LECTURA Y ESCRITURA DE INFORMACIÓN

TEORÍA

FICHEROS DE DATOS

Hasta este momento, los datos que hemos utilizado en nuestros programas solo existían mientras estos estaban en ejecución. Una vez cerrábamos el programa, toda la información que se había generado desaparecía.

Sin embargo, en muchas situaciones necesitamos que esos datos no se pierdan al finalizar el programa, sino que se conserven de alguna forma para poder volver a utilizarlos más adelante, ya sea en la misma aplicación o en otras diferentes.

Esto nos lleva a la necesidad de que los datos tengan persistencia, es decir, que se almacenen de manera permanente, incluso después de que el programa termine de ejecutarse.

Java no impone una estructura en un fichero. Es tarea nuestra estructurar los ficheros para que cumplan con los requisitos de nuestros programas e indicar cómo queremos almacenar y recuperar los datos.

Para trabajar con ficheros se sigue este proceso:

1. Importar los paquetes necesarios. Se tienen que importar las clases apropiadas de los paquetes java.io y java.nio
2. Crear un objeto File. Se crea un objeto File que representa la ubicación del fichero en el sistema de archivos.
3. Abrir el fichero. Antes de leer o escribir un fichero, hay que abrirlo. Se utiliza una clase adecuada.
4. Leer o escribir datos. Una vez abierto el fichero, se utilizan los métodos apropiados de las clases de entrada y salida para leer o escribir datos.
5. Cerrar el fichero. Al acabar, es importante cerrar el fichero para evitar problemas.

Uso de ficheros

En muchos programas es necesario guardar datos más allá del tiempo de ejecución, como guardar información de usuarios o configuraciones, registrar logs o historiales o leer datos externos como catálogos, productos, notas, etc.

Existen dos tipos de ficheros según el criterio del contenido:

- Ficheros de texto o de caracteres. La información se guarda como caracteres, codificados en Unicode, ASCII u otras codificaciones. Se puede acceder a ellos desde cualquier editor de texto plano. Características:
  - Secuencia caracteres
  - Interpretable por un ser humano
  - Generalmente portable
  - Escritura/Lectura menos eficiente que los ficheros binarios
  - Requiere más tamaño que un fichero binario para representar la misma información
- Ficheros binarios o de bytes. La información se guarda en bytes, codificada en binario. Lo que guardan realmente no es el texto en sí, sino su representación en binario. Características:
  - Secuencia de bytes (interpretables como tipos primitivos)
  - No interpretable por un ser humano

  - Generalmente no portable (se tiene que leer en el mismo tipo de ordenador y con e mismo lenguaje de programación con el que fue escrito)
  - Escritura/Lectura eficiente
  - Almacenamiento eficiente de la información

Existen dos tipos de ficheros según el criterio del modo de acceso:

- Secuencial. La información contenida en un archivo se organiza como una secuencia de bytes (o caracteres), de forma que para acceder a un byte en una posición determinada —por ejemplo, el byte número i— es necesario haber leído antes todos los bytes anteriores, desde el primero hasta el (i - 1).
- Acceso directo. Existe también la posibilidad de acceder directamente a la información ubicada en una posición concreta del archivo, es decir, al byte número i, sin necesidad de recorrer los anteriores. Este tipo de acceso se denomina acceso directo o aleatorio, y un ejemplo muy familiar de esta forma de acceso lo encontramos en las estructuras de datos como los arrays o vectores, donde podemos acceder directamente a cualquier elemento por su índice.

Algunos tipos comunes de ficheros son:

- .txt → texto plano
- .csv → texto estructurado por comas
- .dat, .ser → binarios o serializados
- .json, .xml → datos estructurados

java.io y java.nio

Son dos bibliotecas de JAVA que aportan funcionalidades para la entrada y salida de datos (lectura y escritura, manejo de flujos y canales). Tienen objetivos parecidos, pero sus características y enfoques son diferentes:

- Java.io es el paquete de entrada/salida original de JAVA. Se basa en flujos y bloqueo de operaciones I/O
- Java.nio se introdujo como una alternativa más escalable y flexible, basada en canales y buffers

A continuación, se presentan algunas diferencias clave entre java.io y java.nio

- Flujos vs Canales:
  - java.io utiliza flujos (streams) para representar la entrada y salida de datos. Los flujos son secuencias unidireccionales de bytes y pueden ser de entrada (InputStream) o de salida (OutputStream).
  - java.nio utiliza canales (channels) para manejar la entrada y salida de datos. Los canales son bidireccionales y pueden ser utilizados tanto para leer como para escribir datos. Los buffers son objetos que almacenan temporalmente los datos antes de que sean transferidos. En nio, las operaciones de lectura/escritura se realizan utilizando buffers, lo que permite un manejo más eficiente de la memoria.
- Bloqueo vs. No bloqueo:
  - Las operaciones en java.io son bloqueantes, lo que significa que un hilo se bloquea (espera) hasta que la operación de lectura o escritura se complete. Esto

puede resultar en un rendimiento subóptimo en aplicaciones que requieren un alto grado de concurrencia
    - java.nio admite operaciones no bloqueantes, lo que permite a un hilo continuar con otras tareas mientras se completan las operaciones de lectura y escritura. Esto mejora la escalabilidad y el rendimiento en aplicaciones que requieren una alta concurrencia
- Selectores:
    - Los selectores son una característica única de java.nio que permite a un solo hilo monitorear múltiples canales para eventos de I/O. Los selectores hacen posible el manejo eficiente de múltiples conexiones simultáneas utilizando pocos hilos.
    - java.io no proporciona una funcionalidad equivalente a los selectores, lo que puede resultar en un mayor consumo de recursos y menor escalabilidad en aplicaciones que requieren un alto grado de concurrencia.
- Mapeo de archivos en memoria:
    - java.nio proporciona la capacidad de mapear archivos en memoria, lo que permite el acceso directo a los datos del archivo a través de la memoria. El mapeo de archivos en memoria puede mejorar significativamente el rendimiento en ciertos casos de uso, como el procesamiento de archivos grandes.
    - java.io no admite el mapeo de archivos en memoria, lo que significa que todas las operaciones de lectura y escritura deben realizarse a través de flujos de entrada y salida, lo que puede resultar en una menor eficiencia en comparación con el acceso directo a la memoria
- Charset y codificación de caracteres:
    - java.nio proporciona un mejor soporte para la codificación y decodificación de caracteres, incluyendo la capacidad de trabajar con diferentes conjuntos de caracteres (Charsets). La clase Charset en el paquete java.nio.charset permite convertir datos de texto entre diferentes codificaciones de caracteres y manejar la decodificación de bytes a caracteres y la codificación de caracteres a bytes..
    - java.io admite la codificación y decodificación de caracteres, pero su soporte es más limitado que con java.nio. Las clases InputStreamReader y OutputStreamWriter en java.io admiten dichas conversiones, pero el proceso es menos flexible y más propenso a errores que el enfoque basado en Charset de java.nio.

La elección entre utilizar las clases de java.io o java.nio depende de las necesidades y requisitos específicos de la aplicación que se está desarrollando. En general, para aplicaciones simples o que no requieren características avanzadas de I/O, java.io es una opción adecuada debido a su simplicidad y facilidad de uso. Sin embargo, si la aplicación necesita un mayor rendimiento, escalabilidad, manejo de concurrencia o acceso a caracteristicas avanzadas de I/O, java.nio es la mejor opción.

FICHEROS DE TEXTOS

Clase File

Representación abstracta de los nombres y de las rutas de archivos y directorios. Presenta una vista abstracta, independiente del sistema, de los nombres de ruta jerárquicos. No lee ni escribe por sí misma.

La clase File en Java es una parte esencial del paquete java.io, que proporciona funcionalidad para representar y manipular archivos y directorios en el sistema de archivos. La clase File no se utiliza para leer o escribir datos en sí, sino para acceder a la información del archivo o directorio, como la ruta, tamaño, atributos, permisos, etc. otros. El resto de clases que manipulan ficheros parten de la existencia de una clase File, por lo que es la base de cualquier operación de manipulación de ficheros.

Para crear un objeto File que representa un archivo o directorio en el sistema de archivos, se utiliza el constructor File. Pueden utilizarse diferentes constructores según cómo se desee especificar la ruta del archivo:

Para poder manipular la información de los sistemas de ficheros, primero hay que crear una instancia de la clase File de la siguiente manera:

```
File archivo = new File (String rutaFichero);
```

Con `rutaFichero` siendo la ruta donde se almacena el fichero dentro del dispositivo en el que se ejecuta la aplicación. El nombre de ruta puede ser absoluto o relativo.

Un nombre de ruta absoluta es aquel que está completo (no requiere disponer de más información para ubicar el archivo dentro del sistema de ficheros de nuestro sistema operativo). La ruta se indica a partir del elemento raíz y, por el contrario, el nombre de una ruta relativa es aquel que no incluye el elemento raíz, y se tendrá que interpretar a partir de la ruta del directorio de trabajo.

Ejemplo:

	ABSOLUTA	RELATIVA
Windows	C:\eclipse-workspace\ficheros\src\Principal.java	ficheros\src\Principal.java
Linux	/home/profe/ejercicios/Ejercicio1.txt	/ejercicios/Ejercicio1.txt

Teniendo ya el objeto de la clase File (que nos permite trabajar con nuestros ficheros almacenados), se pueden realizar varias operaciones. Las más utilizadas nos permiten conocer las propiedades de un archivo:

Método	**Descripción**
`File (String pathname)`	Crea un objeto de tipo File a partir de su ruta
`boolean createNewFile()`	Crea un nuevo archivo vacío con la ruta definida por el File
`boolean delete()`	Borra el fichero/directorio
`boolean exists()`	Comprueba si existe el archivo o el directorio indicado.
`String getName()`	Devuelve el nombre del archivo o del directorio indicado.
`String getPath()`	Devuelve el nombre de la ruta relativa del archivo.
`String getAbsolutePath()`	Devuelve el nombre de la ruta absoluta del archivo.
`String getParent()`	Devuelve el nombre de ruta del directorio padre, o nulo si no hay directorio padre.

File [] listFiles()	Obtiene listado de ficheros en el directorio
boolean isDirectory()	Indica si es un directorio
boolean isFile	Indica si es un fichero
mkdir() y mkdirs()	Crea un directorio (s) representado por e
long length()	Devuelve el tamaño del archivo en bytes.
boolean canExecute()	Comprueba si la aplicación puede ejecutar el archivo indicado.
boolean canRead()	Comprueba si la aplicación puede leer el archivo indicado.
boolean canWrite()	Comprueba si la aplicación puede modificar el archivo.

EJEMPLO FICHERO

```
File fichero = new File ("ejemplo1.txt");

if(fichero.exists())
```