

**CUADERNO DE LABORATORIO ACCESO A DATOS 2-DAM**

**PORADA**



**Cristian Espinar Martinez**

# ÍNDICE

<b>PORTADA.....</b>	<b>0</b>
<b>ÍNDICE.....</b>	<b>1</b>
<b>Introducción.....</b>	<b>3</b>
<b>TEMA 1 : Introducción a la persistencia en ficheros.....</b>	<b>4</b>
1. ¿Qué es un fichero?.....	4
Tipos de acceso a la información:.....	5
Acceso secuencial:.....	5
Acceso aleatorio:.....	6
Tabla comparativa:.....	6
Sistema de ficheros y rutas.....	7
¿Qué es una ruta?.....	8
Tipos de rutas en Java:.....	8
Absolutas:.....	8
Relativas:.....	8
Cómo obtener el directorio actual del proyecto.....	8
Portabilidad con File.separator.....	9
<b>Tema 2: Gestión de Ficheros y Directorios con la Clase File.....</b>	<b>10</b>
Descripción.....	10
Introducción.....	10
Gestión de archivos con java.io.File.....	11
1. Crear una ruta con File.....	11
2. Métodos útiles para archivos.....	11
3. Gestión de directorios.....	12
Tabla resumen de métodos comunes:.....	14
<b>Tema 3: Lectura y Escritura Secuencial de Ficheros.....</b>	<b>14</b>
Descripción.....	14
Introducción.....	14
Escritura secuencial de ficheros.....	15
Lectura secuencial de ficheros.....	15
Consideraciones importantes.....	16
Tabla comparativa.....	17
<b>Tema 4: Acceso Aleatorio a Ficheros.....</b>	<b>17</b>
Descripción.....	17
Introducción.....	17
Creación y escritura en un archivo aleatorio.....	17
Ejemplo: Escribir varias líneas.....	18
Lectura desde una posición específica.....	18
Ejemplo: Leer desde una posición concreta.....	18
Consideraciones importantes.....	19
Tabla Resumen:.....	20
<b>Unidad Extra : Clase Scanner.....</b>	<b>20</b>

Descripción.....	20
Fundamentos de la clase Scanner.....	20
Métodos principales de Scanner.....	21
Uso de Scanner para leer desde teclado.....	21
Uso de Scanner para leer desde archivos.....	22
Creación de menús interactivos con Scanner.....	23
Comparación con otras clases de entrada.....	24
<b>Tema 4.1: Alternativa moderna con NIO - Gestión de Archivos con Path y Files.....</b>	<b>25</b>
Descripción.....	25
Introducción.....	25
¿Por qué usar Java NIO?.....	25
Conceptos clave en Java NIO.....	26
¿Cuándo utilizar Java NIO?.....	26
Ejemplo básico : uso de Path y Files.....	26
Ejemplo: lectura y escritura con Files y UTF-8.....	27
Ejemplo con canal y bufer.....	28
Ventajas de NIO sobre File.....	28
<b>Tema 5: Ficheros Binarios y Serialización de Objetos.....</b>	<b>29</b>
Descripción.....	29
Introducción.....	29
¿Qué es la serialización?.....	29
Escritura de objetos (serialización).....	29
Lectura de objetos (deserialización).....	30
Tabla resumen.....	31
<b>Tema 6: Lectura y Escritura de Ficheros XML.....</b>	<b>31</b>
Descripción.....	31
Introducción.....	31
Tecnologías para procesar XML en Java.....	32
Lectura de XML con DOM.....	32
Lectura con SAX (manejador de eventos).....	33
Tabla comparativa.....	35
Tema 6.1: Procesamiento XML con DOM (Document Object Model).....	35
Descripción.....	35
¿Qué es DOM?.....	37
Lectura de un documento XML con DOM.....	37
Escritura de un documento XML con DOM.....	38
Esquema del flujo de trabajo DOM.....	40
Archivos utilizados.....	40
<b>ACTIVIDADES.....</b>	<b>41</b>
UD1 - Actividad Práctica: Tema 1.1.....	41
UD1 - Actividad práctica Tema 1.2 - Uso de rutas en java.....	43
UD1 - Actividad Práctica Tema 2.1 - Gestión de archivos y directorios con File.....	46
UD1 - Actividad Práctica Tema 3.1 - Lectura y Escritura.....	50
UD1 - Actividad Práctica Tema 4.1 - Acceso Aleatorio a Ficheros.....	53
UD1 - 5.1 Lectura de fichero con Scanner.....	56

# Introducción

**La persistencia de la información** es una característica clave en cualquier sistema informático que necesita conservar datos más allá de la ejecución de un programa. A lo largo de esta unidad aprenderemos a utilizar diferentes mecanismos de almacenamiento de datos, centrados en el uso de ficheros como medio de persistencia básica en entornos Java. Trabajar con **ficheros** permite desarrollar aplicaciones que no dependen exclusivamente de bases de datos, pero que aún así conservan la información de forma organizada y estructurada. Este conocimiento sienta las bases para abordar más adelante el uso de bases de datos relacionales y no relacionales.

Comprender el funcionamiento de las rutas, los tipos de acceso, los buffers, streams, y la serialización de objetos será esencial para implementar soluciones persistentes eficaces.

## Objetivos de la Unidad

- Comprender los fundamentos de la persistencia en sistemas de ficheros.
- Diferenciar entre tipos de acceso: secuencial vs aleatorio.
- Utilizar correctamente las clases del paquete java.io.
- Crear, leer, escribir, modificar y eliminar ficheros en Java.
- Trabajar con rutas relativas y valores del sistema para garantizar la portabilidad.
- Implementar programas que manipulen archivos de texto y binarios.
- Aplicar técnicas de serialización de objetos y manipulación de ficheros XML con DOM, SAX y JAXB.

# TEMA 1 : Introducción a la persistencia en ficheros

## 1. ¿Qué es un fichero?

Un fichero (o archivo) es una secuencia de bytes almacenada de forma persistente en un dispositivo de almacenamiento. Se identifica mediante una ruta o «path», y puede contener cualquier tipo de información: texto, imágenes, objetos, etc.

Los ficheros son gestionados por el sistema de archivos del sistema operativo, el cual se encarga de:

Asignar una ruta y un nombre único dentro de un directorio.

Proteger el acceso concurrente mediante permisos.

Garantizar la integridad de los datos.

## Tipos de acceso a la información:

### Acceso secuencial:

se lee el fichero de principio a fin, útil para textos planos.

Este es el modo más básico y común de acceder a ficheros de texto. Se lee el archivo desde el principio hasta el final, línea a línea o carácter a carácter, en orden.

Ejemplo:

LecturaSecuencial.java:

```
import java.io.*;  
  
public class LecturaSecuencial {  
    public static void main(String[] args) throws IOException {  
        BufferedReader br = new BufferedReader(new  
FileReader("datos.txt"));  
        String linea;  
  
        while ((linea = br.readLine()) != null) {  
            System.out.println(linea);  
        }  
  
        br.close();  
    }  
}
```

datos.txt:

```
Línea 1  
Línea 2  
Línea 3  
Línea 4  
Línea 5
```

Este programa lee el archivo línea por línea hasta que ya no queden más (cuando `readLine()` devuelve `null`).

Resultado:

```
C:\Users\crist\.jdks\open  
Línea 1  
Línea 2  
Línea 3  
Línea 4  
Línea 5
```

### Acceso aleatorio:

(Random Access): permite saltar a una posición específica del archivo, ideal para estructuras binarias.

Este modo permite acceder directamente a una posición específica del archivo, sin tener que recorrerlo desde el principio.

Para ello se usa la clase RandomAccessFile, que funciona como una combinación de lector y escritor. Se puede mover el «puntero» a cualquier parte del archivo con .seek(posicion).

```
import java.io.*;

public class AccesoAleatorio {
    public static void main(String[] args) throws IOException {
        RandomAccessFile raf = new RandomAccessFile("datos.txt",
"rw");

        raf.seek(20); // Mover el puntero a la posición 20 (byte 20)
        String linea = raf.readLine(); // Leer desde ahí
        System.out.println("Contenido desde byte 20: " + linea);

        raf.close();
    }
}
```

```
C:\Users\crist\.jdks\openjdk-25\bin\ja
Contenido desde byte 20: LÃnea 3
```

Nota: La posición 20 significa el byte número 20, no la línea 20. Para acceder por líneas habría que recorrerlas.

### Tabla comparativa:

Característica	Acceso Secuencial	Acceso Aleatorio (RandomAccessFile)
Tipo de lectura	Desde el inicio hasta el final	Desde una posición específica
Lectura parcial	✗ No directa	✓ Sí

<b>Escritura</b>	<input checked="" type="checkbox"/> Sí, al final normalmente	<input checked="" type="checkbox"/> Sí, en cualquier posición
<b>Clase principal</b>	<code>BufferedReader, Scanner</code>	<code>RandomAccessFile</code>
<b>Ideal para</b>	<b>Archivos de texto</b>	<b>Archivos binarios o con registros fijos</b>

## Tabla comparativa: Acceso secuencial vs aleatorio

Característica	Acceso Secuencial	Acceso Aleatorio
Tipo de uso	Lectura lineal, texto	Datos binarios estructurados
Clases usadas	<code>FileReader, BufferedReader</code>	<code>RandomAccessFile</code>
Velocidad de lectura	Lenta para datos al final	Alta (si conoces la posición)
Modificación	Reescribir todo el archivo	Modificar por posiciones
Uso de punteros	No	Sí ( <code>seek()</code> y <code>getFilePointer()</code> )
Consumo de memoria	Bajo	Moderado

### // Acceso secuencial

```
BufferedReader br = new BufferedReader(new FileReader("datos.txt"));
String linea;
while ((linea = br.readLine()) != null) {
    System.out.println(linea);
}
br.close();
```

### // Acceso aleatorio

```
RandomAccessFile raf = new RandomAccessFile("datos.txt", "rw");
raf.seek(20); // ir al byte 20
String linea = raf.readLine();
raf.close();
```

# Sistema de ficheros y rutas

## ¿Qué es una ruta?

Una ruta es el camino que el sistema operativo necesita para acceder a un archivo o carpeta.

En Java, al trabajar con ficheros, es necesario indicar la ruta completa del archivo al que queremos acceder, ya sea para leerlo, escribirlo, modificarlo o eliminarlo.

### Tipos de rutas en Java:

#### Absolutas:

especifican todo el camino desde la raíz del sistema.

Es la ruta completa desde el origen del sistema de archivos (la raíz).

En Windows:

C:\\\\Users\\\\cristian\\\\Documentos\\\\datos.txt

En Linux/Mac:

/home/cristian/documentos/datos.txt

Ventaja: Es muy precisa.

Desventaja: No es portable entre sistemas ni usuarios.

#### Relativas:

se refieren a la ubicación del archivo respecto al directorio actual del programa.

Es la ruta en relación al directorio actual del proyecto Java (es decir, el «working directory»).

Por ejemplo, si nuestro proyecto está en:

C:/Users/cristian/IdeaProjects/AccesoADatos/

Y usamos la ruta relativa:

"datos/alumnos.txt"

Java buscará el archivo en:

C:/Users/cristian/IdeaProjects/AccesoADatos/datos/alumnos.txt

Ventaja: Es portable entre sistemas y ordenadores.

Desventaja: Puede ser difícil de localizar si no se sabe cuál es el directorio actual.

#### Cómo obtener el directorio actual del proyecto

```
String base = System.getProperty("user.dir");
System.out.println("Ruta base del proyecto: " + base);
```

## Portabilidad con File.separator

Los separadores de carpetas son diferentes según el sistema operativo:

Windows → \\

Linux/Mac → /

En lugar de escribirlo manualmente, usamos:

String separador = File.separator;

### Ejemplo:

```
import java.io.File;

public class RutasEjemplo {
    public static void main(String[] args) {
        String base = System.getProperty("user.dir");
        String sep = File.separator;

        String rutaAbsoluta = base + sep + "datos" + sep +
"ejemplo.txt";
        System.out.println("Ruta absoluta generada: " +
rutaAbsoluta);

        File archivo = new File(rutaAbsoluta);
        System.out.println("¿Existe el archivo? " +
archivo.exists());
    }
}

Ruta absoluta generada: C:\Users\crist\Desktop\java\ACCESO A DATOS\prueba\datos\ejemplo.txt
¿Existe el archivo? false
```

Este código construye una ruta de forma portátil y verifica si el archivo existe.

String separador = File.separator;

String rutaBase = System.getProperty("user.dir");

String ruta = rutaBase + separador + "archivo.txt";

### Ejemplo práctico completo:

```
import java.io.*;

public class GuardarLeerTexto {
    public static void main(String[] args) throws IOException {
        String ruta = System.getProperty("user.dir") + File.separator +
"ejemplo.txt";

        // Escritura
        BufferedWriter bw = new BufferedWriter(new FileWriter(ruta));
        bw.write("Primera linea\n");
    }
}
```

```
bw.write("Segunda linea\n");
bw.close();

// Lectura
BufferedReader br = new BufferedReader(new FileReader(ruta));
String linea;
while ((linea = br.readLine()) != null) {
    System.out.println(linea);
}
br.close();
}

}
```

```
C:\Users\crist\.jdks\open
Primera linea
Segunda linea
```

## Tema 2: Gestión de Ficheros y Directorios con la Clase File

### Descripción

Este tema aborda el uso de la clase File del paquete java.io, fundamental para gestionar archivos y carpetas en Java. Aprenderemos a crear, renombrar, eliminar y comprobar la existencia de ficheros, así como a explorar directorios.

### Introducción

Java proporciona la clase File para representar rutas de archivos y directorios. A través de esta clase es posible interactuar con el sistema de archivos de manera portable y segura. Aunque la clase File no permite leer ni escribir contenido directamente, sí facilita la gestión del sistema de ficheros.

#### Permite:

Comprobar si un archivo o carpeta existe.

Crear nuevos archivos o carpetas.

Eliminar elementos del sistema de ficheros.

Consultar el nombre, tamaño, ruta o permisos de un archivo.

## Gestión de archivos con [java.io.File](#)

### 1. Crear una ruta con File

File archivo = new File("datos/ejemplo.txt");

Esto no crea el archivo en sí, solo representa esa ruta. El archivo puede existir o no.

### 2. Métodos útiles para archivos

```
archivo.exists();      // ¿Existe el archivo?  
archivo.createNewFile(); // Crea el archivo si no existe  
archivo.delete();     // Elimina el archivo  
archivo.getName();    // Nombre del archivo  
archivo.getPath();    // Ruta relativa  
archivo.getAbsolutePath(); // Ruta absoluta  
archivo.length();     // Tamaño en bytes  
archivo.canRead();     // ¿Tiene permiso de lectura?  
archivo.canWrite();    // ¿Tiene permiso de escritura?
```

### Ejemplo Gestión de un archivo

```
import java.io.File;  
import java.io.IOException;  
  
public class GestionArchivo {  
    public static void main(String[] args) {  
        try {  
            File archivo = new File("datos/archivo_nuevo.txt");  
  
            if (!archivo.exists()) {  
                archivo.createNewFile();  
                System.out.println("Archivo creado: " +  
archivo.getAbsolutePath());  
            } else {  
                System.out.println("Ya existe: " +  
archivo.getAbsolutePath());  
            }  
  
            System.out.println("¿Es archivo?: " + archivo.isFile());  
            System.out.println("¿Se puede leer?: " +  
archivo.canRead());  
        }  
    }  
}
```

```

        System.out.println("¿Se puede escribir?: " +
archivo.canWrite());
        System.out.println("Tamaño: " + archivo.length() + " bytes");

    } catch (IOException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
}

```

### 3. Gestión de directorios

```

File carpeta = new File("datos/nueva_carpeta");
carpeta.mkdir();      // Crea la carpeta
carpeta.exists();     // ¿Existe?
carpeta.isDirectory(); // ¿Es un directorio?
Para listar archivos dentro de una carpeta:

```

```

String[] lista = carpeta.list();
for (String nombre : lista) {
    System.out.println(nombre);
}

```

#### Ejemplo: Listado del contenido de una carpeta

```

import java.io.File;

public class GestionCarpeta {
    public static void main(String[] args) {
        File carpeta = new File("datos/listado");

        if (!carpeta.exists()) {
            carpeta.mkdir();
            System.out.println("Carpeta creada: " +
carpeta.getAbsolutePath());
        } else {
            System.out.println("Carpeta ya existente: " +
carpeta.getAbsolutePath());
        }

        File[] contenido = carpeta.listFiles();
        System.out.println("Contenido de la carpeta:");
        if (contenido != null) {
            for (File f : contenido) {
                System.out.println("> " + f.getName() +
(f.isDirectory() ? " (dir)" : ""));
            }
        }
    }
}

```

```

        }
    }
}

```

```

Carpeta creada: C:\Users\crist\Desktop\java\ACCESO A DATOS\prueba\datos\listado
Contenido de la carpeta:

```

### Resumen

La clase File permite gestionar archivos y directorios del sistema operativo desde Java. Aunque no permite leer o escribir contenido, sí es muy útil para realizar operaciones como crear carpetas, comprobar permisos o eliminar elementos del sistema.

### Tabla resumen de métodos comunes:

#### Tabla resumen de métodos comunes

Método	Descripción
<code>exists()</code>	Comprueba si el archivo/carpeta existe
<code>createNewFile()</code>	Crea un nuevo archivo
<code>mkdir()</code>	Crea un nuevo directorio
<code>delete()</code>	Elimina el archivo o directorio
<code>isDirectory()</code>	¿Es un directorio?
<code>isFile()</code>	¿Es un archivo?
<code>getName()</code>	Devuelve el nombre del archivo
<code>getAbsolutePath()</code>	Devuelve la ruta absoluta
<code>length()</code>	Tamaño en bytes
<code>canRead()</code>	¿Tiene permiso de lectura?
<code>canWrite()</code>	¿Tiene permiso de escritura?

# Tema 3: Lectura y Escritura Secuencial de Ficheros

## Descripción

Este tema se centra en las operaciones básicas de lectura y escritura secuencial sobre archivos de texto en Java. Utilizaremos las clases FileReader, BufferedReader, FileWriter y BufferedWriter para manipular datos de manera eficiente.

## Introducción

La lectura y escritura secuencial implica acceder a un archivo desde el principio hasta el final, en orden. Es el modo más habitual para trabajar con archivos de texto donde se procesan líneas completas o caracteres de forma lineal.

Java proporciona diferentes clases dentro del paquete `java.io` para realizar estas tareas de forma eficiente, utilizando buffers que reducen el acceso físico al disco y mejoran el rendimiento.

## Escritura secuencial de ficheros

Para escribir texto en un archivo utilizamos:

- **FileWriter**: conecta el programa con el archivo.
- **BufferedWriter**: permite escribir texto en bloque (más eficiente).

### Ejemplo

```
import java.io.*;

public class EscrituraFichero {
    public static void main(String[] args) {
        try {
            FileWriter fw = new FileWriter("datos/salida.txt");
            BufferedWriter bw = new BufferedWriter(fw);

            bw.write("Primera línea");
            bw.newLine();
            bw.write("Segunda línea");
            bw.newLine();

            bw.flush(); // Forzar la escritura
            bw.close(); // Cerrar el buffer
        }
    }
}
```

```

        System.out.println("Archivo escrito correctamente.");
    } catch (IOException e) {
        System.out.println("Error al escribir: " +
e.getMessage());
    }
}

```

## Lectura secuencial de ficheros

Para leer archivos línea a línea:

**FileReader**: abre el archivo.

**BufferedReader**: permite leer una línea completa con `readLine()`.

```

import java.io.*;

public class LecturaFichero {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("datos/salida.txt");
            BufferedReader br = new BufferedReader(fr);

            String linea;
            while ((linea = br.readLine()) != null) {
                System.out.println("> " + linea);
            }

            br.close();
        } catch (IOException e) {
            System.out.println("Error al leer: " + e.getMessage());
        }
    }
}

```

## Consideraciones importantes

- Siempre cerrar los streams (`close()`).
- Utilizar try-catch para capturar errores de entrada/salida.

- flush() garantiza que lo que está en el buffer se escribe en disco.
- Las rutas deben existir o crearse antes de escribir.

Usar newLine() en lugar de \n asegura compatibilidad multiplataforma.

## Resumen

Este tema ha presentado las clases necesarias para realizar escritura y lectura secuencial de archivos de texto en Java. Estas operaciones son fundamentales para guardar o recuperar datos simples en forma de texto plano.

## Tabla comparativa

Clase	Función
<code>FileWriter</code>	Escribe caracteres en un archivo
<code>BufferedWriter</code>	Mejora el rendimiento usando un buffer
<code>FileReader</code>	Lee caracteres desde un archivo
<code>BufferedReader</code>	Permite leer líneas completas eficientemente

## Tema 4: Acceso Aleatorio a Ficheros

### Descripción

Este tema trata sobre cómo acceder a posiciones específicas dentro de un archivo utilizando la clase RandomAccessFile de Java. Este tipo de acceso es ideal cuando se necesita leer o escribir en ubicaciones concretas sin procesar todo el archivo secuencialmente.

### Introducción

A diferencia del acceso secuencial, el acceso aleatorio permite mover un puntero directamente a una posición determinada dentro del archivo. Esto es útil cuando trabajamos con registros de longitud fija o necesitamos modificar información puntual sin recorrer el archivo completo.

La clase **RandomAccessFile** ofrece métodos para leer y escribir datos primitivos en posiciones arbitrarias.

## Creación y escritura en un archivo aleatorio

RandomAccessFile permite abrir un archivo en modo lectura ("r") o lectura/escritura ("rw"). Podemos usar seek(long pos) para movernos por el archivo y writeXXX() o readXXX() para trabajar con datos.

### Ejemplo: Escribir varias líneas

```
import java.io.*;

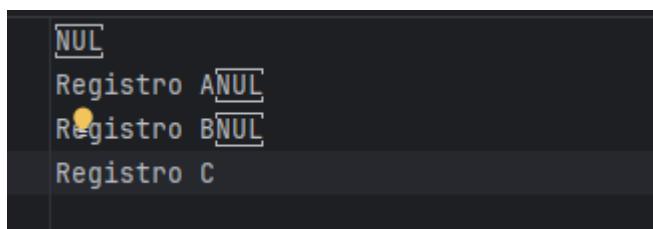
public class EscribirAleatorio {
    public static void main(String[] args) {
        try {
            RandomAccessFile raf = new
RandomAccessFile("datos/aleatorio.txt", "rw");

            raf.writeUTF("Registro A");
            raf.writeUTF("Registro B");
            raf.writeUTF("Registro C");

            raf.close();
            System.out.println("Registros escritos con éxito.");
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Resultado:

Registros escritos con éxito.



```
NUL
Registro A NUL
Registro B NUL
Registro C
```

## Lectura desde una posición específica

Podemos acceder directamente a cualquier posición del archivo utilizando seek(pos).

### Ejemplo: Leer desde una posición concreta

```
import java.io.*;

public class LeerAleatorio {
    public static void main(String[] args) {
        try {
            RandomAccessFile raf = new
RandomAccessFile("datos/aleatorio.txt", "r");

            raf.seek(0); // leer desde el principio
            System.out.println("Primer registro: " +
raf.readUTF());

            raf.seek(raf.getFilePointer()); // continuar donde se
quedó
            System.out.println("Segundo registro: " +
raf.readUTF());

            raf.close();
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```
Primer registro: Registro A
Segundo registro: Registro B
```

## Consideraciones importantes

- **RandomAccessFile** no utiliza buffering, por lo que puede ser más lento para operaciones repetidas.
- Los métodos como **writeUTF** y **readUTF** usan un formato binario específico con encabezados de longitud.
- Si el archivo no existe y se abre en modo "**rw**", se crea automáticamente.
- No es adecuado para archivos de texto plano sin estructura fija.

### Resumen

RandomAccessFile proporciona un mecanismo potente para manipular archivos binarios y trabajar con datos estructurados directamente en disco. Es especialmente útil en sistemas que requieren acceso directo a registros individuales.

## Tabla Resumen:

Método	Descripción
<code>seek(pos)</code>	Mueve el puntero de lectura/escritura
<code>readUTF()</code>	Lee una cadena codificada como UTF
<code>writeUTF(String)</code>	Escribe una cadena codificada como UTF
<code>getFilePointer()</code>	Devuelve la posición actual del puntero
<code>length()</code>	Devuelve el tamaño total del archivo

## Unidad Extra : Clase Scanner

### Descripción

La clase Scanner, introducida en Java 5 y perteneciente al paquete `java.util`, es una de las herramientas más utilizadas para leer datos de diferentes fuentes: teclado, archivos, cadenas de texto o incluso flujos de entrada (`InputStream`).

Su principal ventaja es la simplicidad de uso, ya que permite leer tanto cadenas de texto como valores numéricos directamente, sin necesidad de realizar conversiones manuales, algo que sí ocurre con otras clases de entrada como `BufferedReader`.

Por ello, Scanner es muy útil en entornos educativos, en programas interactivos y en proyectos donde se requiere flexibilidad en la captura de datos.

### Fundamentos de la clase Scanner

- Pertenece al paquete `java.util`.
- Permite leer datos de diferentes tipos (`int`, `double`, `String`, etc.).
- Se construye a partir de una fuente de entrada:

\*\* Teclado: **new Scanner(System.in)**

\*\* Archivo: **new Scanner(new File("ruta.txt"))**

\*\* Cadena de texto: **new Scanner("texto inicial")**

## Métodos principales de Scanner

Método	Descripción
<code>next()</code>	Lee la siguiente palabra (hasta espacio)
<code>nextLine()</code>	Lee una línea completa
<code>nextInt()</code>	Lee un número entero
<code>nextDouble()</code>	Lee un número decimal
<code>hasNext()</code>	Devuelve <code>true</code> si hay más datos disponibles
<code>hasNextInt()</code>	Verifica si el siguiente token es un entero
<code>close()</code>	Cierra el objeto <code>Scanner</code>

## Uso de Scanner para leer desde teclado

```
import java.util.Scanner;

public class EjemploScanner {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Introduce tu nombre: ");
        String nombre = sc.nextLine();

        System.out.print("Introduce tu edad: ");
        int edad = sc.nextInt();

        System.out.println("Hola " + nombre + ", tienes " + edad +
" años.");

        sc.close(); // Siempre cerrar al final
    }
}
```

```
Introduce tu nombre: cristian
Introduce tu edad: 21
Hola cristian, tienes 21 años.
```

## Observaciones importantes

- Es recomendable cerrar siempre el Scanner al final del programa.
- Si se mezclan **nextInt()** y **nextLine()**, es necesario limpiar el buffer de entrada con una llamada extra a **nextLine()**.

## Uso de Scanner para leer desde archivos

Además de leer del teclado, Scanner puede leer directamente desde archivos:

```
import java.io.File;
import java.util.Scanner;

public class LeerArchivoScanner {
    public static void main(String[] args) {
        try {
            File archivo = new File("datos/entrada.txt");
            Scanner sc = new Scanner(archivo);

            while (sc.hasNextLine()) {
                String linea = sc.nextLine();
                System.out.println("Línea: " + linea);
            }

            sc.close();
        } catch (Exception e) {
            System.out.println("Error al leer archivo: " +
e.getMessage());
        }
    }
}
```

**prueba scanner**

```
c:\users\crist\.jdks\openjdk-2
Línea: prueba scanner
```

## Ventajas frente a otras clases de lectura de archivos

- Scanner permite leer tokens individuales (palabras, números, etc.) fácilmente.
- Se pueden aplicar separadores personalizados con el método **useDelimiter()**.
- Sin embargo, para archivos muy grandes, **BufferedReader** suele ser más eficiente.

## Creación de menús interactivos con Scanner

Una aplicación típica en FP es la construcción de menús de consola.

```
import java.util.Scanner;

public class MenuEjemplo {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int opcion;

        do {
            System.out.println("==== MENÚ PRINCIPAL ====");
            System.out.println("1. Saludar");
            System.out.println("2. Calcular suma");
            System.out.println("3. Salir");
            System.out.print("Elige una opción: ");
            opcion = sc.nextInt();

            switch (opcion) {
                case 1:
                    System.out.println("¡Hola usuario!");
                    break;
                case 2:
                    System.out.print("Introduce un número: ");
                    int a = sc.nextInt();
                    System.out.print("Introduce otro número: ");
                    int b = sc.nextInt();
                    System.out.println("La suma es: " + (a + b));
                    break;
                case 3:
                    System.out.println("Saliendo del programa...");
                    break;
                default:
                    System.out.println("Opción inválida.");
            }
        } while (opcion != 3);

        sc.close();
    }
}
```

```

==== MENÚ PRINCIPAL ====
1. Saludar
2. Calcular suma
3. Salir
Elige una opción: 1
¡Hola usuario!
==== MENÚ PRINCIPAL ====
1. Saludar
2. Calcular suma
3. Salir
Elige una opción: 2
Introduce un número: 3
Introduce otro número: 4
La suma es: 7
==== MENÚ PRINCIPAL ====
1. Saludar
2. Calcular suma
3. Salir
Elige una opción: 3
Saliendo del programa...

```

### Comparación con otras clases de entrada

Clase	Ventajas	Inconvenientes
Scanner	Fácil de usar, soporta tipos primitivos	Más lento que BufferedReader
BufferedReader	Muy eficiente en lectura de texto	Necesita conversión manual de tipos
FileReader	Lectura directa de archivos	Requiere envoltorios (buffers)

### Recomendaciones:

- Cerrar siempre con sc.close() para liberar recursos.
- Si se usan métodos mixtos (nextInt y nextLine), limpiar el buffer.
- Para proyectos grandes, combinarlo con clases POJO para estructurar datos.
- Evitar su uso en aplicaciones con ficheros enormes, donde BufferedReader es más eficiente.

# Tema 4.1: Alternativa moderna con NIO - Gestión de Archivos con Path y Files

## Descripción

Este subtema introduce el uso de la API **java.nio.file**, que representa una evolución moderna respecto a la clase **File**. Incluye las clases **Path**, **Files**, y **Paths**, que permiten una gestión más clara, robusta y multiplataforma de archivos y rutas.

## Introducción

**Java NIO (New Input/Output)** es una API introducida en Java 7 que mejora significativamente el rendimiento y la flexibilidad en las operaciones de entrada/salida. A diferencia de **java.io**, que se basa en flujos secuenciales, **java.nio** se apoya en conceptos como canales, búferes y selectores, ofreciendo una forma más potente de manejar datos.

Desde Java 7 se recomienda usar el paquete **java.nio.file** para el manejo de archivos. Las clases más utilizadas son:

- **Path**: representa una ruta de archivo o directorio.
- **Paths**: clase utilitaria para obtener objetos Path.
- **Files**: contiene métodos estáticos para trabajar con archivos y directorios (leer, escribir, mover, borrar, copiar, etc).

Este enfoque permite trabajar de forma más segura y eficiente, con soporte para codificaciones como UTF-8, operaciones atómicas, y uso con streams de Java 8.

## ¿Por qué usar Java NIO?

- **Mayor rendimiento**: gracias a la lectura/escritura mediante búferes y operaciones por bloques
- **Operaciones no bloqueantes**: permite continuar ejecutando otras tareas mientras se completan las operaciones de E/S.
- **Soporte para aplicaciones de red**: ideal para servidores y clientes concurrentes.
- **Capacidad para manejar grandes volúmenes de datos**.

# Conceptos clave en Java NIO

## Canales (Channel)

Un canal representa una conexión con una entidad de E/S, como un archivo, un socket o un dispositivo. Se puede leer y escribir datos en el canal de forma más directa que con los streams tradicionales.

Ejemplo: **FileChannel**, **SocketChannel**, **DatagramChannel**.

## Búferes (Buffer)

Los búferes son bloques de memoria que permiten almacenar datos de entrada o salida temporalmente. Cada canal está vinculado a un búfer, el cual facilita operaciones por bloques en lugar de byte a byte.

Ejemplo: **ByteBuffer**, **CharBuffer**, **IntBuffer**.

## Selectores (Selector)

Permiten supervisar múltiples canales de forma no bloqueante. Son especialmente útiles en aplicaciones de red (como servidores) donde se deben gestionar múltiples conexiones simultáneamente con eficiencia.

## ¿Cuándo utilizar Java NIO?

- Aplicaciones de red multicliente (chat, servidores web).
- Manipulación de archivos de gran tamaño.
- Aplicaciones de alto rendimiento: trading, juegos, procesadores de datos.
- Operaciones asíncronas o concurrentes de E/S.
- Ejemplo básico: uso de Path y Files

## Ejemplo básico : uso de Path y Files

```
import java.nio.file.*;  
  
public class CrearArchivoNIO {  
    public static void main(String[] args) {  
        try {  
            Path ruta = Paths.get("datos/archivo_nio.txt");  
  
            if (!Files.exists(ruta)) {  
                Files.createDirectories(ruta.getParent());  
                Files.createFile(ruta);  
                System.out.println("Archivo creado.");  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

        } else {
            System.out.println("El archivo ya existe.");
        }

        System.out.println("Nombre: " + ruta.getFileName());
        System.out.println("Ruta absoluta: " +
ruta.toAbsolutePath());
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
}
}

```

Archivo creado.  
Nombre: archivo\_nio.txt  
≡ archivo\_nio.txt

## Ejemplo: lectura y escritura con Files y UTF-8

```

import java.nio.file.*;
import java.util.List;
import java.nio.charset.StandardCharsets;
import java.io.IOException;

public class LecturaEscrituraNIO {
    public static void main(String[] args) {
        Path ruta = Paths.get("datos/nio_contenido.txt");

        try {
            List<String> lineas = List.of("Línea A", "Línea B",
"Línea C");
            Files.write(ruta, lineas, StandardCharsets.UTF_8);
            System.out.println("Contenido escrito.");
        }

        List<String> leidas = Files.readAllLines(ruta,
StandardCharsets.UTF_8);
        System.out.println("Contenido leído:");
        for (String linea : leidas) {
            System.out.println("> " + linea);
        }
    } catch (IOException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
}

```

```
Contenido escrito.  
Contenido leido:  
> Línea A  
> Línea B  
> Línea C
```

## Ejemplo con canal y bufer

```
import java.nio.file.*;  
import java.nio.channels.SeekableByteChannel;  
import java.nio.ByteBuffer;  
import java.io.IOException;  
  
public class NIOExample {  
    public static void main(String[] args) throws IOException {  
        Path path = Paths.get("miArchivo.txt");  
  
        try (SeekableByteChannel channel =  
             Files.newByteChannel(path, StandardOpenOption.WRITE,  
             StandardOpenOption.CREATE)) {  
            byte[] data = "Hola, mundo!".getBytes();  
            ByteBuffer buffer = ByteBuffer.wrap(data);  
            channel.write(buffer);  
        }  
    }  
}
```

## Ventajas de NIO sobre File

Característica	File	Path + Files (NIO)
Manejo de rutas	Limitado	Muy flexible, multiplataforma
Crear archivos	.createNewFile()	Files.createFile(path)
Leer/escribir todo	No nativo	Files.readAllLines, write
UTF-8 por defecto	No	Sí (con charset configurable)
Streams	No	Sí (Files.lines(), Files.walk())
Seguridad	Menos chequeos	Más validaciones

# Tema 5: Ficheros Binarios y Serialización de Objetos

## Descripción

Este tema se centra en el tratamiento de ficheros binarios y en la serialización de objetos en Java, permitiendo guardar estructuras complejas (como listas, objetos y árboles) en disco y recuperarlas más tarde.

## Introducción

Cuando se desea guardar información más estructurada que simples cadenas de texto, una opción eficiente es utilizar ficheros binarios. En estos archivos los datos se guardan en formato no legible para humanos, pero mucho más eficiente para la lectura/escritura por parte del programa.

Además, Java permite guardar objetos completos mediante un proceso llamado serialización, que convierte un objeto en una secuencia de bytes para almacenarlo o enviarlo a través de la red.

## ¿Qué es la serialización?

**La serialización** es el proceso mediante el cual un objeto en memoria se transforma en una secuencia de bytes que puede guardarse en un fichero o transmitirse. Su operación inversa es la deserialización, que reconstruye el objeto original a partir del flujo de datos.

Para que una clase sea serializable debe:

- Implementar la interfaz **java.io.Serializable**.
- Tener un identificador **serialVersionUID** (recomendado).

## Escritura de objetos (serialización)

```
import java.io.*;

public class EscribirObjeto {
    public static void main(String[] args) {
        Persona p = new Persona("Ana", 30);

        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("datos/persona.dat"))) {
            oos.writeObject(p);
```

```

        System.out.println("Objeto serializado
correctamente.");
    } catch (IOException e) {
        System.out.println("Error al escribir objeto: " +
e.getMessage());
    }
}

class Persona implements Serializable {
    private static final long serialVersionUID = 1L;
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}

```

Objeto serializado correctamente.

## Lectura de objetos (deserialización)

```

import java.io.*;

public class LeerObjeto {
    public static void main(String[] args) {
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("datos/persona.dat"))) {
            Persona p = (Persona) ois.readObject();
            System.out.println("Nombre: " + p.getNombre());
            System.out.println("Edad: " + p.getEdad());
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Error al leer objeto: " +
e.getMessage());
        }
    }
}

```

### Ventajas de la serialización

- Permite almacenar el estado completo de objetos complejos.
- Muy útil para guardar listas, colecciones y objetos anidados.
- Fácil de implementar con ObjectOutputStream y ObjectInputStream.

## Consideraciones

- Todos los objetos incluidos deben ser también Serializable.
- No se deben serializar objetos con conexiones abiertas, hilos, etc.
- Puede romperse la compatibilidad si cambia la estructura de la clase.

## Tabla resumen

Clase	Descripción
<code>Serializable</code>	Interfaz que marca una clase como serializable
<code>ObjectOutputStream</code>	Permite escribir objetos en un flujo binario
<code>ObjectInputStream</code>	Permite leer objetos desde un flujo binario
<code>writeObject(obj)</code>	Serializa un objeto y lo guarda en el archivo
<code>readObject()</code>	Recupera el objeto previamente serializado

## Tema 6: Lectura y Escritura de Ficheros XML

### Descripción

Este tema aborda el tratamiento de ficheros XML en Java, presentando los distintos modelos de análisis disponibles: DOM, SAX y StAX, así como APIs modernas como JAXP y JAXB. También se introducen conceptos como XPath y el uso de POJOs para mapear estructuras XML.

### Introducción

**XML (eXtensible Markup Language)** es un formato estándar para almacenar y transmitir datos estructurados. Es ampliamente usado en interoperabilidad de aplicaciones, configuraciones, intercambio de información entre sistemas y servicios web.

Java proporciona múltiples formas de manipular XML, dependiendo de si necesitamos procesar todo el documento en memoria o manejar grandes volúmenes de datos línea por línea.

## Tecnologías para procesar XML en Java

Tecnología	Descripción breve	Ventajas	Inconvenientes
DOM (Document Object Model)	Carga el XML completo en memoria como un árbol	Acceso aleatorio, permite modificar el XML	Alto consumo de memoria
SAX (Simple API for XML)	Analiza el XML como flujo de eventos	Bajo consumo de memoria	No permite modificar ni navegar hacia atrás
StAX	Permite lectura y escritura controlada de eventos	Control completo, combinación de SAX y DOM	Mayor complejidad en el código
JAXP	API general para procesar XML con DOM o SAX	Flexible y estándar	Requiere conocer el modelo subyacente
JAXB	Permite mapear clases Java a XML y viceversa	Muy productivo con anotaciones	Requiere XSD y configuración previa
XPath	Lenguaje para buscar nodos dentro de un XML	Precisión, potente en combinación con DOM	Solo lectura

### Estructura de un XML de ejemplo

```
<libros>
  <libro>
    <titulo>Java Avanzado</titulo>
    <autor>Juan Pérez</autor>
  </libro>
  <libro>
    <titulo>XML en Java</titulo>
    <autor>Ana Gómez</autor>
  </libro>
</libros>
```

## Lectura de XML con DOM

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.*;

import java.io.File;

public class LeerXML_DOM {
    public static void main(String[] args) {
        try {
            File archivo = new File("datos/libros.xml");
```

```

        DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document doc = builder.parse(archivo);

        NodeList lista = doc.getElementsByTagName("libro");
        for (int i = 0; i < lista.getLength(); i++) {
            Element libro = (Element) lista.item(i);
            String titulo =
libro.getElementsByTagName("titulo").item(0).getTextContent();
            String autor =
libro.getElementsByTagName("autor").item(0).getTextContent();
            System.out.println("Libro: " + titulo + ", Autor: "
+ autor);
        }

    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
}
}

```

```

C:\Users\crist\.jdks\openjdk-25\bin\java.e:
Libro: Java Avanzado, Autor: Juan Pérez
Libro: XML en Java, Autor: Ana Gómez

```

**libros.xml:**

```

<libros>
    <libro>
        <titulo>Java Avanzado</titulo>
        <autor>Juan Pérez</autor>
    </libro>
    <libro>
        <titulo>XML en Java</titulo>
        <autor>Ana Gómez</autor>
    </libro>
</libros>

```

## Lectura con SAX (manejador de eventos)

```

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import java.io.File;

```

```

public class LeerXML_SAX {
    public static void main(String[] args) {
        try {
            File archivo = new File("datos/libros.xml");
            SAXParserFactory factory =
SAXParserFactory.newInstance();
            SAXParser parser = factory.newSAXParser();

            DefaultHandler handler = new DefaultHandler() {
                boolean titulo = false;
                boolean autor = false;

                public void startElement(String uri, String
localName, String qName, Attributes attributes) {
                    if (qName.equals("titulo")) titulo = true;
                    if (qName.equals("autor")) autor = true;
                }

                public void characters(char[] ch, int start, int
length) {
                    if (titulo) {
                        System.out.println("Título: " + new
String(ch, start, length));
                        titulo = false;
                    }
                    if (autor) {
                        System.out.println("Autor: " + new
String(ch, start, length));
                        autor = false;
                    }
                }
            };

            parser.parse(archivo, handler);
        } catch (Exception e) {
            System.out.println("Error SAX: " + e.getMessage());
        }
    }
}

```

**Título:** Java Avanzado  
**Autor:** Juan Pérez  
**Título:** XML en Java  
**Autor:** Ana Gómez

## Resumen

- Java ofrece distintas estrategias para leer y manipular XML. La elección depende de:
- Si necesitamos leer o modificar el XML.
- El tamaño del archivo y la eficiencia requerida.

El uso de herramientas como XSD, anotaciones y POJOs.

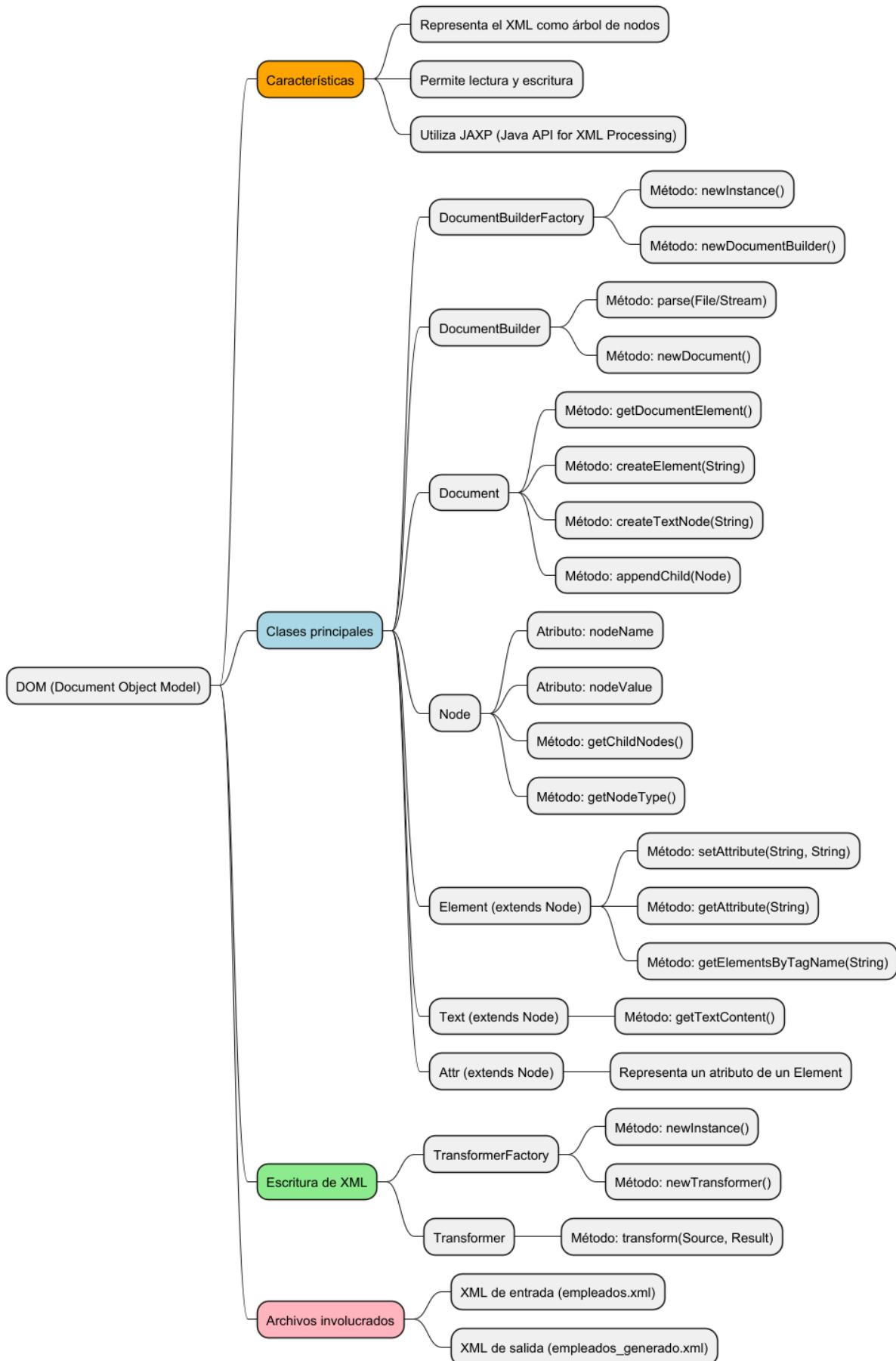
## Tabla comparativa

Tecnología	Memoria	Modificación	Precisión	Complejidad
DOM	Alta	Sí	Media	Baja
SAX	Baja	No	Media	Media
StAX	Media	Sí (eventos)	Alta	Alta
JAXB	Media	Sí (POJO)	Alta	Baja
XPath	Media	No	Muy alta	Media

## Tema 6.1: Procesamiento XML con DOM (Document Object Model)

### Descripción

Esta unidad profundiza en el uso del modelo DOM para la lectura, análisis, modificación y escritura de documentos XML en Java. Se estudian las clases fundamentales, su estructura jerárquica, y se realizan ejercicios paso a paso para manipular nodos, atributos y contenido textual.



## ¿Qué es DOM?

**DOM (Document Object Model)** es un modelo de estructura de árbol en memoria que representa todos los elementos de un documento XML como nodos organizados jerárquicamente.

- Cada elemento, atributo o contenido textual es un nodo.
- Permite acceso aleatorio, lectura, modificación y escritura.
- Utiliza la API estándar de Java JAXP (javax.xml.parsers, org.w3c.dom, javax.xml.transform).

### Clases principales y su uso

Clase	Descripción
<code>DocumentBuilderFactory</code>	Crea instancias para analizar XML
<code>DocumentBuilder</code>	Construye un <code>Document</code> desde un archivo
<code>Document</code>	Representa todo el documento XML
<code>Node</code>	Nodo genérico (base de todos los nodos)
<code>Element</code>	Nodo que representa una etiqueta XML
<code>Attr</code>	Nodo atributo de un elemento
<code>Text</code>	Nodo que contiene texto
<code>TransformerFactory</code>	Crea transformadores para escribir XML
<code>Transformer</code>	Transforma <code>Document</code> a fichero de salida

## Lectura de un documento XML con DOM

```
<empleados>
    <empleado id="E101">
        <nombre>Juan Pérez</nombre>
        <puesto>Analista</puesto>
    </empleado>
    <empleado id="E102">
        <nombre>María López</nombre>
        <puesto>Diseñadora</puesto>
    </empleado>
</empleados>
```

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
```

```

import java.io.File;

public class DOMLeerEjemplo {
    public static void main(String[] args) {
        try {
            File archivo = new File("datos/empleados.xml");
            DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            Document doc = db.parse(archivo);

            NodeList lista = doc.getElementsByTagName("empleado");

            for (int i = 0; i < lista.getLength(); i++) {
                Element emp = (Element) lista.item(i);
                String id = emp.getAttribute("id");
                String nombre =
emp.getElementsByTagName("nombre").item(0).getTextContent();
                String puesto =
emp.getElementsByTagName("puesto").item(0).getTextContent();
                System.out.println("ID: " + id + " | Nombre: " +
nombre + " | Puesto: " + puesto);
            }

        } catch (Exception e) {
            System.out.println("Error al leer: " + e.getMessage());
        }
    }
}

```

ID: E101 | Nombre: Juan Pérez | Puesto: Analista  
ID: E102 | Nombre: María López | Puesto: Diseñadora

## Escritura de un documento XML con DOM

```

import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.*;
import java.io.File;

public class DOMEscribirEjemplo {
    public static void main(String[] args) {

```

```

try {
    DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
    DocumentBuilder db = dbf.newDocumentBuilder();
    Document doc = db.newDocument();

    Element root = doc.createElement("empleados");
    doc.appendChild(root);

    Element emp = doc.createElement("empleado");
    emp.setAttribute("id", "E001");

    Element nombre = doc.createElement("nombre");
    nombre.appendChild(doc.createTextNode("Ana Torres"));
    emp.appendChild(nombre);

    Element puesto = doc.createElement("puesto");

    puesto.appendChild(doc.createTextNode("Desarrolladora"));
    emp.appendChild(puesto);

    root.appendChild(emp);

    TransformerFactory tf =
TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();
    DOMSource source = new DOMSource(doc);
    StreamResult result = new StreamResult(new
File("datos/empleados_generado.xml"));
    transformer.transform(source, result);

    System.out.println("Archivo XML generado
correctamente.");
}

} catch (Exception e) {
    System.out.println("Error al escribir: " +
e.getMessage());
}
}
}

```

Archivo XML generado correctamente.

empleados generado

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><empleados><
```

## Esquema del flujo de trabajo DOM

1. Crear un DocumentBuilderFactory
2. Obtener un DocumentBuilder
3. Analizar el XML y obtener un Document
4. Navegar con getElementsByTagName, getAttribute, getChildNodes
5. Modificar con createElement, appendChild, setAttribute
6. Guardar con Transformer (usando DOMSource y StreamResult)

## Archivos utilizados

- **Entrada:** datos/empleados.xml
- **Salida:** datos/empleados\_generado.xml

Puedes usar el siguiente contenido de ejemplo como archivo de entrada:

```
<empleados>
  <empleado id="E101">
    <nombre>Juan Pérez</nombre>
    <puesto>Analista</puesto>
  </empleado>
  <empleado id="E102">
    <nombre>María López</nombre>
    <puesto>Diseñadora</puesto>
  </empleado>
</empleados>
```

El modelo DOM permite trabajar con documentos XML en memoria de forma jerárquica y flexible. Es ideal cuando se necesita leer, modificar y volver a guardar el documento completo. Sus métodos permiten recorrer nodos, crear elementos y atributos, y generar nuevos archivos XML.

# ACTIVIDADES

## UD1 - Actividad Práctica: Tema 1.1

LecturaSecuencial:

```
import java.io.*;
public class LecturaSecuencial {
    public static void main(String[] args) {
        try {
            File archivo = new File("datos.txt");
            BufferedReader br = new BufferedReader(new
                FileReader(archivo));
            String linea;
            System.out.println("Lectura completa del archivo (modo
secuencial):");
            while ((linea = br.readLine()) != null) {
                System.out.println("> " + linea);
            }
            br.close();
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " +
                e.getMessage());
        }
    }
}
```

```
Lectura completa del archivo (modo secuencial):
> linea 1
> linea 2
> linea 3
> linea 4
> linea 5
```

AccesoAleatorio.java:

```
import java.io.*;
public class AccesoAleatorio {
    public static void main(String[] args) {
        try {
```

```

RandomAccessFile raf = new
RandomAccessFile("datos.txt",
                "r");
// Cambia el valor para probar: 0, 10, 15, 30, etc.
long posicion = 15;
raf.seek(posicion); // Mover el puntero al byte 15
System.out.println("Lectura desde byte " + posicion + ":");
String linea = raf.readLine(); // Leer desde esa
posición
System.out.println("> " + linea);
raf.close();
} catch (IOException e) {
    System.out.println("Error en acceso aleatorio: " +
                       e.getMessage());
}
}
}
}

```

0,10,15,30

¿Qué línea se imprime?

Lectura desde byte 0:  
> linea 1

Lectura desde byte 10:  
> linea 2

Lectura desde byte 15:  
> 2

Process finished with exit code 0

Lectura desde byte 30:  
> ea 4

¿Aparece una línea cortada?

si

¿Por qué cambia el resultado según el valor de seek()?  
por que mueve el puntero de lectura a un byte específico

# UD1 - Actividad práctica Tema 1.2 - Uso de rutas en java

Aprender a construir rutas absolutas y relativas en Java de forma **portable y multiplataforma**, utilizando `System.getProperty()` y `File.separator`.

## Paso 1: Preparar el proyecto

1. Abre tu proyecto en IntelliJ IDEA.
2. Crea una carpeta dentro de la raíz del proyecto llamada:

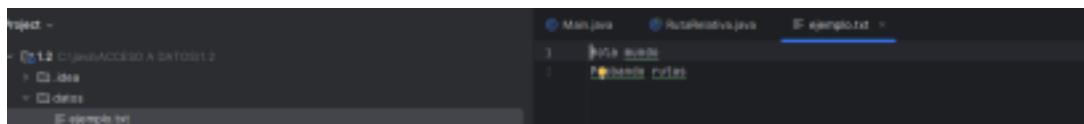
datos

3. Dentro de esa carpeta, crea un archivo llamado:

ejemplo.txt

Y añade dos líneas de texto, por ejemplo:

Hola mundo  
Probando rutas



## Paso 2: Código Java

Crea una clase llamada `RutaRelativa` dentro del `src` y copia este código:

```
import java.io.File;

public class RutaRelativa {
    public static void main(String[] args) {
        // Obtener ruta base del proyecto
        String rutaBase = System.getProperty("user.dir");
        String separador = File.separator;

        // Construir ruta completa relativa
        String rutaRelativa = rutaBase + separador + "datos" +
        separador
        + "ejemplo.txt";

        // Crear objeto File con esa ruta
        File archivo = new File(rutaRelativa);
        // Mostrar información
        System.out.println("Ruta base del proyecto: " + rutaBase);
```

```

System.out.println("Separador de carpetas del sistema: " +
separador);
    System.out.println("Ruta relativa completa: " + rutaRelativa);
System.out.println("¿Existe el archivo? " + archivo.exists());
System.out.println("Ruta absoluta real: " +
archivo.getAbsolutePath());
}
}

import java.io.File;
public class RutaRelativa {
    public static void main(String[] args) {
// Obtener ruta base del proyecto
String rutaBase = System.getProperty("user.dir"); String
separador = File.separator;
// Construir ruta completa relativa
String rutaRelativa = rutaBase + separador + "datos" +
separador
+ "ejemplo.txt";
// Crear objeto File con esa ruta
File archivo = new File(rutaRelativa);
// Mostrar información
System.out.println("Ruta base del proyecto: " +
rutaBase);
System.out.println("Separador de carpetas del sistema: " +
separador);
System.out.println("Ruta relativa completa: " +
rutaRelativa);
System.out.println("¿Existe el archivo? " +
archivo.exists());
System.out.println("Ruta absoluta real: " +
archivo.getAbsolutePath());
}
}

```

## Resultado esperado en consola

Algo como:

```

Ruta base del proyecto:
C:\Users\Alumno\IdeaProjects\AccesoADatos
Separador de carpetas del sistema: \
Ruta relativa completa:
C:\Users\Alumno\IdeaProjects\AccesoADatos\datos\ejemplo.txt
¿Existe el archivo? true
Ruta absoluta real:
C:\Users\Alumno\IdeaProjects\AccesoADatos\datos\ejemplo.txt

```

```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2020.2.2\lib\idea_rt.jar=51499:file.encoding=UTF-8"
Ruta base del proyecto: C:\java\ACCESO A DATOS\I
Separador de carpetas del sistema: \
Ruta relativa completa: c:\java\ACCESO A DATOS\I\datoslejemplo.txt
Existe el archivo? true
Ruta absoluta real: C:\java\ACCESO A DATOS\I\datoslejemplo.txt
```

## Preguntas de reflexión

### 1. ¿Qué ocurriría si usas una ruta absoluta y cambias de ordenador o carpeta?

¿Seguiría funcionando tu código?

En otro ordenador esa ruta puede que no exista entonces si no existe no

¿Qué error verías?

java.io.FileNotFoundException

### 2. ¿Por qué es importante usar rutas relativas en aplicaciones reales?

¿Qué ventajas te aporta al distribuir tu aplicación?

Por que funciona en cualquier ordenador

### 3. ¿Qué ventaja tiene usar `File.separator` frente a escribir `/` o `\` directamente?

Tiene la ventaja que `file.separator` se adapta al sistema operativo

¿Funcionaría el mismo código en Windows, Linux y Mac sin cambiar nada?

Si lo introduces manualmente en vez de `file.separator` no funcionaría en algunos sistemas

## Extra (para los avanzados)

Sustituye `System.out.println(...)` por un log personalizado.

Muestra el nombre del archivo con `archivo.getName()`.

Usa `archivo.getParent()` para ver dónde está contenido.

```
import java.io.File;

public class RutaRelativa {

    // Método auxiliar para log
    private static void log(String mensaje) {
        System.out.println("[LOG] " + mensaje);
    }
}
```

```

public static void main(String[] args) {
    // Obtener ruta base del proyecto
    String rutaBase = System.getProperty("user.dir");   String
    separador = File.separator;

    // Construir ruta completa relativa
    String rutaRelativa = rutaBase + separador + "datos" +
    separador + "ejemplo.txt";

    // Crear objeto File con esa ruta
    File archivo = new File(rutaRelativa);

    // Mostrar información con log personalizado
    log("Ruta base del proyecto: " + rutaBase);
    log("Separador de carpetas del sistema: " +
    separador);
    log("Ruta relativa completa: " + rutaRelativa);
    log("¿Existe el archivo? " + archivo.exists());
    log("Ruta absoluta real: " +
    archivo.getAbsolutePath());
    log("Nombre del archivo: " + archivo.getName());
    log("Carpeta contenedora: " + archivo.getParent());  }
}

```

```

*C:\Program Files\Java\jdk-21\bin\java.exe* "-javaagent:C:\Program Files\JetBra
[LOG] Ruta base del proyecto: C:\java\ACCESO A DATOS\1.2
[LOG] Separador de carpetas del sistema: \
[LOG] Ruta relativa completa: C:\java\ACCESO A DATOS\1.2\datos\ejemplo.txt
[LOG] ¿Existe el archivo? true
[LOG] Ruta absoluta real: C:\java\ACCESO A DATOS\1.2\datos\ejemplo.txt
[LOG] Nombre del archivo: ejemplo.txt
[LOG] Carpeta contenedora: C:\java\ACCESO A DATOS\1.2\datos

```

## UD1 - Actividad Práctica Tema 2.1 - Gestión de archivos y directorios con File

Practicar el uso de la clase `File` en Java para realizar operaciones básicas con archivos y directorios:

- Comprobar si un archivo existe.
- Crear un nuevo archivo.
- Crear una carpeta.
- Listar el contenido de un directorio.
- Verificar permisos y propiedades de un archivo.

# Instrucciones

1. Abre IntelliJ IDEA o tu entorno de desarrollo.
2. Crea un nuevo proyecto Java.
3. Dentro del proyecto, crea una carpeta llamada `datos`.
4. Crea una clase Java llamada `GestionFicheros`.
5. Copia y ejecuta el siguiente código.

## Código Java completo

```
import java.io.File;
import java.io.IOException;
public class GestionFicheros {
    public static void main(String[] args) {
        try {
            // 1. Crear la ruta al archivo datos/fichero.txt
            File archivo = new File("datos/fichero.txt");
            // 2. Si el archivo no existe, créalo
            if (!archivo.exists()) {
                archivo.getParentFile().mkdirs(); // Asegurar carpeta
                archivo.createNewFile();
                System.out.println("Archivo creado correctamente.");
            } else {
                System.out.println("El archivo ya existe.");
            }
            // 3. Mostrar información del archivo
            System.out.println("Nombre: " + archivo.getName());
            System.out.println("Ruta absoluta: " + archivo.getAbsolutePath());
            System.out.println("¿Se puede leer? " +
                archivo.canRead());
            System.out.println("¿Se puede escribir? " +
                archivo.canWrite());
            System.out.println("¿Es un archivo? " +
                archivo.isFile());
            // 4. Crear un nuevo directorio llamado datos/pruebas
            carpeta = new File("datos/pruebas");
            if (!carpeta.exists()) {
                carpeta.mkdir();
                System.out.println("Carpeta creada.");
            } else {
                System.out.println("La carpeta ya existe.");
            }
            // 5. Listar contenido de la carpeta datos
            File carpetaDatos = new File("datos");
            File[] lista = carpetaDatos.listFiles();
            System.out.println("Contenido de la carpeta 'datos':");
            if (lista != null) {
                for (File f : lista) {
                    System.out.println("- " + f.getName() + " (" + (f.isDirectory() ? "directorío" : " (archivo)"));
                }
            }
        }
    }
}
```

```
    }
} catch (IOException e) {
    System.out.println("Error de entrada/salida: " +
e.getMessage());
}
}
}
```

# Resultado esperado

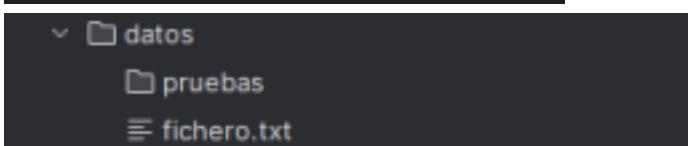
El programa creará un archivo `fichero.txt` dentro de la carpeta `datos` si no existe.

Mostrará la información del archivo.

Creará un subdirectorio llamado `pruebas` .

Listará el contenido completo de la carpeta datos .

```
Carpeta creada.  
Contenido de la carpeta 'datos':  
- fichero.txt (archivo)  
- pruebas (directorio)
```



## Código con preguntas

```
public class GestiónFicheros {
    public static void main(String[] args) {
        try {
            // 1. Crear la ruta al archivo datos/fichero.txt    File archivo =
new File("datos/fichero.txt");

            // 2. Si el archivo no existe, créalo    if
(!archivo.exists()) {
                archivo.getParentFile().mkdirs(); // Asegurar carpeta
                archivo.createNewFile();
                System.out.println("Archivo creado correctamente.");
            } else {
                System.out.println("El archivo ya existe.");
            }
        // 3. Mostrar información del archivo    System.out.println("Nombre: " + archivo.getName());
        }
    }
}
```

```

" + archivo.getName()); System.out.println("Ruta absoluta: " +
archivo.getAbsolutePath());
System.out.println("¿Se puede leer? " +
archivo.canRead());
System.out.println("¿Se puede escribir? " +
archivo.canWrite());
System.out.println("¿Es un archivo? " +
archivo.isFile());

// 4. Crear un nuevo directorio llamado datos/pruebas File carpeta =
new File("datos/pruebas"); if (!carpeta.exists()) {
    carpeta.mkdir();
    System.out.println("Carpeta creada."); } else {
    System.out.println("La carpeta ya existe."); }

// 5. Listar contenido de la carpeta datos File
carpetaDatos = new File("datos"); File[] lista =
carpetaDatos.listFiles(); System.out.println("Contenido de
la carpeta
'datos':");
if (lista != null) {
    for (File f : lista) {
        System.out.println("- " + f.getName() + (f.isDirectory() ? "
(directorio) : " (archivo))); }
}
}

} catch (IOException e) {
    System.out.println("Error de entrada/salida: " +
e.getMessage());
}
}
}

/*
Preguntas de reflexión:

1. ¿Qué ocurre si borras datos/fichero.txt y vuelves a ejecutar el
programa?
- lo volverá a crear
2. ¿Y si cambias los permisos del archivo para que no se pueda
escribir?
- archivo.canWrite() devolverá false. Intentar escribir en el

```

archivo generará una excepción o fallo dependiendo de la operación.

3. ¿Por qué es importante comprobar si un archivo existe antes de crearlo?

- Para evitar sobrescribir datos existentes y prevenir errores al intentar crear un archivo con el mismo nombre.

4. ¿Qué sucede si intentas crear un archivo en una ruta donde no existe la carpeta contenedora?

- El programa lanzará una excepción IOException. Por eso es importante usar

archivo.getParentFile().mkdirs() para asegurarse de que la carpeta exista antes

de crear el archivo.

\*/

## UD1 - Actividad Práctica Tema 3.1 - Lectura y Escritura

Practicar la lectura y escritura de ficheros de texto utilizando `FileWriter` , `BufferedWriter` , `FileReader` y `BufferedReader` .

Escribir líneas en un archivo de texto.

Leer el contenido de ese archivo línea por línea.

Visualizar la salida por consola.

1. Crea una carpeta llamada `datos` en la raíz del proyecto.

2. Crea una clase Java llamada `GestorFicheroTexto` .

3. Implementa los siguientes pasos en el método `main` :

Escribe 3 líneas de texto en un archivo llamado `datos/registro.txt` . Lee el contenido de ese archivo y muéstralolo por consola.

Usa `BufferedWriter` y `BufferedReader` .

```
import java.io.*;

public class GestorFicheroTexto {

    public static void main(String[] args) {
```

```

try {

    // Escritura

    FileWriter fw = new
    FileWriter("datos/registro.txt");

    BufferedWriter bw = new
    BufferedWriter(fw);    bw.write("Registro

1");

    bw.newLine();

    bw.write("Registro 2");
    bw.newLine();

    bw.write("Registro 3");

    bw.newLine();

    bw.flush();

    bw.close();

    System.out.println("Archivo escrito con

éxito."); // Lectura

    FileReader fr = new
    FileReader("datos/registro.txt");

    BufferedReader br = new BufferedReader(fr);

    String linea;

    System.out.println("Contenido del

archivo:");  while ((linea = br.readLine()) != null) {  System.out.println("> " + linea);
}
}

```

```

    }

    br.close();

} catch (IOException e) {

    System.out.println("Error: " +
e.getMessage());
}

}
/*

```

Preguntas de reflexión con respuestas:

1. ¿Qué ocurre si se vuelve a ejecutar el programa sin cambiar el nombre del archivo?

- Si vuelvo a ejecutar el programa, el archivo existente se sobrescribe y se pierden las líneas anteriores. Solo quedan "Registro 1", "Registro 2" y "Registro 3".

2. ¿Cómo podrías añadir texto sin borrar el contenido anterior?

- Para añadir texto sin borrar lo que ya existe, puedo abrir el FileWriter en modo append:

```
FileWriter fw = new FileWriter("datos/registro.txt", true);
```

Esto agrega nuevas líneas al final del archivo sin borrar lo que ya estaba.

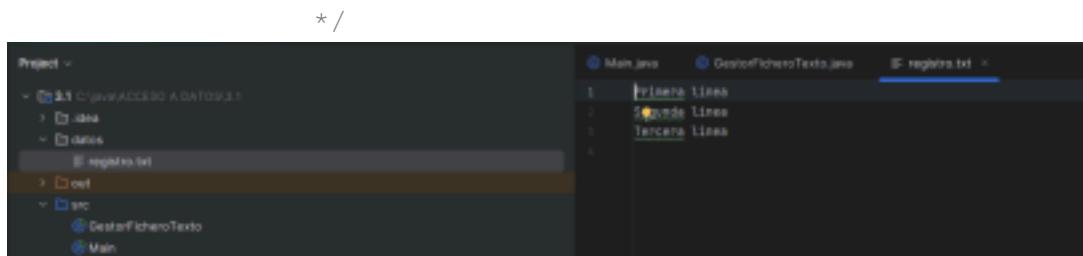
3. ¿Qué diferencias observas si eliminás el BufferedWriter y usas solo FileWriter?

- Si uso solo FileWriter, el programa sigue funcionando, pero es menos eficiente porque cada

escritura va directo al disco. BufferedWriter guarda los datos en un buffer y los escribe de golpe, además permite usar newLine() fácilmente.

4. ¿Por qué es importante cerrar los buffers después de usarlos?

- Es importante cerrar los buffers para liberar los recursos del sistema y asegurarse de que todo lo que estaba en memoria se escriba realmente en el archivo. Si no se cierra, podríamos perder información.



## UD1 - Actividad Práctica Tema 4.1 - Acceso Aleatorio a Ficheros

Utilizar la clase `RandomAccessFile` para escribir y leer registros de texto desde posiciones específicas de un archivo. Aprender a manipular el puntero del archivo mediante `seek()` y a controlar la lectura secuencial o puntual de datos.

1. Crea una carpeta llamada `datos` en tu proyecto.

2. Crea una clase Java llamada `AccesoAleatorioEjercicio` . 3.

Implementa un programa que:

Escriba tres registros con texto en un archivo binario usando `writeUTF` . Lea y muestre el primer y segundo registro usando `seek` y `readUTF` . Mida y muestre la posición del puntero antes y después de cada operación.

```

import java.io.*;

public class AccesoAleatorioEjercicio {

    public static void main(String[] args) {

        try {

            // 1. Crear archivo binario y escribir tres registros

            RandomAccessFile raf = new
            RandomAccessFile("datos/registros.dat", "rw");

            raf.writeUTF("Registro 1");
            raf.writeUTF("Registro 2");
            raf.writeUTF("Registro 3");
            // 2. Volver al inicio para leer

            raf.seek(0);

            // Leer primer registro

            System.out.println("Posición antes de leer 1:
" + raf.getFilePointer());

            String r1 = raf.readUTF();

            System.out.println("Registro 1: " + r1);

            System.out.println("Posición después de leer
1: " + raf.getFilePointer());

            // Leer segundo registro

            System.out.println("Posición antes de leer 2:

```

```

    " + raf.getFilePointer());

    String r2 = raf.readUTF();

    System.out.println("Registro 2: " + r2);

    System.out.println("Posición después de leer
2: " + raf.getFilePointer());

raf.close();

} catch (IOException e) {
    System.out.println("Error: " +
e.getMessage());
}

}

/*

```

Preguntas de reflexión con respuestas:

1. ¿Qué indica el valor que devuelve getFilePointer()?

- Indica la posición actual del puntero dentro del archivo, es decir, el byte donde se encuentra listo para leer o escribir.

2. ¿Qué sucede si cambias el orden de lectura?

- Puedo leer los registros en cualquier orden usando seek(). Si leo en orden diferente, solo necesito mover el puntero a la posición

correcta, de lo contrario leería datos equivocados.

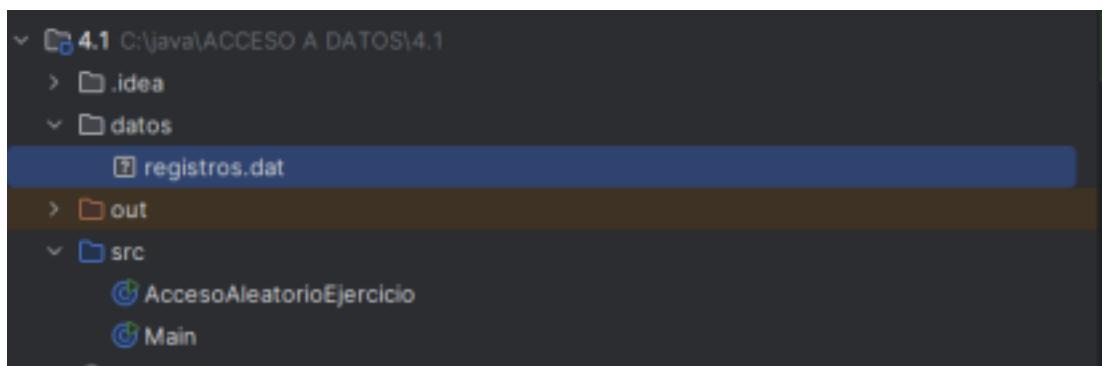
3. ¿Por qué RandomAccessFile no es recomendable para archivos de texto plano sin estructura?

- Porque RandomAccessFile funciona mejor con archivos con registros de tamaño conocido o estructurado. En texto plano, los registros pueden tener longitudes variables, lo que hace difícil calcular posiciones y manipular datos de forma segura.

4. ¿Cómo podrías modificar solo el tercer registro sin afectar los demás?

- Primero uso seek() para mover el puntero al inicio del tercer registro, y luego uso writeUTF() para sobrescribirlo. Así, los otros registros permanecen intactos.

```
* /  
'C:\Program Files\Java\jdk-21\bin\java.exe' *-  
Posición antes de leer 1: 0  
Registro 1: Registro 1  
Posición después de leer 1: 12  
Posición antes de leer 2: 12  
Registro 2: Registro 2  
Posición después de leer 2: 24  
  
Process finished with exit code 0
```



## UD1 - 5.1 Lectura de fichero con Scanner

**El objetivo de esta práctica es la lectura de un archivo de texto en formato CSV (*comma-separated values*).** El formato CSV es una representación de datos donde los registros van separados por un retorno de carro y los campos por un carácter de punto y coma.

Para realizar este supuesto práctico **créate primero un fichero de texto llamado productos.txt** copiando y pegando este contenido:

**El archivo representa los datos de un almacén** donde cada fila es un artículo que consta de categoría, nombre, precio y unidades separados por punto y coma.

**Deberás leer secuencialmente cada uno de los artículos** mostrando en la consola una línea similar a ésta por cada uno de ellos:

**Té Dharamsala (Bebidas) -- Precio: 18,00 € -- Stock: 39**

**Primero comprueba si existe el fichero y muestra en una línea el número de bytes de almacenamiento que ocupa.**

**Al final del anterior listado, debes mostrar en pantalla los siguientes datos:**

- Número total de artículos
- Promedio de precios
- Importe total calculado como suma de precio\*stock

```
LeerProductos.java  ≡ productos.txt ×

1 Bebidas;Té Dharamsala;18.00;39
2 Bebidas;Cerveza Sasquatch;14.00;111
3 Bebidas;Café de Malasia;46.00;17
4 Bebidas;Cerveza Outback;15.00;15
5 Condimentos;Sirope de regaliz;10.00;13
6 Condimentos;Azúcar negra Malacca;19.45;27
7 Condimentos;Sandwich de vegetales;43.90;24
8 Repostería;Pastas de té de chocolate;9.20;25
9 Repostería;Mermelada de Sir Rodney's;81.00;40
10 Repostería;Chocolate Schoggi;43.90;49
11 Repostería;Chocolate blanco;16.25;65
12 Repostería;Tarta de azúcar;49.30;17
13 Lácteos;Queso Cabrales;21.00;22
14 Lácteos;Queso Manchego La Pastora;38.00;86
15 Lácteos;Queso gorgonzola Telino;12.50;0
16 Lácteos;Queso Gudbrandsdals;36.00;26
17 Lácteos;Queso Mozzarella Giovanni;34.80;14
18 Granos/Cereales;Pan fino;9.00;61
19 Granos/Cereales;Cereales para Filo;7.00;38
20 Granos/Cereales;Raviolis Angelo;19.50;36
21 Carnes;Buey Mishí Kobe;97.00;29
22 Carnes;Cordero Alice Springs;39.00;0
23 Carnes;Empanada de cerdo;7.45;21
24 Carnes;Paté chino;24.00;115
25 Frutas/Verduras;Cuajada de judías;23.25;35
26 Frutas/Verduras;Queso de soja Longlife;10.00;4
27 Pescado/Marisco;Pez espada;31.93;31
28 Pescado/Marisco;Algas Konbu;6.00;24
29 Pescado/Marisco;Arenque ahumado;9.50;5
30 Pescado/Marisco;Arenque salado;12.00;95
31 Pescado/Marisco;Caviar rojo;15.00;101
32
```

LeerProductos.java:

```
import java.io.*;
import java.util.*;

public class LeerProductos {
```

```
public static void main(String[] args) {  
    String archivo = "productos.txt";  
    File file = new File(archivo);  
  
    // Comprobamos si el archivo existe y su tamaño tambien  
    if (!file.exists()) {  
        System.out.println("El archivo '" + archivo + "' no  
        existe.");  
        return;  
    } else {  
        System.out.println("El archivo '" + archivo + "' existe  
        y ocupa " + file.length() + " bytes.\n");  
    }  
  
    int totalArticulos = 0;  
    double sumaPreciosParaPromedio = 0.0;  
    double importeTotal = 0.0;  
  
    // Leer el archivo con Scanner  
    try (Scanner sc = new Scanner(file)) {  
        while (sc.hasNextLine()) {  
            String linea = sc.nextLine();  
            String[] partes = linea.split(";");  
            if (partes.length == 4) {  
                String categoria = partes[0];  
                String nombre = partes[1];  
                double precio = Double.parseDouble(partes[2]);  
                int stock = Integer.parseInt(partes[3]);  
            }  
        }  
    }  
}
```

```

        System.out.printf("%s (%s) -- Precio: %.2f €
-- Stock: %d%n", nombre, categoria, precio, stock);

        totalArticulos++;

        sumaPreciosParaPromedio += precio;

        importeTotal += precio * stock;

    }

}

} catch (FileNotFoundException e) {

    System.out.println("No se pudo abrir el archivo.");
}

double promedioPrecios = totalArticulos > 0 ?
sumaPreciosParaPromedio / totalArticulos : 0;

System.out.println("\nResumen:");

System.out.println("Número total de artículos: " +
totalArticulos);

System.out.printf("Promedio de precios: %.2f €%n",
promedioPrecios);

System.out.printf("Importe total del almacén: %.2f €%n",
importeTotal);

}

```

Té Dharamsala (Bebidas) -- Precio: 18,00 € -- Stock: 39  
Cerveza Sasquatch (Bebidas) -- Precio: 14,00 € -- Stock: 111  
Café de Malasia (Bebidas) -- Precio: 46,00 € -- Stock: 17  
Cerveza Outback (Bebidas) -- Precio: 15,00 € -- Stock: 15  
Sirope de regaliz (Condimentos) -- Precio: 10,00 € -- Stock: 13  
Azúcar negra Malacca (Condimentos) -- Precio: 19,45 € -- Stock: 27  
Sandwich de vegetales (Condimentos) -- Precio: 43,90 € -- Stock: 24  
Pastas de té de chocolate (Repostería) -- Precio: 9,20 € -- Stock: 25  
Mermelada de Sir Rodney's (Repostería) -- Precio: 81,00 € -- Stock: 40  
Chocolate Schoggi (Repostería) -- Precio: 43,90 € -- Stock: 49  
Chocolate blanco (Repostería) -- Precio: 16,25 € -- Stock: 65  
Tarta de azúcar (Repostería) -- Precio: 49,30 € -- Stock: 17  
Queso Cabrales (Lácteos) -- Precio: 21,00 € -- Stock: 22  
Queso Manchego La Pastora (Lácteos) -- Precio: 38,00 € -- Stock: 86  
Queso gorgonzola Telino (Lácteos) -- Precio: 12,50 € -- Stock: 0  
Queso Gudbrandsdals (Lácteos) -- Precio: 36,00 € -- Stock: 26  
Queso Mozzarella Giovanni (Lácteos) -- Precio: 34,80 € -- Stock: 14  
Pan fino (Granos/Cereales) -- Precio: 9,00 € -- Stock: 61  
Cereales para Filo (Granos/Cereales) -- Precio: 7,00 € -- Stock: 38  
Raviolis Angelo (Granos/Cereales) -- Precio: 19,50 € -- Stock: 36  
Buey Mishi Kobe (Carnes) -- Precio: 97,00 € -- Stock: 29  
Cordero Alice Springs (Carnes) -- Precio: 39,00 € -- Stock: 0  
Empanada de cerdo (Carnes) -- Precio: 7,45 € -- Stock: 21  
Paté chino (Carnes) -- Precio: 24,00 € -- Stock: 115  
Cuajada de judías (Frutas/Verduras) -- Precio: 23,25 € -- Stock: 35  
Queso de soja Longlife (Frutas/Verduras) -- Precio: 10,00 € -- Stock: 4  
Pez espada (Pescado/Marisco) -- Precio: 31,93 € -- Stock: 31  
Algas Konbu (Pescado/Marisco) -- Precio: 6,00 € -- Stock: 24  
Arenque ahumado (Pescado/Marisco) -- Precio: 9,50 € -- Stock: 5  
Arenque salado (Pescado/Marisco) -- Precio: 12,00 € -- Stock: 95  
Caviar rojo (Pescado/Marisco) -- Precio: 15,00 € -- Stock: 101

Resumen:

Número total de artículos: 31

Promedio de precios: 26,42 €

Importe total del almacén: 29576,93 €