PROGRAMACIÓN DE SERVICIOS Y PROCESOS – TAREA 1

Ejercicio 01:

Para el primer ejercicio se han creado dos clases:

- GenerarPalabras.java
- OrdenarPalabras.java

A continuación, se describe la clase **GenerarPalabras**, que permite genera palabras aleatorias mediante una ejecución de un código. Las palabras que se generan no son palabras reales, son ficticias, aunque son legibles.

Librerías importadas para poder generar las palabras.

- **Random:** esta librería permite generar valores aleatorios en java.

Para poder controlar las palabras se han asignado dos variables de tipo String (vocales, consonantes).

- **vocales** = "aeiou"
- **consonantes** = "bcdfghjklmnpqrstvwxyz"

También se ha agregado una variable de tipo entero "numPalabras", que almacena un valor numérico que es el número de palabras que se van a generar.

Con un bucle for que se recorre el número de veces del valor de la variable numPalabras. A su vez, este bucle contiene un condicional para alternar entre vocales y consonantes para que sean legibles las palabras. También cuenta con una variable que genera de forma automática un valor aleatorio entre 3 y 8, que será la cantidad caracteres que formarán la palabra. Y por último, otro bucle for que formará la palabra.

- **usarVocal:** variable de tipo Booleana que almacena de forma aleatoria con el método de *nextBoolean()* que nos ofrece la librería de Random, si la variable es "true" o "false".
- **longitud:** variable de tipo entero que genera de forma aleatoria un número entero entre 3 y 8, mediante el método *nextInt()*.
- **palabra:** constructor para almacenar los caracteres que formarán la palabra.

El bucle for que forma las palabras con los caracteres que se irán obteniendo aleatoriamente, contiene una condicional para controlar si se usa la vocal o la consonante.

Para generar la palabra se irá almacenando los caracteres de forma aleatoria, para ello según la condición se usará la variable vocal o consonante para obtener la letra que se va a usar. Esto se consigue con el método *charAt()* y *nextInt()*.

Antes de finalizar este segundo bucle for cambiamos el valor de la "usarVocal" para que se puede alternar entre las vocales y las consonantes.

Y al finalizar el primer bucle for generamos una variable de tipo entero auxiliar que agregaremos al final de la palabra para poder comprobar que la palabra fue ordenada correctamente las palabras en la segunda clase OrdenarPalabras.

Ahora nos centraremos en la clase **"OrdenarPalabras"** esta clase recibe por consola la palabras que las irá almacenando en una lista, que mediante el método *sort()* ordenará de forma alfabética.

Para conseguir esto, utilizamos un *try-with-resource* gestionar automáticamente los recursos, y garantizar que se cierran correctamente después de usarlos

Creamos un constructor que permitirá recibir las palabras desde la consola. Cuando se reciba la palabra mediante un bucle while se comprobará que se sigue recibiendo palabras, en cuanto la variable palabra este vacía se finalizará. Mientras la variable palabra contenga información se irán agregando a la lista de palabras.

Una vez finalice el bucle, se ordena las palabras mediante *Collections.sort()* con este método podemos ordenar de forma alfabética las palabras, comenzando por la a y finalizando por la z.

Se ha utilizado una variable auxiliar para mostrar la posición nueva de la palabra. A diferencia de la clase **"GeneraPalabras"** en este caso el valor de variable auxiliar se podrá por delante de la palabra, y así podremos ver el puesto actual, con el que tenían al principio.

Por último, con un bucle for se imprime el resultado del orden de las palabras y se podrá compara su posición actual con la anterior.

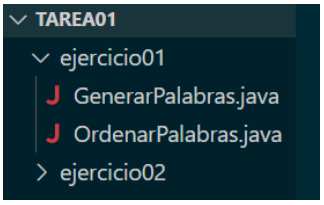
Convertir los archivos, GenerarPalabras.java y OrdenarPalabras.java

Para poder utilizar una tubería deberes convertir estos archivos a clases. Para realizar este proceso, yo lo he usado la terminal VSC. Utilizando el siguiente código se generarán las dos clases de forma automática.

Código:

javac GenerarPalabras.java OrdenarPalabras.java

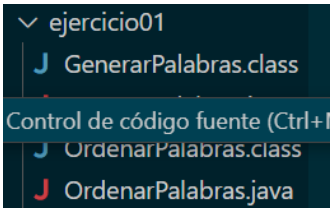
Como se puede ver en la siguiente imagen no existen todavía las clases, solo se encuentran los archivos .java.



Ahora se introduce el código mencionado anteriormente en la terminal y se generarán las clases que hemos desarrollado.

```
PS C:\00-PROGRAMACION\PSP\Tarea01\ejercicio01> javac GenerarPalabras.java OrdenarPalabras.java
```

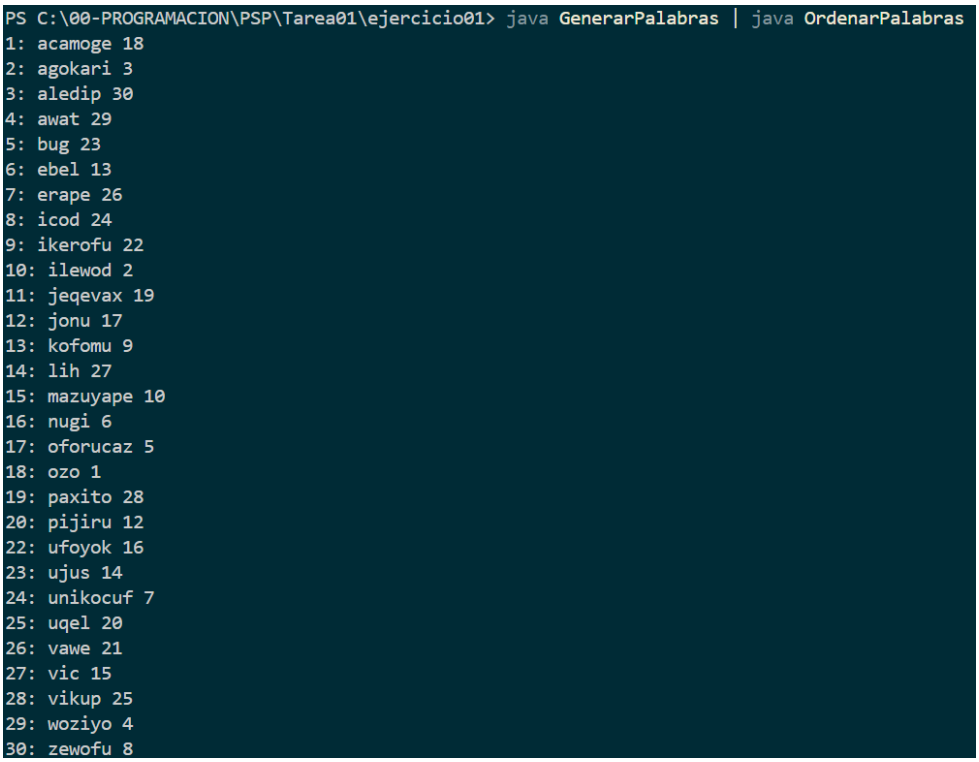
Y ya si podremos ver como se han generado las clases de ambos desarrollos.



Por último, añadimos el siguiente código para que se ejecuten ambas aplicaciones mediante una tubería y uno genere la tubería y el otro las ordena. El código usado sería: **java GenerarPalabras | java OrdenarPalabras**

```
PS C:\00-PROGRAMACION\PSP\Tarea01\ejercicio01> java GenerarPalabras | java OrdenarPalabras
```

Lo que nos generaría la siguiente respuesta, como se puede ver en la imagen.



Como se puede ver en la imagen, vemos las palabras ordenadas de 1 al 30 y en la posición en la que fue generada al final de la palabra.

Ejercicio 02

Segunda parte

En este ejercicio se leerá un archivo txt, el que contendrá 100 palabras, en este caso las palabras son reales.

Para la realización de este apartado he comenzado por la segunda clase ***"CoordinarProcesos"***, ya que será la que leerá el archivo.

Esta clase tiene varias funciones:

- **Descripción general**
- **Descripción de la primera función leer archivos.txt de entrada**
- **Descripción de la segunda función generar los sub-archivos.**
- **Descripción de la tercera función de generar procesos.**
- **Descripción de la cuarta función de ejecución de los procesos.**
- **Descripción de la quinta función de unificar los archivos de salida.**

Descripción general

Antes de ejecutar las diferentes funciones se ha agregado un condicional que se ejecuta en caso de no se hayan introducido parámetro al ejecutar el programa.

Se han agregado diferentes variables para almacenar los datos recibidos, así como los que se generen durante la ejecución del programa.

- **ficheroEntrada:** variable de tipo String, almacena el nombre del archivo.txt del que se obtendrán las palabras.
- **numInstancias:** variable de tipo entero, que almacena el número de procesos que debería realizar el programa.
- **ficheroSalida:** variable de tipo String, que almacena el nombre del archivo.txt que tendrá el archivo de salida del programa.
- **nomFicheroSalida:** Array de tipo String, que almacena el nombre del archivo de salida para generar los archivos que recibirán las particiones de las palabras.
- **listaPalabras:** Lista que almacena todas las palabras del archivo.txt de entrada.

Descripción de la primera función leer archivos.txt de entrada

Mediante un try-whit-resource para controlar que se gestione automáticamente el recurso de lectura del archivo y se cierre correctamente después de usarlo.

Mediante BufferedReader y FileReader, se lee el archivo que se ha pasado como argumento de entrada al ejecutar el programa.

Se ha agregado una variable de tipo String que almacena la palabra que se ha leído.

A continuación, se ejecuta un bucle while que se recorre mientras existan palabras en el archivo, cada vez que se recorre y exista una palabra se almacenará en la lista de palabras que declaramos al principio.

Descripción de la segunda función generar los sub-archivos.

La primera parte de esta función, esta compuesta por una variable de tipo entero que almacenará el número de palabras que contendrá cada proceso. Para ello se redondea hacia arriba, el resultado obtenido del dividir el número total de las palabras que hay por el número de instancias.

También se declara una lista de tipo String para almacenar el nombre de los sub-archivos creados.

A continuación, se generán los nombres de los sub-archivos, para realizar esto se utiliza un bucle for que se recorre tantas veces como se haya indicado como argumento de número instancias. Se almacena el nombre generado dentro de un nuevo String nuevoArchivo.

Una vez generado, se almacena dentro de la lista nomSubArchivos, para utilizarlos más adelante.

Se vuelve a usar un try-with-out, para cerrar automáticamente el recurso.

En el parámetro que se indica en el try-with-out se introduce un BufferedWriter con un FileWriter. Se crea el archivo en caso de que no existe, y se escribe la nueva palabra. Dentro de esta try-with-out, se introduce un bucle for que se recorre tantas palabras se tengan que introducir para esta instancia, se agrega la palabra y un espacio en blanco que esto permite que la siguiente palabra se escriba en la siguiente palabra.

Descripción de la tercera función de generar procesos.

En esta parte del código se generará los procesos que más adelante se ejecutarán.

Se han declarado dos listas, una para almacenar los nombres de los archivos de salida que se van a pasar a la otra clase como parámetro y otra lista para almacenar los procesos.

Utilizaremos un bucle for que se recorrerá tantas veces como instancias se hayan recibido como parámetro.

Dentro de este bucle se declara un variable de tipo String que almacena los nombres de los sub-archivos de salida que se generen. Una vez generado se almacena dentro de la lista.

Ahora se configura un ProcessBuilder que contendrá los argumentos de subArchivo y subArchivo de salida, para cuando se ejecute ProcesarPalabras.jar.

Por último al proceso se le configura inheritIO(), que estandariza el proceso de uso de la entrada/salida.

Se inicia el proceso con start() y lo almacenamos en la lista de procesos.

Descripción de la cuarta función de ejecución de los procesos.

Es una de las partes más simples del código, simplemente se crea un bucle for que recorre la lista de procesos, y las va ejecutando, una a una.

Descripción de la quinta función de unificar los archivos de salida.

En esta parte final se unificarán todos los archivos generados en los procesos indicados al iniciar la aplicación.

Para ello utilizaremos un try-whit-out, para controlar que el proceso se cierre de forma automática. Creamos el fichero de salida que se indicó como parámetro al iniciar la aplicación. Además, se ejecutará el siguiente código que unirá el resultado de todos los procesos.

Habrá un bucle for que se recorrerá tantas veces como sub-archivos de salida se hayan generado en el proceso.

En este bucle encontramos otro try-with-out, de ese modo controlamos de forma automática el cierre del proceso. Como parámetro introducimos un BufferedReader y FileReader, para ir leyendo los archivos generados de salida y poder unificarlo dentro del archivo de salida recibido como parámetro inicial.

Para ir escribiendo las palabras se ha utilizado un bucle while que ira incluyendo palabra por palabra en el archivo de salida. Cada vez que se escriba una palabra se agrega un salto de líneas para evitar que se escriba a continuación de la palabra anterior.

Por último, se muestran dos mensajes informados al usuario del todo el proceso.

Primera parte

Para finalizar el desarrollo he realizado el primer apartado, ya que por motivos de compresión me resultaba más cómodo de realizar.

Para esta clase "ProcesarPalabras" contamos con una función que convierte las palabras que contienen un número de caracteres pares en mayúscula y el si son minúsculas las escribe en minúsculas.

En la primera parte del código se crean dos variables de tipo String, una para almacenar el argumento del nombre del archivo que contiene las palabras y otra para almacenar el nombre del archivo de salida.

A continuación, he utilizado un try-whit-out, para que se cierre el proceso de forma automática, como parámetros se han pasado el primero sería para la lectura del archivo y el segundo para el archivo de escritura (entra y salida).

Se a añadido una variable para almacenar la línea leída.

Se utiliza un bucle while que se recorre mientras se lea una palabra, en el momento que ya no se reciba ninguna más se finaliza.

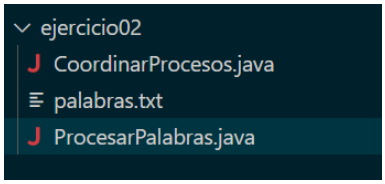
Dentro del bucle tenemos un condicional que si el resto de la división entre la longitud de la palabra y 2 es igual a cero, la palabra se escribirá en mayúscula, de lo contrario se escribirá minúscula.

Una vez escrita la palabra se agrega una nueva línea.

Parte tres de la tarea.

Una vez creadas las clases tenemos que convertirla a "-jar" para ello tenemos que hacer tres pasos.

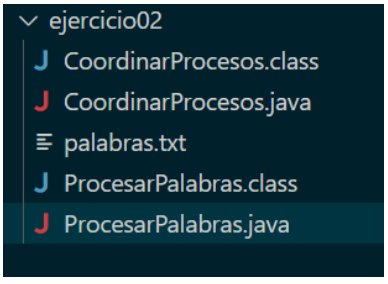
Primer paso



Como se puede ver en la imagen anterior no contamos con las clases, para eso tendremos que crearlas. Para ello utilizaremos el comando **"javac CoordinarProcesos.java ProcesarPalabras.java"**.

```
PS C:\00-PROGRAMACION\PSP\Tarea01\ejercicio02> javac CoordinarProcesos.java ProcesarPalabras.java
```

Ahora como se puede ver tenemos las dos clases creadas.



Segundo paso.

A continuación, tenemos que convertir estos archivos en -jar, pero para hacer esto, primero hay que generar dos archivos denominados MANIFEST, uno por cada clase.

Estos archivos contendrán la versión del archivo y la main-class, que hará referencia al nombre de cada clase. No hay que olvidar que después de estos dos datos hay que dar un salto de línea, si esto no se hace no funcionará el archivo jar cuando se genere.

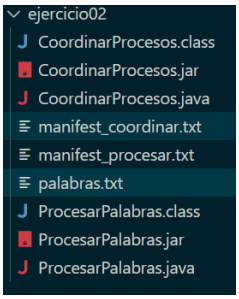
Tercer paso

Por último, convertimos las clases en un .jar, para ello utilizamos el siguiente código:

- **"jar cfm CoordinarProcesos.jar manifest_coordinar.txt CoordinarProcesos.class"**
- **"jar cfm ProcesarPalabras.jar manifest_procesar.txt ProcesarPalabras.class"**

```
PS C:\00-PROGRAMACION\PSP\Tarea01\ejercicio02> jar cfm CoordinarProcesos.jar manifest_coordinar.txt CoordinarProcesos.class
PS C:\00-PROGRAMACION\PSP\Tarea01\ejercicio02> jar cfm ProcesarPalabras.jar manifest_procesar.txt ProcesarPalabras.class
```

Al ejecutar estos códigos habremos convertido las clases.

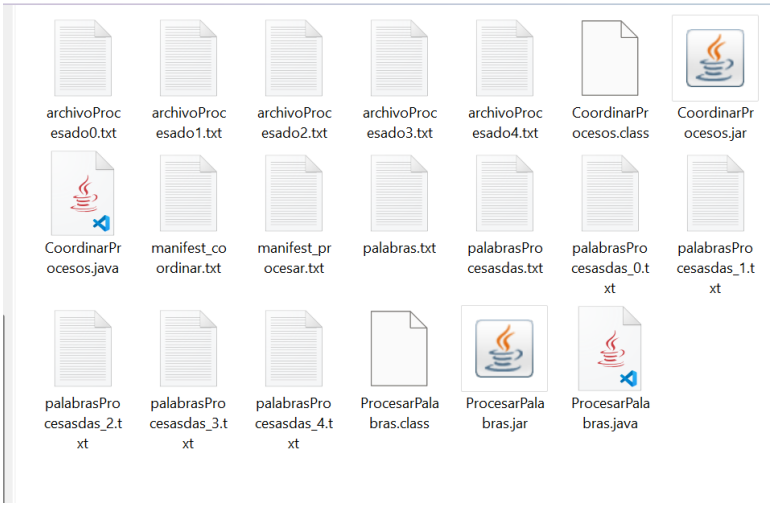


Ya podemos ejecutar el programa y verificar su funcionamiento. Para ello utilizando el siguiente código se ejecutará la aplicación y los procesos que le indiquemos como parámetro. Para el ejemplo se utilizará el archivo palabras.txt que contiene 100 palabras.

- **"java -jar CoordinarProcesos.jar palabras.txt 5 palabrasProcesadas.txt**

```
PS C:\00-PROGRAMACION\PSP\Tarea01\ejercicio02> java -jar CoordinarProcesos.jar palabras.txt 5 palabrasProcesasdas.txt
Se han procesado el fichero palabras.txt durante 5 instancias.
Si lo desea puede comprobar el resultado en el fichero de salida indicado palabrasProcesasdas.txt.
```

En la siguiente imagen se puede ver como se han generado todas los sub-archivos que se han indicado. Así como el archivo de salida que indicamos.



Ejemplo del archivo de salida que se paso como parámetro.