Tema XI: FICHEROS

"Los ordenadores son buenos siguiendo instrucciones,
no leyendo tu mente"

INTRODUCCIÓN

Los programas necesitan comunicarse con su entorno, tanto para recoger datos como para devolver resultados. La manera de representar estas entradas y salidas en Java es a base de streams (flujos de datos). Son una abstracción empleada en muchos lenguajes de programación, entre ellos JAVA, para representar cualquier fuente que produzca o consuma información. Su nombre viene de que pueden considerarse como una cadena de datos con una longitud posiblemente desconocida.

Un stream es una conexión entre el programa y la fuente o destino de los datos. Esto de lugar a una forma general de representar muchos tipos de comunicaciones. Por ejemplo cuando se quiere imprimir algo en pantalla, se hace a través de un stream que conecta el monitor al programa. Se da a ese stream la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/escritura de archivos, la comunicación a través de Internet.

Existen dos tipos de flujos: binarios (byte) y texto. En los primeros, como su nombre indica, la información "que fluye" está en formato binario, en los segundos en modo texto.
Cada una de estas dos categorías se divide a su vez en: flujos de entrada y flujos de salida. Los primeros nos proporcionan datos, son entradas a nuestro programa, en los segundos nuestro programa escribe datos, es decir, salidas de nuestro programa.

Clase Path

La clase **Path** proporciona muchas utilidades relacionadas con ficheros y con la obtención de información básica sobre esos ficheros. Es una forma de referenciar una ruta (de un fichero o directorio). No se utiliza para abrir ficheros ni operar con la información que contienen.

Importamos java.nio.file.

Tenemos que instanciar un objeto Path que represente la ruta donde se ubica el archivo que queremos procesar, ese archivo puede existir o no existir.

- Podemos crear un Path a partir de una cadena (usamos \\ en Windows o / en Windows y Linux)

```
Path p = Paths.get("D:/tmp/foo");
```

- Podemos crear un Path a partir de variables de entorno y concatenando ubicaciones.

```
Path p = Paths.get(System.getProperty("user.home"),"logs", "foo.log");
// Coge la ruta del usuario y le concatena la carpeta logs y a ésta el archivo foo.log.
```

- Podemos crear un Path a partir de una cadena pasada como parámetro.

```
Path p = Paths.get(args[0]);
```

Con la clase Path podemos realizar las siguientes operaciones:

- Recuperar partes de una ruta.
- Eliminar redundancias de una ruta.
- Convertir una ruta.
- Unir dos rutas.
- Crear una ruta relativa a otra dada.
- Comparar dos rutas.

Ejemplo: EjemplosPathsBasico.java

Clase Files

La clase Files nos va a permitir leer, escribir, manipular archivos y directorios a través de métodos estáticos.
La clase Files trabaja sobre objetos Path.
Las principales operaciones que vamos a poder realizar sobre archivos y directorios son:

- Verificación de existencia y accesibilidad

```
boolean Files.exist(Path p)
```

boolean Files.isRegularFile(Path p)
boolean Files.isDirectory(Path p)

- Crear un fichero y un directorio

  Files.createFile(Path p)
  Files.createDirectory(Path p)

- Borrar un archivo o directorio

  Files.delete(Path p)
  Files.deleteIfExists(Path p)

- Copiar un archivo o directorio

  Files.copy(Path origen, Path destino)

- Mover un archivo o directorio

  Files.move(Path origen, Path destino)

- Recorrer un directorio

```
if (Files.isDirectory(Paths.get(".."))) {
        try {
            DirectoryStream<Path> stream = Files.newDirectoryStream(Paths.get(".."));
            for (Path path : stream) {
                    System.out.println(path.getFileName());
                    }
          } catch (IOException e) {
                System.err.println(e);
         }
```

Ejemplo: EjemplosFiles.java

Ejemplos

1. Programa que:
   a) Pide ruta y nombre de un fichero y nos da información sobre él
   b) Permite borrar un fichero
   c) Lista los archivos contenidos en un directorio y muestra información sobre ellos o lo crea si no existe

Clase Files para leer y escribir

Además de para realizar operaciones con archivos, vamos a utilizar la clase Files para las operaciones de I/O.

Las operaciones I/O (Entrada/Salida) son las que permiten que un programa acceda a datos externos, tanto para leerlos como para escribirlos.

En Java para estas operaciones se trabaja con dos paquetes:

- java.io: Más antiguo.
  - Se basa sólo en Streams. Un Stream es una secuencia de bytes entre un origen y un destino. No se almacenan en ninguna parte y no te puedes mover hacia delante ni hacia atrás.
  - Las operaciones son bloqueantes, es decir mientras se está realizando el programa tiene que esperar.
- java.nio: Java7.
  - Se basa en Buffers. Los bytes se almacenan en unas estructuras de datos llamadas buffers. Nos podemos mover hacia delante y hacia atrás.
  - Las operaciones no son bloqueantes, es decir mientras se está realizando una operación el programa puede continuar haciendo otras cosas.

Vamos a utilizar El paquete **java.nio.file** para trabajar con **archivos** independientemente de si realizamos I/O con streams (java.io) ó con buffers (java.nio)

La mayoría de los métodos que crean objetos para trabajar con archivos reciben un parámetro llamado OpenOptions. Es opcional y si no se especifica el método funcionará con su comportamiento por defecto. Los valores principales que podemos darle a este parámetro son:

- WRITE – Abre el archivo para escritura.
- APPEND – Añade datos al final de un archivo. Se usa junto con WRITE y CREATE.
- TRUNCATE_EXISTING – Trunca el archivo a 0 bytes. Se usa junto con WRITE.
- CREATE_NEW – Crea un archivo Nuevo y lanza una excepción si ya existe.
- CREATE – Abre el archive si existe o crea uno Nuevo si no existe.

- DELETE_ON_CLOSE – Borra el archivo cuando se cierra el stream. Es útil para archivos temporales.

Para que reconozca estos parámetros debemos importar en el programa:

import static java.nio.file.StandardOpenOption.*;

Las opciones de apertura se pueden introducir en la llamada al método separadas por comas o en un array de OpenOptions, una en cada celda.

Si no se pone nada por defecto se suele abrir el fichero para escribir creándolo si no existe y borrando su contenido si existe, esto es con las opciones: CREATE, TRUNCATE_EXISTING y WRITE.

Archivos pequeños: leer y escribir de una sola vez

Podemos leer o escribir el contenido entero de un archivo de una sola vez. Los métodos aseguran que el archivo se cierra al procesar todo el contenido o si se lanza una excepción.

- Leer un archivo binario de una sóla vez: Nos devuelve un array de bytes con toda la información del fichero:

  byte[] Files.readAllBytes(Path p)

- Escribir un archivo binario de una sóla vez:

  Files.write(Path p, byte[] b, Charset c, OpenOptions o)

  Ejemplo:LeerFicheroPequeñoBytes.java

- Leer un archivo de texto de una sóla vez: Nos devuelve un ArrayList con las líneas leídas:

  List <String> Files.readAllines(Path p)

- Escribir un archivo de texto de una sóla vez:

  Files.write(Path p, List <String> s, Charset c, OpenOptions o)

  Ejemplo: LeerFicheroPequeñoCaracteres.java

Archivos binarios o texto sin buffer

La lectura y escritura es secuencial. Se utilizan las clases: InputStream (para el fichero del cual se lee) y OutputStream (para el fichero en el cual se escribe).
Para leer se utiliza el método read de InputStream que puede leer de 1 a n bytes del fichero y devuelve el número de bytes leídos y -1 si alcanza el final del fichero.
Para escribir se utiliza el método write de OutputStream, que escribe de 1 a n bytes en el fichero.
Tenemos que cerrar los archivos después de procesarlos usando el método close.

Ejemplo: ArchivosSecuencialSinBuffer

Archivos de texto con buffer

La lectura y escritura también es secuencial. Se utilizan las clases: BufferedReader (para el fichero del cual se lee) y BufferedWriter (para el fichero en el que se escribe).
Para leer se utiliza el método readLine de BufferedReader que lee línea a línea del archivo y devuelve null cuando llega al final del fichero.

Ejemplo: ArchivoSecuencialLeer.java

Para escribir se utiliza el método write de BufferedWriter , al cual se le pasa un String y el número de caracteres a escribir en el archivo.

Ejemplo:ArchivoSecuencialEscribir.java

Si queremos añadir información al final del archivo creamos el BufferedWriter con las opciones: WRITE y APPEND.

Tenemos que cerrar los archivos después de procesarlos usando el método close.

Ejemplos:

2. Codificar un programa que lee cadenas de teclado y las escribe en un archivo, cada cadena en una línea.
3. Codificar un programa que lee un fichero de texto y escribe en otro sólo las líneas que empiezan por mayúscula.

Acceso aleatorio a archivos de texto

Se puede acceder de forma aleatoria a cualquier posición de un archivo de texto. Para ello utilizamos la clase RandomAccessFile de java.io junto con el método seek. Al escribir en una posición del fichero estamos sobreescribiendo su contenido.

Ejemplo:ArchivoAleatorioJava.java