Principales diferencias: `java.io` frente a `java.nio`

- **Orientación:** `java.io` está orientado a flujos de datos (*Stream-oriented*), procesando byte a byte o carácter a carácter. `java.nio` permite trabajar orientándose a canales y buffers (*Buffer-oriented*), moviendo bloques de datos más grandes y eficientes.
- **Bloqueo:** Las operaciones de `java.io` son estrictamente bloqueantes (*Blocking I/O*). `java.nio` permite operaciones no bloqueantes (*Non-blocking I/O*), ideales para servidores de alta concurrencia.
- **Representación de Rutas:** `java.io.File` mezcla la definición de la ruta con los métodos de manipulación física. `java.nio` separa esto limpiamente: **`Path`** representa la ruta lógica y **`Files`** ejecuta las acciones físicas.
- **Gestión de Errores y Metadatos:** `java.io` suele devolver valores booleanos (`false`) si una operación falla, sin dar detalles. `java.nio` lanza excepciones específicas (como `NoSuchFileException`) y ofrece acceso nativo a los atributos del sistema de archivos (permisos POSIX, enlaces simbólicos, etc.).

Clases y Métodos involucrados en operaciones NIO.2

Para realizar cualquier operación, primero necesitas representar la ruta del archivo o directorio mediante **`Path.of()`** (o `Paths.get()` en versiones anteriores de Java):

java

```
Path ruta = Path.of("ruta/al/archivo.txt");
```

Usa el código con precaución.

A continuación se detallan los métodos estáticos de la clase **`java.nio.file.Files`** para cada acción solicitada:

• Crear un fichero/directorio

- **Fichero:** `Files.createFile(Path)` crea un archivo vacío. Lanza excepción si ya existe.
- **Directorio simple:** `Files.createDirectory(Path)` crea un único directorio.
- **Directorios jerárquicos:** `Files.createDirectories(Path)` crea un directorio y todos los padres necesarios que no existan.

• Borrar un fichero/directorio

- **Borrado estricto:** `Files.delete(Path)` borra el archivo o directorio (este último debe estar vacío). Lanza excepción si no existe.
- **Borrado seguro:** `Files.deleteIfExists(Path)` devuelve un booleano indicando si el archivo existía y fue borrado, evitando excepciones si no se encuentra.

• Renombrar / Mover un fichero/directorio

En NIO.2, renombrar y mover se realizan con el mismo método, ya que renombrar es simplemente mover el archivo a la misma ruta con un nombre distinto.

- **Método:** `Files.move(Path source, Path target, CopyOption... options)`.
- **Opciones comunes:** `StandardCopyOption.REPLACE_EXISTING` (reemplaza si ya existe el destino) o `StandardCopyOption.ATOMIC_MOVE` (operación atómica).

• **Abrir un fichero/directorio**

NIO.2 no requiere "abrir" formalmente un fichero de forma persistente, sino que abres flujos de datos (*Streams* o *Channels*) asociados a él que se gestionan automáticamente.

- **Abrir para lectura continua:** `Files.newInputStream(Path)` o `Files.newBufferedReader(Path)`.
- **Listar directorio:** `Files.list(Path)` o `Files.newDirectoryStream(Path)` para inspeccionar los archivos contenidos.

• **Indicar un juego de caracteres (Charset)**

Para especificar codificaciones como **ISO-8859-1** o **UTF-8**, se utilizan las clases `java.nio.charset.Charset` y `java.nio.charset.StandardCharsets`.

- **Uso:** Se pasa como parámetro opcional en los métodos de lectura/escritura.
- **Ejemplo:** `StandardCharsets.ISO_8859_1` o `Charset.forName("ISO-8859-1")`.

• **Leer todas las líneas de un fichero (como lista de Strings)**

- **Método:** `Files.readAllLines(Path)` (utiliza UTF-8 por defecto).
- **Con Charset específico:** `Files.readAllLines(Path, Charset)`.
- *Nota para archivos muy grandes:* Es mejor usar `Files.lines(Path)`, que devuelve un `Stream<String>` y no carga todo el archivo en memoria a la vez.

• **Escribir una línea, o líneas en un fichero**

- **Método:** `Files.write(Path, Iterable<? extends CharSequence>, OpenOption...)`.
- Permite pasarle una lista de `String` (`List<String>`) directamente. Crea el archivo si no existe o lo sobrescribe por defecto.

• **Añadir información a un fichero ya existente (Append)**

- **Método:** Se utiliza el mismo método `Files.write`, pero añadiendo la opción de apertura adecuada.
- **Parámetro clave:** `StandardOpenOption.APPEND`.
- **Ejemplo:** `Files.write(ruta, lineas, StandardOpenOption.APPEND);`.

• **Cerrar fichero**

- Si utilizas los métodos directos de `Files` (como `readAllLines` o `write`), **NIO maneja la apertura y el cierre automáticamente**, por lo que no tienes que preocuparte por cerrar recursos.
- Si abres flujos avanzados (como `BufferedReader` o `Stream`), debes usar la estructura **try-with-resources** para que se cierren solos al terminar el bloque, ya que todas estas clases implementan `AutoCloseable`:

  java

  ```
  try (BufferedReader reader = Files.newBufferedReader(ruta,
  StandardCharsets.ISO_8859_1)) {
      // Leer el archivo aquí
  } // Se cierra automáticamente aquí
  ```