PRÁCTICA 1

Estructuras de datos lineales de la Java Collections Framework

Objetivos

- Repasar las estructuras de datos lineales más comunes de la JCF y sus métodos asociados.
- Repasar los *iteradores* asociados a estructuras de datos.
- Repasar los métodos asociados a la clase `String` y su uso.
- Repasar la lectura y escritura en archivos de texto.
- Repasar el tratamiento de excepciones en Java.
- Crear nuevas estructuras de datos, resultado de la composición de otras estructuras de datos ya existentes.
- Resolver problemas sencillos con dichas estructuras de datos.

Requerimientos

Para superar esta práctica se debe realizar lo siguiente:

- Dominar las estructuras de datos lineales `ArrayList` y `LinkedList`, y la combinación de ellas para la resolución de problemas sencillos.
- Contestar adecuadamente a las preguntas que se proponen y en el archivo correspondiente.
- Pasar los respectivos test que se proporcionan.
- Entrega de todo el material en el repositorio en la fecha acordada.

Enunciado

1. **Ejercicio con colecciones lineales**. Editor de líneas.

Un **editor de líneas** es un programa que manipula texto, línea por línea. Hubo un tiempo en que este tipo de editores eran los que se utilizaban en diferentes sistemas operativos como MSDOS (edlin), UNIX (ed), etc. Pero con la llegada de los editores de pantalla completa (el cursor se mueve a la línea que desea editar), los editores de líneas dejaron de utilizarse. Linux / Unix y Windows todavía tienen un editor de líneas, pero sólo se suelen utilizar después de una caída del sistema.

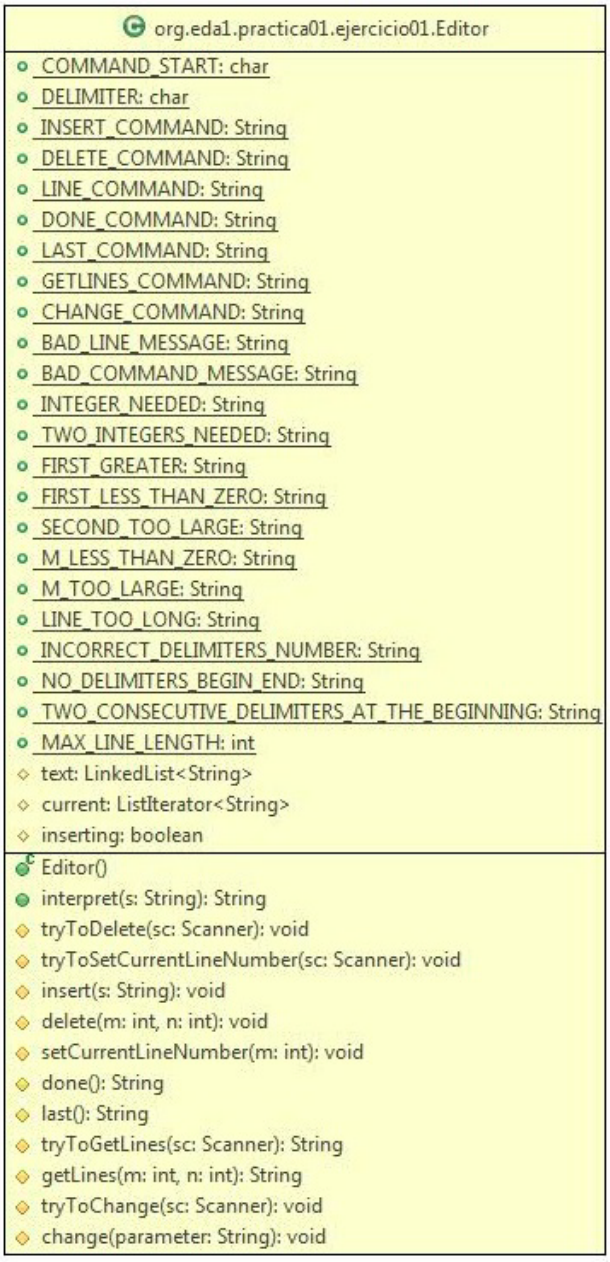
Suponemos que cada línea del editor tiene como máximo *80* caracteres. La primera línea del texto se considera como línea 0, y una de las líneas que lo forman se designa como *línea actual*. Para la implementación de la estructura de datos que gestionará el texto, estará basada en **`LinkedList<String>`** y para determinar qué línea es la actual, utilizaremos un **`ListIterator<String>`**.

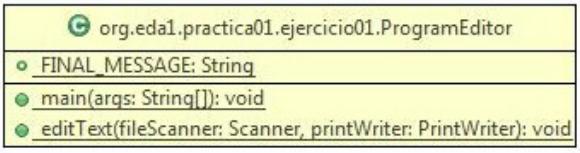
Para gestionar el texto mediante este tipo de editores vamos a tener siete comandos de edición, cuyo funcionamiento pasamos a explicar a continuación.

- **$Insert** Inserta en el texto todas las líneas que haya hasta el comando de edición siguiente. Si existe una línea actual establecida, cada línea se insertará antes de dicha línea actual. De lo contrario, cada una de las líneas se insertan al final del texto, considerando como línea actual la línea ficticia justo después de la última línea del texto.
- **$Delete m n** Se eliminarán todas y cada una de las líneas en el texto entre líneas **m** y **n**, ambas inclusive. La línea actual se convertirá entonces, en la primera línea justo después de la última línea eliminada. Así que si se elimina la última línea del texto, la línea actual será línea ficticia justo después de la última línea del texto. También, debe generar un mensaje de error si (1) **m** es mayor que **n**, o si (2) **m** es menor que **0**, o si (3) **n** es mayor que el último número de línea en el texto, o por último (4) los parámetros **m** y **n** del comando deben ser números enteros.
- **$Line m** La línea **m** se convierte en la línea actual. Debe de mostrarse un mensaje de error si **m** es 0 o menor que 0, o mayor que el número de líneas en el texto. También debe controlarse el caso de que no se introduzca ningún número entero, **m**, asociado como parámetro a este comando de edición, o que **m** sea menor que 0 o mayor que el último número de línea del texto.
- **$Done** Este comando termina la ejecución del editor de líneas e imprime el texto completo. No altera ni el texto ni la línea actual.
- **$Last** Este comando devuelve el número de línea que tiene la última línea en el texto. No altera ni el texto ni la línea actual.
- **$GetLines m n** Devuelve el número de línea y el contenido de dicha línea, desde la línea **m** hasta la línea **n**, ambas inclusive. Es decir, éste es un comando que devuelve el número de líneas y el contenido de las mismas de todo el texto. No altera ni el texto ni la línea actual. Al igual que ocurre con el comando *$Delete*, se debe generar un mensaje de error si (1) **m** es mayor que **n**, o si (2) **m** es menor que **0**, o si (3) **n** es mayor que el último número de línea en el texto, o por último (4) los parámetros **m** y **n** del comando deben ser números enteros.
- **$Change %X%Y%** Este comando realiza cambios en la línea actual, y éste consiste en que cada ocurrencia del String dado por **X** se reemplazará por el String dado por **Y**. Destacar que el String **Y** puede ser la cadena vacía %of %%. Se deberá generar un error si el delimitador (%) ocurre menos de tres veces en el parámetro del comando, si no hay un delimitador al principio y al final, y también en el caso de que haya dos delimitadores consecutivos al principio.

Es interesante recordar que se debe imprimir un mensaje de error para cualquier comando no permitido del tipo *$End*, *$insert*, *Insert*, etc.

Una vez controlada toda la sintaxis del editor de líneas hay que implementar un sencillo intérprete de comandos que procese cada línea y actúe de acuerdo con lo que debe de ejecutar cada comando con sus respectivos parámetros.





Adicionalmente, contestar a las dos siguientes cuestiones.

- De las principales colecciones lineales implementados en la JCF de Java (**`ArrayList`**, **`LinkedList`**, **`Vector`**, **`Stack`**, etc.), razone detalladamente cuál es la que cree más conveniente para resolver este ejercicio.
- Para la resolución de este ejercicio se ha propuesto utilizar el **`LinkedList`**, ¿sería muy complicado realizar la misma implementación con **`ArrayList`**?, ¿qué habría que hacer?.

Las respuestas a estas preguntas se entregarán en un archivo PDF (memoria de la práctica 1) con nombre **practica_01.pdf** y se almacenará en una carpeta denominada **Memorias\Practica_01** en el proyecto de prácticas de cada alumno.

2. Ejercicio con colecciones lineales un poco más complejas. Ciudades donde empresas de software desarrollan sus proyectos.

Se dispone de la siguiente información para gestionarla en una estructura de datos: Empresas de software, proyectos que desarrollan y lugares donde tienen las sedes de dicho proyectos. Y como archivo de entrada se propone el siguiente (empresasProyectosCiudades.txt):

```
Adobe Photoshop San Antonio
Microsoft Word Washington
Ramsoft EZJava New_York
Microsoft VisualC++ Stanford
Borland Delphi Jackson
Microsoft Excel Sacramento
Adobe Flash Charleston
Borland C++Builder Ohio
Microsoft VisualC++ Philadelphia
Adobe Illustrator Miami
Microsoft Word New_York
Borland JBuilder Miami
Microsoft VisualC++ Miami
Borland JBuilder Tucson
Microsoft Word Orlando
Adobe Photoshop Houston
Borland Delphi Detroit
Microsoft Word Miami
Ramsoft EZJava Stanford
Microsoft Excel Los_Angeles
Adobe Illustrator Sacramento
Microsoft Excel Phoenix
Borland C++Builder Portland
Microsoft Word Memphis
Adobe Flash Boston
Microsoft Excel San_Francisco
Borland C++Builder Berkeley
Microsoft VisualC++ Washington
Borland C++Builder Wisconsin
Microsoft VisualC++ New_York
Borland JBuilder Santa_Fe
Microsoft Word Maryland
Borland JBuilder Denver
Adobe Flash Washington
Borland Delphi Chicago
Microsoft Excel Las_Vegas
Borland Delphi Milwaukee
Adobe Illustrator New_Orleans
Borland Delphi Miami
Adobe Photoshop Seattle
Ramsoft EZJava Washington
Borland C++Builder Washington
```

Como **EJERCICIO** se pide lo siguiente: implementar un programa en Java que lea desde un archivo de entrada la anterior lista de **Empresa_Software Proyecto_Software Ciudad_Donde_Se_Desarrolla**, almacenando en una estructura de datos basada en **ArrayList** de la JCF.

- Según se lee cada línea del archivo (**Empresa_Software Proyecto_Software Ciudad_Donde_Se_Desarrolla**), el programa debe comprobar si dicha tripleta está ya en el contenedor. Si lo está, no hacer nada pues será una línea repetida; en caso contrario, se inserta en el contenedor. Conforme se van insertando los elementos en el **`ArrayList`**, no es necesario que permanezcan ordenados.
- Una vez que todas las líneas (tripletas) han sido leídas del archivo de entrada y almacenadas en memoria en la estructura de datos basada en **`ArrayList`**, se generará como resultado un archivo de salida tal y como se indica a continuación.

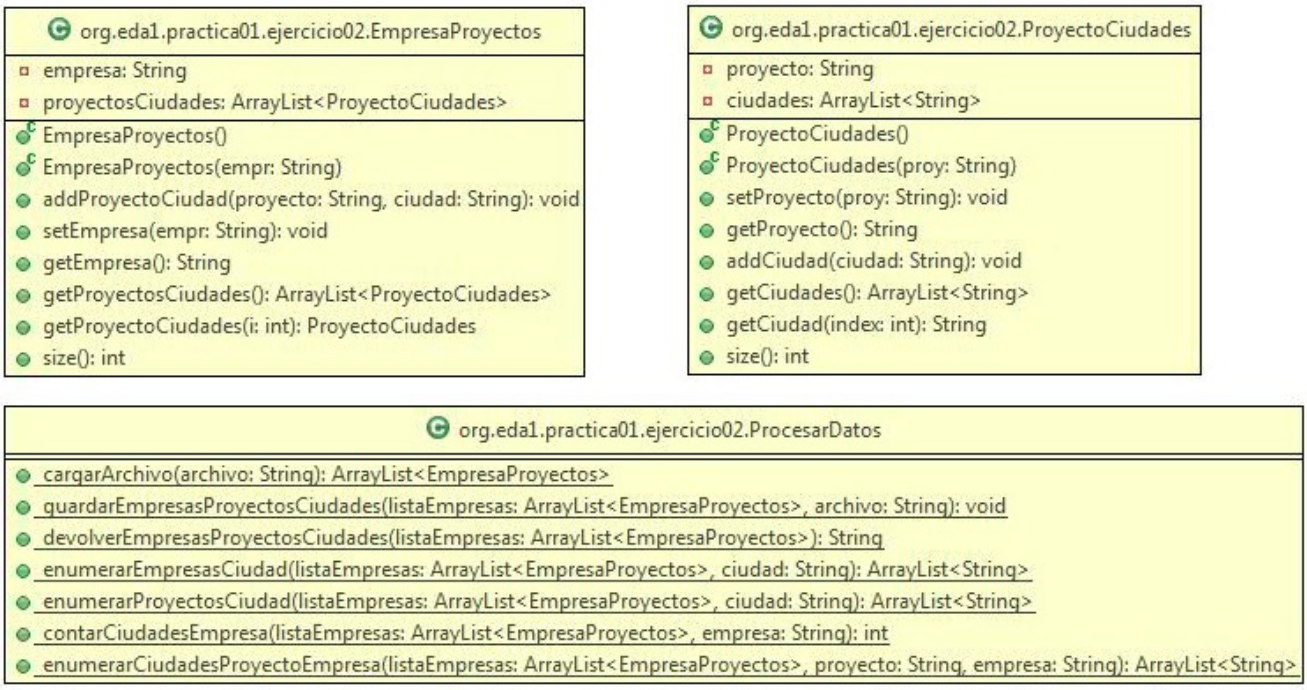
```
Adobe: Photoshop<San_Antonio, Houston, Seattle>; Flash<Charleston, Boston, Washington>; Illustrator<Miami, Sacramento, New_Orleans>
Microsoft: Word<Washington, New_York, Orlando, Miami, Memphis, Maryland>; VisualC++<Stanford, Philadelphia, Miami, Washington, New_York>; Excel<Sacramento, Los_Angeles, Phoenix, San_Francisco, Las_Vegas>
Ramsoft: EZJava<New_York, Stanford, Washington>
Borland: Delphi<Jackson, Detroit, Chicago, Milwaukee, Miami>; C++Builder<Ohio, Portland, Berkeley, Wisconsin, Washington>; JBuilder<Miami, Tucson, Santa_Fe, Denver>
```

Como sugerencia para la realización del ejercicio, podemos plantear una estructura de datos basada en **`ArrayList`**. De forma genérica podría plantearse lo siguiente (lista de listas de listas)

```
ArrayList(Empresas, ArrayList(Proyectos, ArrayList(Ciudades)))
```

Además, una vez en organizados los datos en la ED indicada anteriormente, se debe responder (con la implementación de la correspondiente función) a las siguientes ***consultas***:

- Enumerar las empresas que tienen su sede en la ciudad Miami.
- Enumerar el número de *proyectos* con sede en Washington.
- ¿En cuántas ciudades diferentes se desarrollan proyectos de Microsoft?.
- Devolver las ciudades donde desarrollándose el proyecto *Word*, se desarrollan a su vez proyectos de otras empresas distintas de *Microsoft*.



Para finalizar, como ya hemos podido comprobar, para la resolución de este ejercicio se ha propuesto utilizar el **`ArrayList`**. ¿Sería apropiado realizar la misma implementación con **`LinkedList`**?, ¿qué habría que hacer?. Contestar a estas preguntas en el archivo PDF correspondiente a la práctica 1 (**practica_01.pdf**), separando convenientemente las respuestas de los diferentes ejercicios que componen la práctica.