TP 5 : Les flux (entrées/sorties) (A RENDRE II)

TP4 et TP5 sont à rendre ensemble dans UN projet java le 22 mai 2016, à ma@lri.fr.

Attention :
1. Créez un projet java nommé **app3.java.Nom** où Nom est votre nom.
2. vous rendrez une archive compressée au format **app3.java.Nom.zip.** Pour créer cette archive, vous cliquez File -> Export -> General -> Archive File -> Next, et vous rentrez le nom demandé.

Objectifs : Dans ce TP, nous verrons comment lire et écrire des flux de données, avec comme application la manipulation de fichiers.

Les entrées/sorties en Java sont représentées de telle façon qu'il est possible, à un certain niveau d'abstraction, de traiter exactement de la même façon des données, qu'elles viennent d'un fichier local ou d'un flux de données transitant sur le réseau. Pour cela, Java représente des « flux » qui sont divisés en quatre catégories :

- Les flux d'entrée (input stream) et de sortie (output stream)
- Les flux binaires (traitement au niveau des octets) et texte

Il existe donc quatre types de flux entrant, résultant de la combinaison de ces deux critères. Ces quatre types de flux se traduisent par **quatre suffixes** dans les noms des classes de lecture/écriture :

Flux d'entrée binaire : InputStream	Flux d'entrée texte : Reader
Flux de sortie binaire : OutputStream	Flux de sortie texte : Writer

On peut donc savoir, par exemple, qu'une instance de classe « BufferedWriter » est un flux sortant texte. De façon analogue, les entrées et sorties standard (System.in et System.out) sont de type InputStream et OutputStream, on peut donc en conclure qu'ils sont de type binaire, c'est pourquoi il faut utiliser une sorte de convertisseur (dans nos TP la classe Scanner) pour en récupérer du texte.

Les flux peuvent porter sur plusieurs types de données différents, qui se traduisent par autant de **préfixes**. On peut citer par exemple le préfixe « File » qui traite des fichiers, « String » qui porte sur des chaînes de caractères ou « ByteArray » et « CharArray » qui portent sur des tableaux d'octets et de caractères respectivement. Il existe également « Object » et « Pipe » que nous ne verrons pas ici. **L'association d'un préfixe et d'un suffixe donne une classe permettant de traiter un flux** : FileInputStream permet de lire un fichier binaire, StringWriter écrit dans une chaîne de caractères, etc. Il existe quelques exceptions : par exemple ByteArray n'existe que pour les flux binaires, de même que CharArray n'existe que pour les flux de type texte.

Enfin, il est possible de rajouter des fonctionnalités à un flux, ce qui est représenté également par un préfixe. On peut citer par exemple *Buffered* qui met les données en tampon lors de la lecture ou l'écriture pour améliorer les performances, *LineNumber* pour lire (uniquement) des données de type texte en gardant le numéro de ligne en mémoire ou encore *PushBack* qui permet de faire comme si des caractères n'avaient pas été lus. L'ajout de ce type de fonctionnalité est réalisé à l'aide du *design*

pattern nommé « decorator » : chaque « decorator » garde la même interface que l'objet qu'il décore et on peut ajouter plusieurs décorateurs les uns derrière les autres.

Exercice 1 : Lecture d'un fichier

1. Créez une classe TP5Exercice1. Créez une méthode principale qui demande à l'utilisateur d'entrer un nom de fichier (par exemple « c:\windows\win.ini »), ouvre le fichier et affiche son contenu à la console.
   Pour représenter le fichier, vous instancierez java.io.File. Vous pouvez tester si le fichier existe grâce à la méthode *File.exists*. Pour ouvrir le fichier, créez une instance de FileReader en lui passant votre instance de fichier (attention à bien traiter les exceptions). Afin de pouvoir lire le fichier, vous devrez créer une instance de BufferedReader et lui passer votre instance de FileReader, ce qui vous permettra d'utiliser la méthode *readLine* (qui renvoie *null* lorsqu'il n'y a plus de ligne à lire).
   Voici un exemple de résultat attendu :

```
Veuillez entrer un nom de fichier :
c:\windows\win.ini
Contenu du fichier :
Ligne : ; for 16-bit app support
Ligne : [fonts]
Ligne : [extensions]
Ligne : [mci extensions]
```

2. Que se passe-t-il lorsque le fichier n'existe pas ? Faites attention à bien traiter les erreurs.
3. Remplacez le BufferedReader par un LineNumberReader. Voici un exemple de résultat attendu :

```
Veuillez entrer un nom de fichier :
c:\windows\win.ini
Contenu du fichier :
Ligne 1 : ; for 16-bit app support
Ligne 2 : [fonts]
Ligne 3 : [extensions]
Ligne 4 : [mci extensions]
```

4. Modifiez votre programme de façon à afficher tous les fichiers d'un dossier et non un simple fichier. Pour cela, utilisez File et testez si le dossier est valide grâce à la méthode isDirectory. Vous pouvez lister les fichiers d'un dossier grâce à la méthode *listFiles*. Devant le nom des éléments du dossier, affichez un « F » si c'est un fichier et un « D » si c'est un dossier.

Exercice 2 : Ecriture d'un fichier

1. Créez un fichier nommé « prenoms.txt » contenant une liste de prénoms dans le désordre.
2. Créez une nouvelle classe contenant une méthode principale. Faites en sorte que la méthode principale liste le fichier « prenoms.txt », charge son contenu dans une ArrayList<String>, trie cette liste (en utilisant Collections.sort) et sauvegarde le contenu dans un nouveau fichier nommé « prenoms_out.txt ». Testez.
   Vous ferez attention à entrer le nom complet du dossier dans lequel votre fichier se trouve.

Exercice 3 : Sujet libre

1. Créez un package nomé **app3.tp6.exerciceexercise3** dans votre projet app3.java.Nom qui présent un sujet de votre choix concernant une Class, une API, un package, un framework ou un outil lié à Java, par exemple design patterns, et qui n'est pas vu dans les TPs précents.
2. Créez une nouvelle classe nomée **MonSujetLibreTest.java** contenant une méthode principale qui montre l'utilistation de votre choix.
3. Ajoutez un fichier **rapport.txt** dans votre projet pour expliquer brièvement sur le choix et vos codes.