

PROMETEO

Unidad 7: Entrada/Salida y Ficheros

Programación

Técnico Superior de DAM / DAW

Sesión 16 – Entrada/Salida estándar. Lectura con Scanner

Cuando desarrollas aplicaciones reales, tarde o temprano necesitas que tu programa interactúe con el exterior: recibir datos que introduce un usuario, procesar archivos, mostrar resultados o generar reportes. A esto se le llama entrada/salida (I/O) y es uno de los pilares de cualquier software profesional, desde herramientas de consola hasta sistemas distribuidos que intercambian información entre servidores.

System.in

Entrada estándar.
Normalmente, el teclado.

System.out

Salida estándar. Lo que ves en la consola.

System.err

Salida de errores.
Pensada para mensajes críticos o diagnósticos.

Un stream es una secuencia de datos que fluye entre tu programa y el sistema operativo. Cuando lees del teclado, estás consumiendo bytes que viajan desde System.in. Cuando escribes en pantalla, envías datos hacia System.out. Esta abstracción hace posible que el mismo código funcione en entornos distintos (consola, archivos, red...) sin cambiar la lógica principal.

Para trabajar con entrada de datos de forma más amigable, Java ofrece la clase Scanner, probablemente la herramienta más práctica para principiantes y también ampliamente utilizada en aplicaciones de producción por su versatilidad.

Un Scanner puede leer desde diferentes orígenes:

- Teclado: `new Scanner(System.in)`
- Archivos: `new Scanner(new File("datos.txt"))`
- Cadenas de texto: `new Scanner("123 456 Hola")`

Métodos específicos para tipos

- `nextInt()` → lee números enteros
- `nextDouble()` → lee reales
- `nextLine()` → lee líneas completas con espacios
- `hasNext()` y variantes → comprueban si hay más datos disponibles

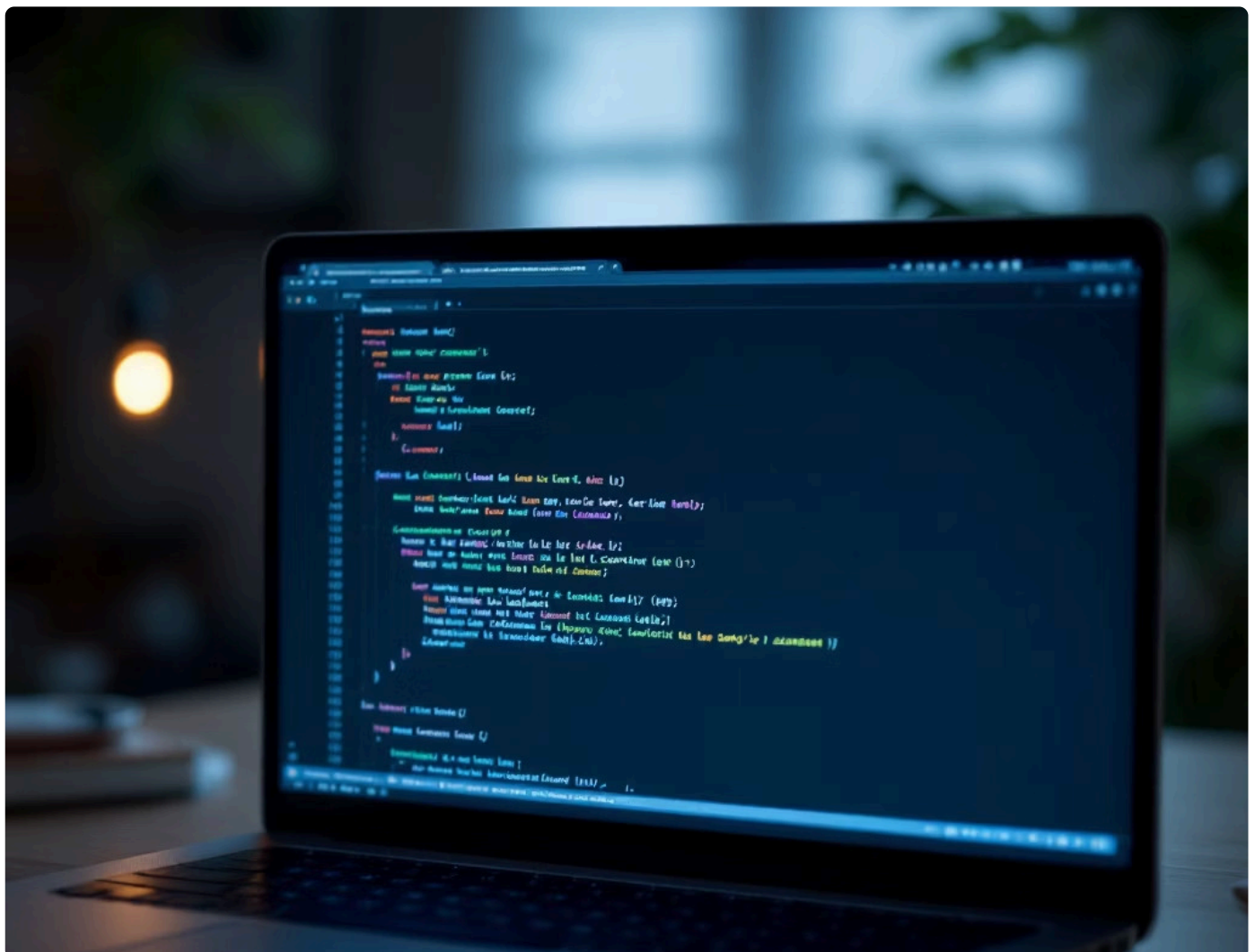
❏ Diferencia importante

Mientras `next()` lee hasta el siguiente espacio, `nextLine()` consume toda la línea, incluidos sus espacios internos.

Una de las diferencias más importantes, y causa frecuente de errores entre estudiantes, es el comportamiento distinto entre `next()` y `nextLine()`. Además, después de leer un número con `nextInt()`, queda pendiente el salto de línea, por lo que necesitas llamar a `nextLine()` para limpiar el buffer antes de leer otro string. Si no lo haces, tu programa capturará la línea vacía sin que el usuario pueda escribir nada.

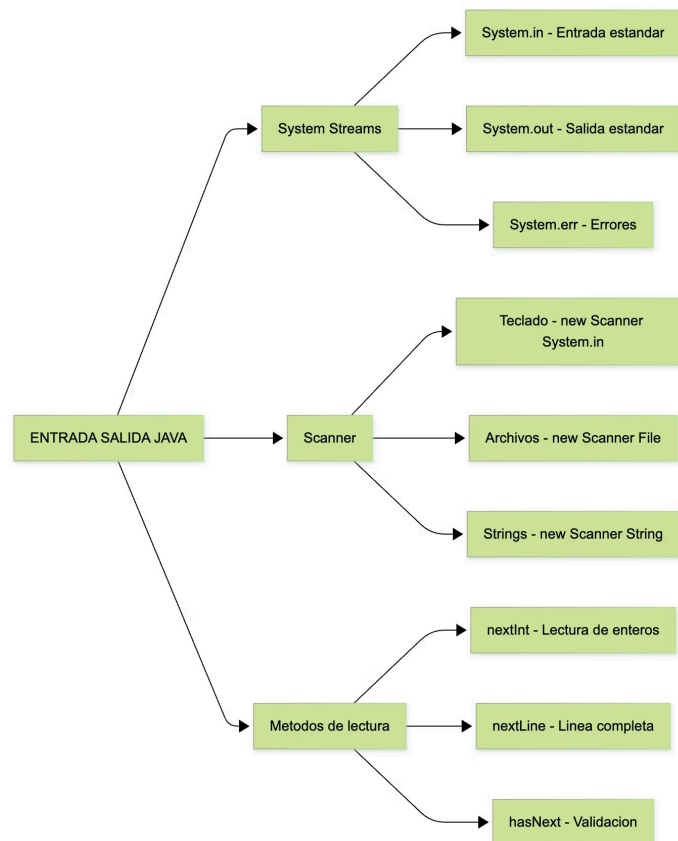
Para entrada/salida con archivos, Scanner facilita la lectura estructurada sin necesidad de manejar manualmente buffers o procesos complejos. Puedes procesar archivos línea por línea o palabra a palabra, lo que permite manejar datos grandes de forma eficiente sin cargar todo en memoria.

Comprender estos mecanismos te permitirá crear programas interactivos, importar datos desde fuentes externas, validar entrada de usuario y trabajar con archivos de forma robusta. Es un paso esencial para avanzar hacia aplicaciones más grandes, integraciones con APIs y manipulación de datos a escala.



Esquema Visual

A continuación tienes un esquema conceptual en formato Mermaid que representa cómo funciona la entrada/salida estándar y el papel del Scanner como herramienta principal para lectura de datos.



Descripción detallada del esquema

01

ENTRADA/SALIDA JAVA es el nodo principal que agrupa todos los mecanismos de comunicación entre el programa y el exterior.

02

La rama **System Streams** representa los tres flujos nativos de Java: System.in: el origen de entrada, normalmente el teclado. System.out: el destino para mostrar mensajes. System.err: usado para errores y diagnósticos.

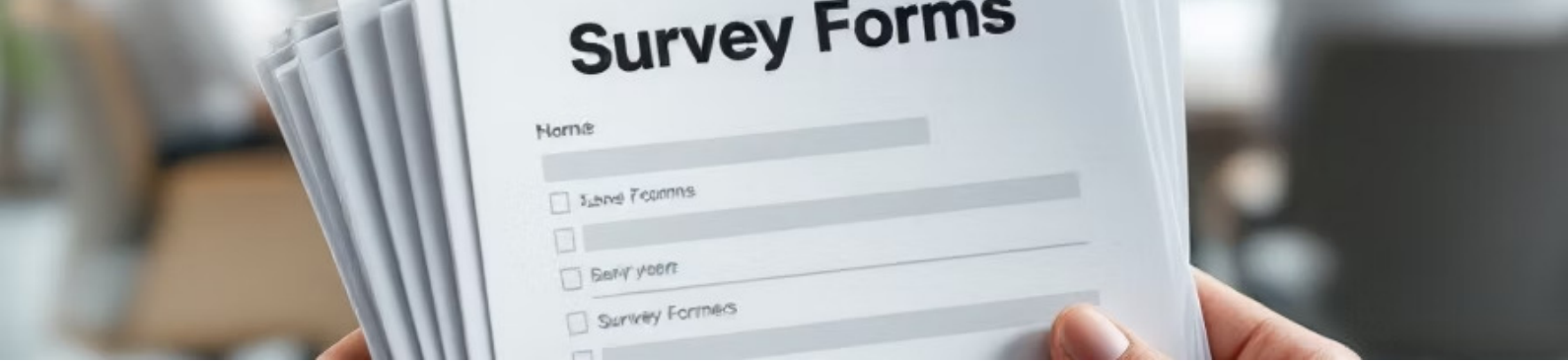
03

La rama **Scanner** muestra sus tres posibles fuentes principales: lectura desde teclado, lectura desde archivo, lectura desde cadenas de texto internas.

04

Finalmente, la rama **Métodos de Lectura** detalla los métodos clave utilizados para capturar distintos tipos de datos y validar entrada.

Este mapa conceptual resume cómo encajan los diferentes elementos de I/O en Java y qué función cumple cada uno.



Caso de Estudio – Sistema de Encuestas de SurveyMonkey

SurveyMonkey procesa millones de encuestas diarias provenientes de usuarios de todo el mundo. Aunque su infraestructura es compleja y distribuida, uno de los componentes esenciales para el procesamiento de datos locales y migraciones de grandes volúmenes es sorprendentemente sencillo: lectura mediante Scanner.

Cada encuesta puede tener respuestas largas, con espacios, comas, saltos de línea e incluso caracteres especiales. Los archivos CSV generados pueden alcanzar tamaños de hasta 10GB, por lo que no es viable cargarlos en memoria.

Necesitan procesarlos línea por línea, de forma secuencial y eficiente. Para lograrlo, el sistema utiliza:

```
Scanner sc = new Scanner(new FileInputStream(archivo));
```



Procesamiento eficiente

Procesar archivos de cualquier tamaño sin conocer previamente el número de líneas.



Lectura completa

Leer línea completa con `nextLine()`, necesaria porque las respuestas pueden contener espacios.



Validación

Validar la existencia de más contenido mediante `hasNextLine()`.



Separación

Separar campos con `String.split(",")`, manteniendo respuestas complejas sin perder formato.

SurveyMonkey prioriza la eficiencia evitando estructuras enormes en memoria. Scanner, al trabajar sobre un flujo, solo mantiene en RAM la línea actual, permitiendo que sistemas de análisis, generación de estadísticas y procesamiento paralelo funcionen en tiempo real.

Este enfoque permite transformar archivos gigantes en reportes estadísticos dinámicos, detectando patrones, midiendo tendencias y analizando respuestas abiertas sin que el sistema se ralentice o consuma más memoria de la necesaria.

La clave del éxito es que Scanner, usado correctamente y en combinación con flujos (`FileInputStream`), ofrece una lectura flexible, segura y eficiente que se adapta tanto a datos pequeños como a volúmenes masivos.



Herramientas y Consejos



Usa try-with-resources para cerrar automáticamente Scanner

Esto evita fugas de memoria y es una buena práctica profesional:

```
try (Scanner sc = new Scanner(new File("datos.txt"))) {  
    // Lectura segura  
}
```

El Scanner se cierra automáticamente al finalizar el bloque.



Verifica la entrada antes de leer

Evita errores del tipo InputMismatchException:

```
if (sc.hasNextInt()) {  
    int edad = sc.nextInt();  
}
```

Validar primero te permite ofrecer mensajes claros al usuario y mantener el programa estable.



Configura delimitadores personalizados

Si procesas formatos específicos como CSV, puedes cambiar el comportamiento por defecto:

```
sc.useDelimiter(",");
```

Esto te permite leer campos directamente sin necesidad de partir cadenas manualmente.

Mitos y Realidades

✗ Mito: "Scanner es lento y no sirve para archivos grandes."

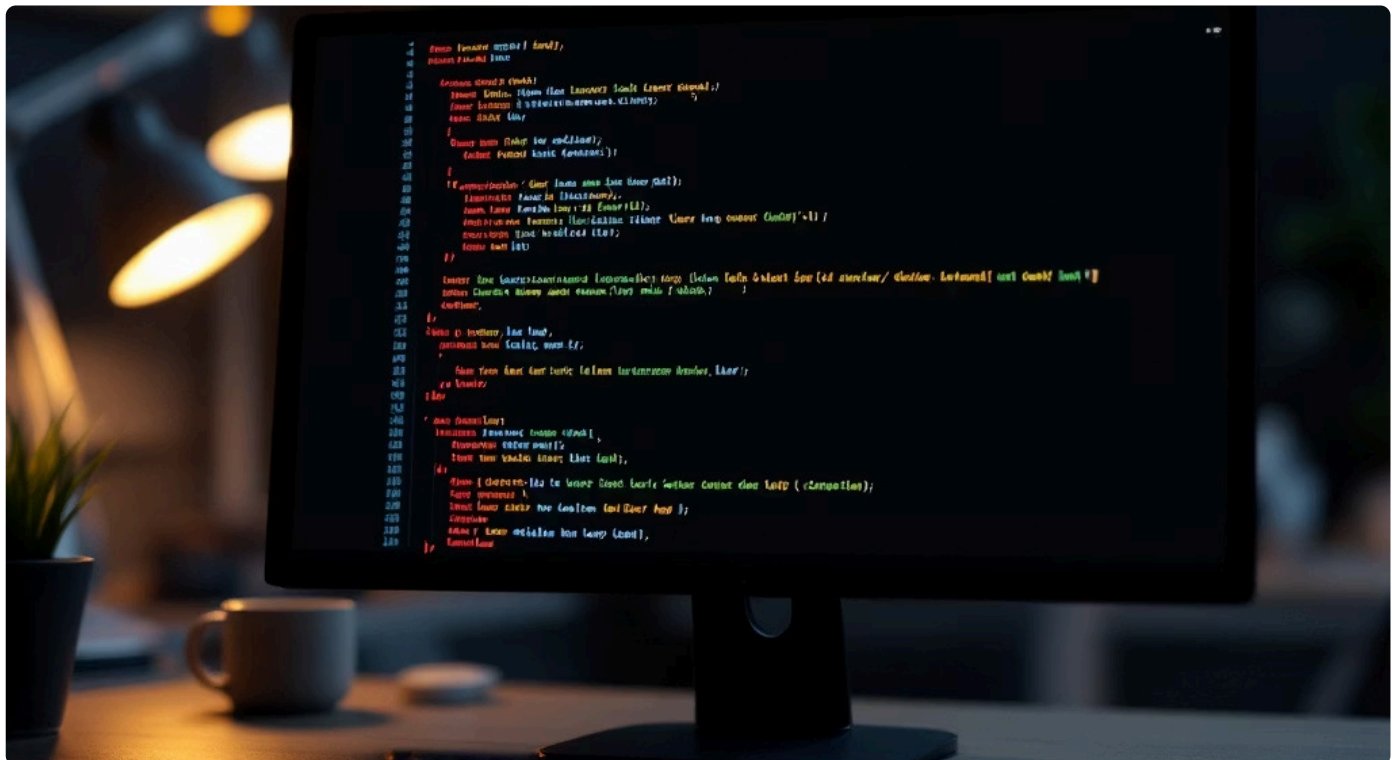
→ **FALSO.** Con el buffer adecuado y un flujo como `FileInputStream`, `Scanner` es suficientemente eficiente para la mayoría de aplicaciones profesionales. Solo en casos extremos se recomienda `BufferedReader`.

✗ Mito: "`next()` y `nextLine()` son intercambiables."

→ **FALSO.** `next()` lee hasta espacio. `nextLine()` lee toda la línea. Combinarlos sin cuidado provoca errores, especialmente después de `nextInt()`.

📄 Resumen Final

- La entrada/salida permite que el programa se comuniquen con el exterior.
- `System.in/out/err` son los streams estándar para teclado, consola y errores.
- `Scanner` es versátil: teclado, archivos, strings.
- `next()` lee hasta espacio; `nextLine()` captura la línea completa.
- SurveyMonkey procesa archivos de hasta 10GB usando `Scanner` línea a línea.
- `try-with-resources` garantiza cierre automático y seguro de `Scanner`.

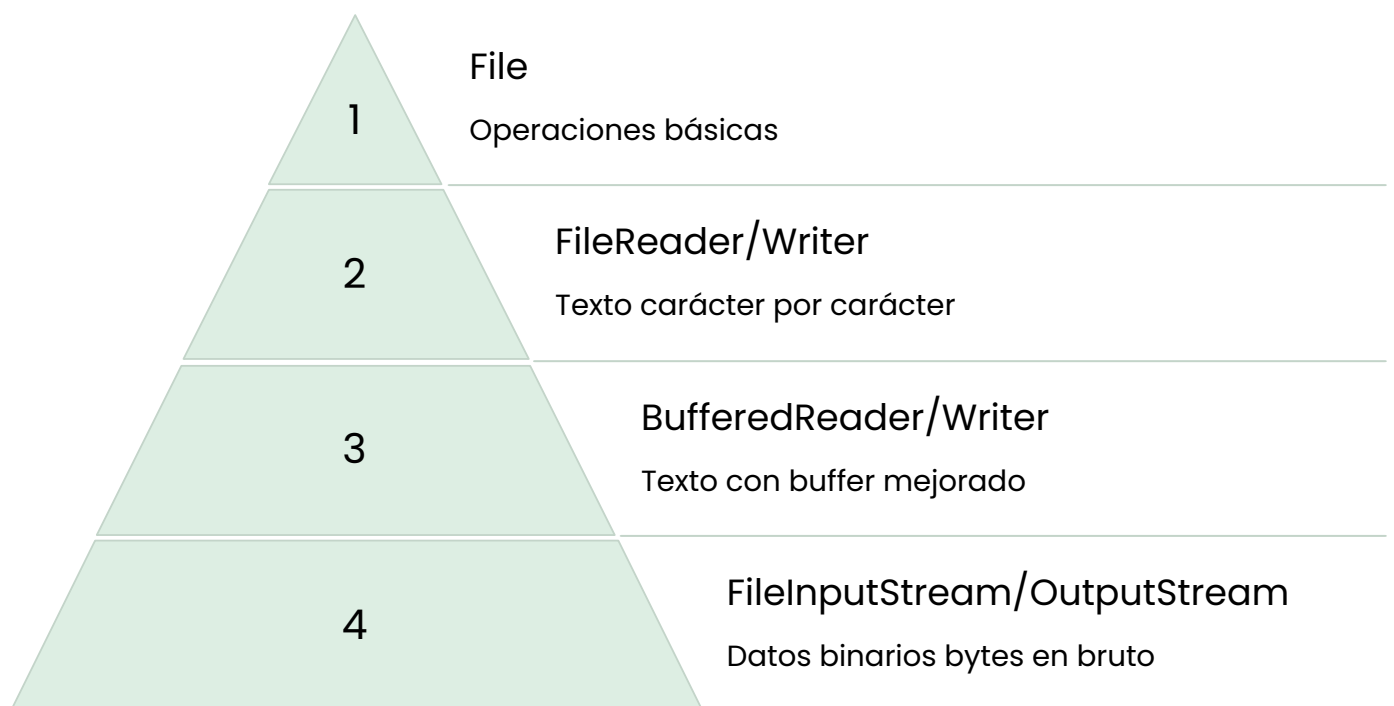




Sesión 17 – Manejo de ficheros y registros

Cuando trabajas en proyectos reales, tarde o temprano necesitas que tu programa recuerde información más allá de la ejecución. Un sistema sin persistencia solo funciona mientras está abierto; una vez finaliza, todos los datos desaparecen. Por eso la gestión de archivos es fundamental en prácticamente cualquier aplicación profesional: inventarios, ventas, reportes, configuraciones, logs, usuarios y cualquier elemento que necesite conservarse en el tiempo.

Java ofrece un ecosistema robusto para manejar archivos, estructurado en distintos niveles según el tipo de operación y el formato de datos que necesitas tratar.



En la base está la clase `File`, que representa rutas y archivos del sistema operativo y te permite realizar operaciones esenciales como crear nuevos archivos, comprobar si existen, listar directorios, renombrar o eliminar. Aunque `File` no lee ni escribe contenidos, sí actúa como punto de partida para trabajar con I/O.

Para datos de texto

Cuando quieres procesar datos de texto, las clases **`FileReader`** y **`FileWriter`** te permiten leer y escribir carácter por carácter. Funcionan, pero no son eficientes para archivos grandes porque cada operación produce un acceso directo al disco.

Para datos binarios

Para datos binarios —imágenes, PDF, archivos comprimidos, audio— se utilizan **FileInputStream** y **FileOutputStream**, diseñados para manejar bytes en bruto.

Para mejorar este rendimiento, Java incorpora `BufferedReader` y `BufferedWriter`, que trabajan acumulando datos en memoria temporal (buffers) y enviando o recibiendo información en bloques más grandes. Esto reduce el número de accesos al disco y acelera significativamente la lectura y escritura, especialmente en archivos grandes.

Un concepto clave en la persistencia es el de **registros**. Un registro es una unidad lógica de información compuesta por varios campos relacionados. Por ejemplo, un producto de inventario puede tener: ID, nombre, precio y cantidad. Un empleado puede tener: nombre, ID, salario y departamento.

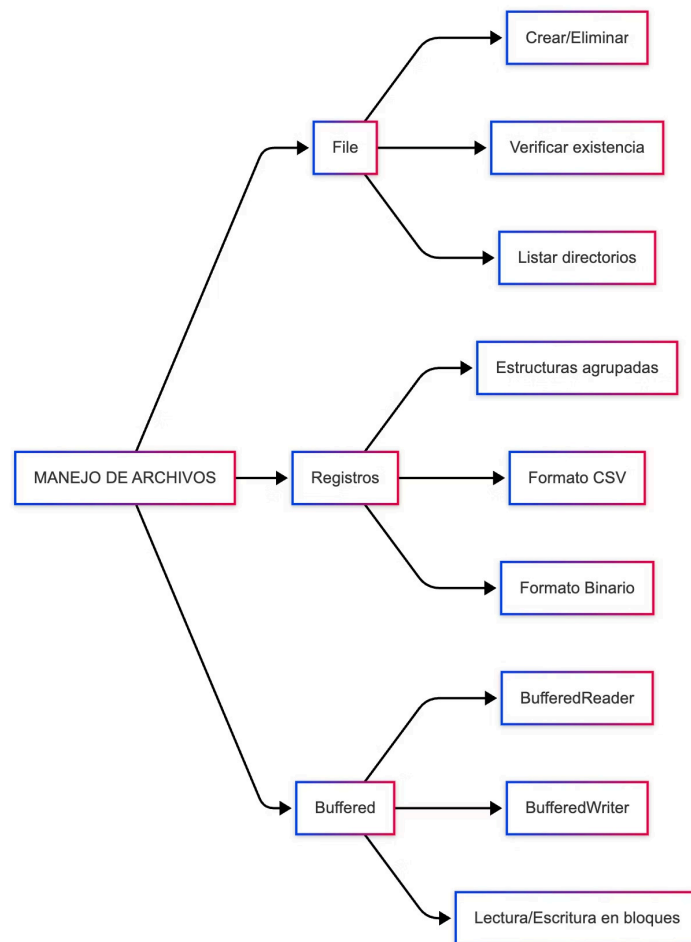
Estos registros pueden guardarse en formatos como CSV (texto estructurado por comas) o en formatos binarios optimizados para lectura rápida. Los CSV tienen la ventaja de ser fácilmente legibles por humanos y compatibles con hojas de cálculo o herramientas de análisis, mientras que los binarios suelen ser más eficientes en tamaño y velocidad.

Organizar registros correctamente es esencial para que tus programas puedan recuperar, actualizar o eliminar información sin inconsistencias. Si lo combinas con una estructura ordenada de archivos —por ejemplo, un archivo por día, por mes o por categoría— puedes construir sistemas robustos que escalen con el tiempo sin necesidad de bases de datos complejas.

Finalmente, entender la diferencia entre acceder a un archivo carácter por carácter y hacerlo mediante buffer es clave en el rendimiento. Los buffers permiten procesar datos en bloques de manera eficiente y son indispensables en aplicaciones que manejan grandes volúmenes de información. En la práctica profesional, rara vez utilizarás `FileReader` o `FileWriter` directamente; casi siempre trabajarás con `BufferedReader` y `BufferedWriter` por la mejora de rendimiento que proporcionan.

La gestión de archivos no es solo un aspecto técnico: es una herramienta que te permite crear sistemas reales, capaces de almacenar información persistente de forma segura, estructurada y eficiente.

Esquema Visual



Descripción completa del esquema

MANEJO DE ARCHIVOS
Es el concepto principal que agrupa las operaciones de persistencia.

La rama File
Representa operaciones básicas sobre archivos y directorios: creación, eliminación, verificación de existencia, manipulación de rutas.

La rama Registros
Refleja cómo se organiza información compleja en archivos: estructuras con varios campos relacionados, almacenamiento en CSV para texto estructurado, almacenamiento binario para optimización.

La rama Buffered
Muestra las herramientas clave para mejorar rendimiento: `BufferedReader` y `BufferedWriter`, lectura y escritura por bloques, reducción de accesos al disco.

Este esquema te permite visualizar cómo interactúan las distintas capas del manejo de archivos en Java.



Caso de Estudio – Sistema de Ventas de Shopify

Shopify gestiona uno de los ecosistemas de comercio electrónico más grandes del mundo, con 1,7 millones de comercios activos que generan millones de transacciones diarias. Para poder analizar ventas, generar reportes y mantener información accesible para los comerciantes, necesita un sistema resistente y eficiente de manejo de archivos.

Estructura de cada transacción

- ID de transacción
- Fecha
- Cliente
- Lista de productos
- Total de la compra
- Método de pago

Para reportes

Para reportes y exportaciones, Shopify utiliza archivos **CSV**, ya que son fáciles de procesar por herramientas de análisis como Excel, Tableau o Google Sheets y pueden manipularse mediante librerías estándar.

Para rendimiento

Para procesos internos de alto rendimiento, emplean archivos **binarios**, porque permiten leer y escribir registros más rápido y ocupan menos espacio en disco.

Una parte esencial de su arquitectura es la escritura en lotes. En lugar de escribir transacciones una por una (lo que generaría decenas de miles de accesos a disco por minuto), Shopify usa un `BufferedWriter` para acumular hasta 1000 transacciones antes de escribirlas como un bloque. Este método reduce drásticamente la carga del sistema operativo y acelera la generación de reportes diarios y mensuales.

Para el análisis de datos históricos, el sistema utiliza `BufferedReader`, que permite leer archivos de ventas muy grandes, algunos de hasta 50GB por tienda grande.

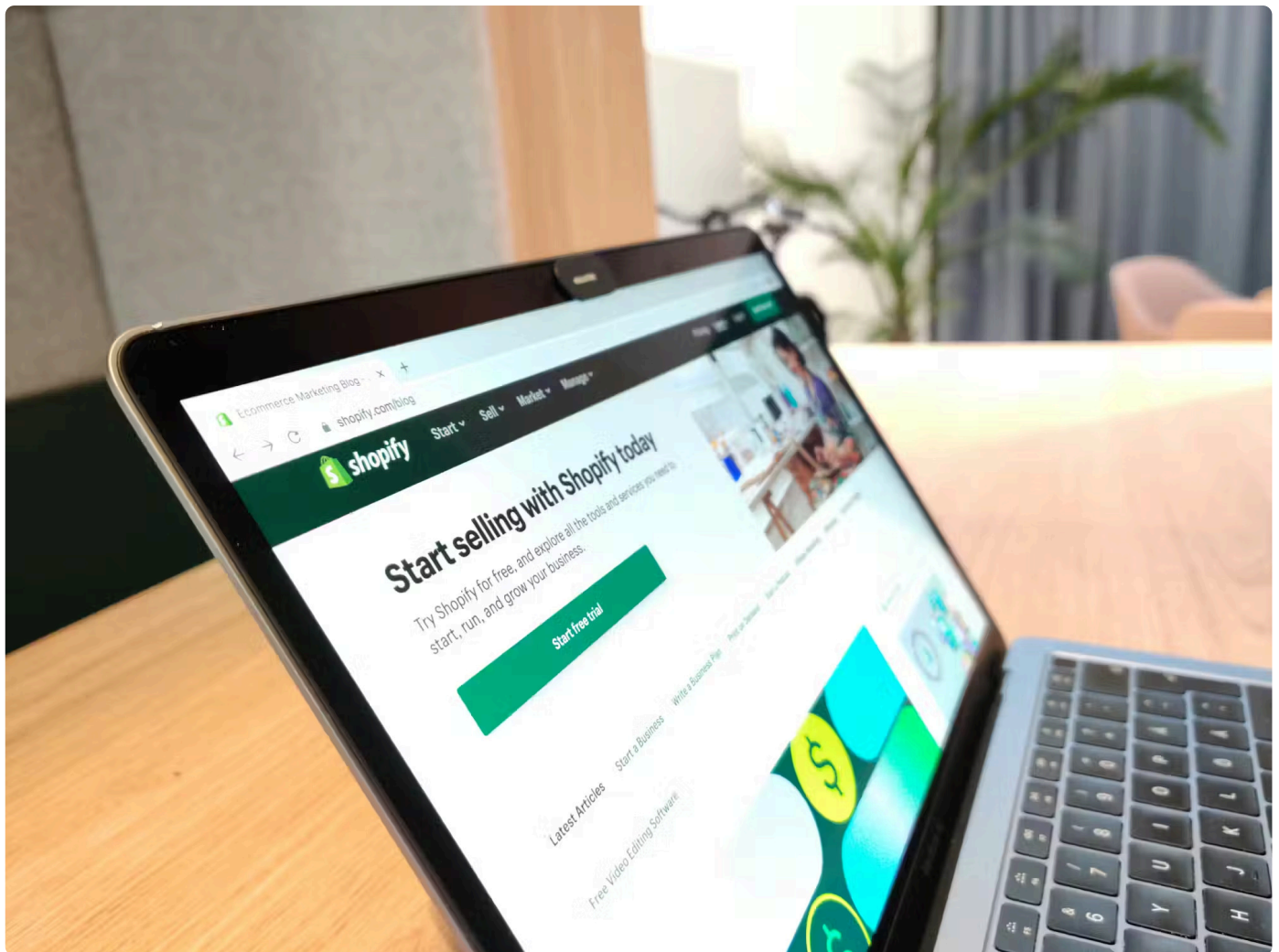
Al trabajar en bloques de 8KB, evita cargar el archivo completo en memoria y procesa cada línea de forma secuencial, identificando tendencias, detectando estacionalidad y generando métricas de rendimiento comercial.

La organización de los archivos también sigue un patrón rígido y lógico:

- ventas_2024_01_15.csv
- ventas_2024_01_16.csv
- ventas_2024_02_01.csv

Esto permite buscar información por períodos concretos sin tener que recorrer grandes volúmenes de datos. Para un equipo de desarrollo, este diseño facilita tareas como auditorías, copias de seguridad, integraciones con terceros o migraciones hacia sistemas de análisis avanzados.

Gracias a esta arquitectura basada en registros, buffers y formatos adecuados, Shopify puede ofrecer estadísticas en tiempo real a millones de tiendas sin comprometer rendimiento ni estabilidad.



Herramientas y Consejos



Usa try-with-resources para cerrar archivos automáticamente

Es fundamental para evitar fugas de memoria y corrupción de archivos:

```
try (BufferedWriter writer = Files.newBufferedWriter(path)) {  
    writer.write("dato");  
}
```

Java garantiza el cierre correcto incluso si ocurre una excepción.



Verifica la existencia del archivo antes de intentar leerlo

Evita errores y mejora la experiencia del usuario:

```
if (Files.exists(path)) {  
    // procesar archivo  
}
```

Con esto evitas un `FileNotFoundException` y puedes ofrecer alternativas cuando el archivo no exista.



Crea copias de seguridad antes de modificar archivos importantes

Una práctica muy común en aplicaciones empresariales:

```
Files.copy(original, backup);
```

Esto te protege ante errores inesperados, bloqueos del sistema o interrupciones.

Mitos y Realidades

✗ Mito: "Los archivos de texto siempre son más lentos que los binarios."

→ **FALSO.** La diferencia solo es relevante en archivos muy grandes o procesos intensivos. Los archivos de texto permiten depurar, visualizar y trabajar con herramientas estándar, lo que los hace muy útiles en entornos profesionales.

✗ Mito: "BufferedReader y BufferedWriter solo sirven para archivos enormes."

→ **FALSO.** El buffering reduce las llamadas al sistema operativo en cualquier archivo, por pequeño que sea; por eso siempre mejora el rendimiento, incluso en operaciones moderadas.

Resumen Final

- El manejo de archivos permite persistir datos más allá de la ejecución.
- File gestiona operaciones básicas del sistema de archivos.
- FileWriter/Reader trabajan con texto; los streams manejan binarios.
- Los registros agrupan datos relacionados (CSV o binario).
- BufferedReader/Writer mejoran el rendimiento al trabajar por bloques.
- Shopify maneja archivos de hasta 50GB gracias a buffers y almacenamiento estructurado.