, la commande `ls` affichera ceci :

   ```
   > ls
   d sd2 42
     un_autre_fichier 46
     un_dernier_fichier 51
   ```

2. `cd` : cette commande prend en argument un nom (chemin relatif) d'un autre dossier inclus dans le dossier courant (ou bien la chaîne `".."`, qui permet de remonter vers le dossier père), et a pour effet que cet autre dossier devient le nouveau dossier courant. Ainsi, si le dossier courant est le dossier `sd2` de l'exemple donné à l'étape 2, les 3 commandes successives suivantes feront de `sd3` le nouveau dossier courant :

   ```
   > cd ..
   > cd ..
   > cd sd3
   ```

Si le nom indiqué n'est pas celui d'un dossier existant, la commande aura pour seul effet d'afficher un message le signalant à l'utilisateur.

3. `mkdir` : cette commande prend en argument un nom de dossier (sans espaces), et crée un nouveau dossier vierge, qui porte ce nom et est inclus dans le dossier courant. Ainsi, si le dossier courant est le dossier `sd3` de l'exemple donné à l'étape 2, la commande `mkdir sd4` créera un nouveau dossier `sd4`, initialement vide, et inclus dans le dossier `sd3`.

4. `mkfile` : cette commande prend en argument un nom de fichier (sans espaces), et crée un nouveau fichier, qui porte ce nom et est inclus dans le dossier courant. L'utilisateur est alors invité à saisir le contenu de ce fichier (l'appui sur la touche "retour" met fin à la saisie) : s'il désire saisir un contenu sur plusieurs lignes, il lui suffit de saisir 3 caractères `"_"` consécutifs à la fin de chaque ligne du fichier.

5. `less` : cette commande prend en argument le nom d'un fichier inclus dans le dossier courant, et affiche son contenu (rappelons que les `"___"` apparaissant dans le fichier doivent être remplacés par des retours à la ligne). Ainsi, si le dossier courant est le dossier `sd1` de l'exemple donné à l'étape 2, les commandes successives `mkfile nouveau_fichier` (qui crée le fichier `nouveau_fichier` dans le dossier courant) et `less nouveau_fichier` produiront l'affichage suivant :

```
> mkfile nouveau_fichier
 Contenu du fichier ? Voici ___le texte saisi___ par l'utilisateur

> less nouveau_fichier
Voici
le texte saisi
 par l'utilisateur
```

   Dans le cas où aucun fichier du dossier courant ne porte le nom passé en argument, la commande `less` aura pour seul effet d'afficher un message le signalant à l'utilisateur.

6. `pwd` : cette commande n'attend aucun argument, et provoque l'affichage du chemin absolu associé au dossier courant. Ainsi, si le dossier courant est le dossier `sd3` de l'exemple donné à l'étape 2, les commandes successives `cd ..`, `cd sd1`, `cd sd2`, `pwd`, produiront l'affichage suivant :

```
> cd ..
> cd sd1
> cd sd2
> pwd
/sd1/sd2
```

7. `rm` : cette commande prend en argument le nom d'un dossier ou d'un fichier inclus dans le dossier courant, et le supprime (lui et tout ce qu'il contient, si c'est un dossier). Ainsi, si le dossier courant est le dossier `sd1` de l'exemple donné à l'étape 2, les commandes successives `rm sd2` et `ls` produiront l'affichage suivant :

   ```
   > rm sd2
   > ls
     un_autre_fichier 46
     un_dernier_fichier 51
   ```

   Dans le cas où aucun dossier ou fichier inclus dans le dossier courant ne porte le nom passé en argument, la commande `rm` aura pour seul effet d'afficher un message le signalant à l'utilisateur.

8. `quit` et `exit` : ces commandes, qui n'attendent aucun argument et produisent toutes deux le même effet, mettent fin à la session en cours. Dans le cadre de ce projet qui ne fait que simuler le comportement de commandes Linux, elles mettent donc fin à l'exécution du programme.

Touches finales

Pour compléter le projet, il reste à écrire le programme principal. Ce dernier récupérera un seul paramètre au moment de l'appel du programme : le nom du fichier décrivant l'arborescence de fichiers à utiliser. Si rien n'est passé en paramètre, alors l'arborescence de fichiers initiale sera vide (c'est-à-dire ne contiendra qu'un seul nœud, représentant le dossier racine).

Il faudra ensuite, au fur et à mesure de la lecture de ce fichier (étape 2), construire cette arborescence à l'aide des méthodes de la classe décrite à l'étape 1. Le reste du programme principal se résume à gérer une variable de type `ArbreFichiers` qui contiendra une référence vers le nœud représentant le dossier courant (initialement, ce sera la racine), ainsi qu'une boucle dans laquelle on affichera une invite de commande sommaire (`>`) et on lira la commande saisie par l'utilisateur (si une commande produit un affichage, alors on sautera une ligne à la fin de cet affichage), et qui s'exécutera tant que l'utilisateur ne saisit pas une commande mettant fin au programme (`quit` ou `exit`). S'il saisit n'importe quelle autre commande valide, alors on exécutera les opérations qui y sont associées (étape 3). S'il saisit une commande invalide, alors on se contentera d'afficher un message le lui signalant.

Le projet est à rendre par e-mail, à l'adresse : `cedric.bentz@cnam.fr`. Lors du rendu, il faudra bien évidemment fournir le code source commenté de toutes vos classes (fichiers `.java`), et leur bytecode (fichiers `.class`).