PRÁCTICA DE ACCESO A DATOS BINARIOS Y BASES DE DATOS EN JAVA

FUNDAMENTOS DEL SISTEMA I/O BINARIO EN JAVA

Jerarquía de clases

Java I/O se organiza en dos jerarquías principales:

- **Stream-based**: para datos binarios (*InputStream/OutputStream*)
- **Reader/Writer-based**: para datos de texto (*Reader/Writer*)

Las clases que estudiaremos pertenecen al primer grupo, diseñadas específicamente para manejar bytes y datos binarios de forma eficiente.

Conceptos clave:

- **Byte**: unidad básica de datos binarios (*8 bits*)
- **Buffer**: memoria intermedia que reduce accesos costosos al disco
- **Primitivos**: tipos básicos de Java (*int, double, boolean, etc.)*
- **Stream**: flujo de datos secuencial
- **Conexión JDBC**: sesión activa con una base de datos

FILEINPUTSTREAM - LECTURA BÁSICA DE BYTES

Descripción técnica

FileInputStream es una clase fundamental que extiende InputStream, específicamente diseñada para leer archivos en formato binario. Lee datos byte por byte directamente desde el sistema de archivos.

Arquitectura interna:

FileInputStream → InputStream → Sistema Operativo

Constructores detallados:

```
// Básico - recibe nombre de archivo
FileInputStream(String name) throws FileNotFoundException
// Con objeto File
FileInputStream(File file) throws FileNotFoundException
// Con descriptor de archivo (avanzado)
FileInputStream(FileDescriptor fdObj)
```

Métodos principales con detalles:

1. **int read()**
   - Retorna: valor del byte (*0-255*) o -1 al EOF
   - Bloquea hasta que hay un byte disponible
   - Ejemplo de uso:

```
int b;
while ((b = fis.read()) != -1) {
    System.out.print(b + " ");
}
```

2. **int read(*byte[] b*)**
   - Llena el array completamente si es posible
   - Retorna: número real de bytes leídos o -1
   - Más eficiente que read() individual

```
byte[] buffer = new byte[1024];
int bytesRead = fis.read(buffer);
```

3. **int read(*byte[] b, int off, int len*)**
   - Lee hasta len bytes en b[*off...off+len-1*]
   - Permite reutilizar arrays grandes leyendo en segmentos

```
byte[] bigBuffer = new byte[4096];
int read = fis.read(bigBuffer, 1000, 500); // Lee 500 bytes en posición 1000
```

4. int available()

- Retorna estimación de bytes disponibles para lectura sin bloqueo
- Útil para pre-dimensionar buffers

5. long skip(long n)

- Salta n bytes del stream
- Retorna número real de bytes saltados

Limitaciones importantes:

- **Sin buffer**: cada read() puede generar una llamada al sistema operativo
- **Solo bytes**: no interpreta tipos primitivos automáticamente
- **No thread-safe**: no debe compartirse entre hilos sin sincronización

Cuándo usar FileInputStream:

- Lectura de archivos binarios (*imágenes, audio, PDFs*)
- Procesamiento de datos no textuales
- Como base para DataInputStream
- Evitar para: archivos de texto (*usar FileReader en su lugar*)

FILEOUTPUTSTREAM - ESCRITURA BÁSICA DE BYTES

Descripción técnica

FileOutputStream extiende OutputStream y proporciona una interfaz para escribir bytes en archivos. Es el complemento de FileInputStream para operaciones de escritura binaria.

Arquitectura interna:

FileOutputStream → OutputStream → Sistema Operativo

Constructores con opciones:

// Sobrescribe archivo existente

FileOutputStream(String name) throws FileNotFoundException

// Control de modo append

FileOutputStream(String name, boolean append) throws IOException

// Con objeto File

FileOutputStream(File file) throws FileNotFoundException

FileOutputStream(File file, boolean append) throws IOException

// Con descriptor de archivo

FileOutputStream(FileDescriptor fdObj)

Métodos de escritura detallados:

1. **void write(*int b*)**
    - Escribe un único byte (*toma los 8 bits bajos del int*)

fos.write(65); // Escribe byte con valor 65

fos.write(0xFF); // Escribe byte con valor 255

2. **void write(*byte[] b*)**
    - Escribe todo el array de bytes

byte[] datos = {72, 111, 108, 97}; // "Hola" en ASCII

fos.write(datos);

3. **void write(*byte[] b, int off, int len*)**
    - Escribe segmento específico del array

byte[] buffer = new byte[100];

fos.write(buffer, 10, 50); // Escribe 50 bytes desde posición 10

Métodos de gestión:

- **void flush()**: fuerza escritura inmediata del buffer interno al disco
- **void close()**: cierra el stream (*hace flush automáticamente*)

Gestión de archivos:

- **Modo sobrescritura (*default*)**: elimina contenido previo
- **Modo append**: añade al final del archivo existente
- **Creación automática**: si el archivo no existe, se crea

Consideraciones de rendimiento:

- Sin buffer explícito, pero el SO puede tener su propio buffer
- Para múltiples escrituras, mejor usar BufferedOutputStream
- flush() es costoso, úsalo solo cuando necesites garantizar escritura inmediata

---

DATAINPUTSTREAM - LECTURA DE TIPOS PRIMITIVOS

Descripción avanzada

DataInputStream implementa el patrón Decorator, envolviendo cualquier InputStream y añadiendo capacidades para leer tipos primitivos de Java directamente desde un stream binario.

Arquitectura con interpretación de tipos:

DataInputStream → FileInputStream → Sistema Operativo

Ventajas de DataInputStream:

- **Lectura tipada**: lee int, double, boolean, etc. directamente
- **Formato portable**: usa big-endian (*orden de bytes estándar*)
- **Complemento perfecto**: funciona con DataOutputStream
- **Strings UTF**: soporte para cadenas con writeUTF/readUTF

Constructor:

```
// Envuelve cualquier InputStream
DataInputStream(InputStream in)
```

Ejemplo de creación:

```
DataInputStream dis = new DataInputStream(
   new FileInputStream("datos.bin")
);
```

Métodos de lectura tipada:

1. **boolean readBoolean()**
   - Lee 1 byte y retorna true si es distinto de cero
2. **byte readByte()**
   - Lee 1 byte con signo (*-128 a 127*)
3. **int readUnsignedByte()**
   - Lee 1 byte sin signo (*0 a 255*)
4. **short readShort()**
   - Lee 2 bytes con signo
5. **int readUnsignedShort()**
   - Lee 2 bytes sin signo
6. **int readInt()**
   - Lee 4 bytes y retorna un int

```
int edad = dis.readInt();
```

7. **long readLong()**
   - Lee 8 bytes y retorna un long
8. **float readFloat()**
   - Lee 4 bytes en formato IEEE 754

9. double readDouble()

- Lee 8 bytes en formato IEEE 754

```
double salario = dis.readDouble();
```

10. String readUTF()

- Lee string con prefijo de longitud en formato UTF-8 modificado
- Primero lee 2 bytes con la longitud, luego los caracteres

```
String nombre = dis.readUTF();
```

11. void readFully(*byte[] b*)

- Lee exactamente b.length bytes, bloqueando si es necesario
- Lanza EOFException si llega al final antes de completar

Orden de lectura importante:

¡CRÍTICO! Debes leer los datos en el mismo orden en que fueron escritos:

```
// Escritura
dos.writeInt(25);
dos.writeDouble(1500.50);
dos.writeUTF("Pedro");
// Lectura (mismo orden)
int edad = dis.readInt();
double salario = dis.readDouble();
String nombre = dis.readUTF();
```

Manejo de fin de archivo:

```
try {
   while (true) {
      int numero = dis.readInt();
      System.out.println(numero);
   }
} catch (EOFException e) {
```

```
    // Fin del archivo alcanzado
    System.out.println("Lectura completada");
}
```

Casos de uso específicos:

- Lectura de archivos de datos estructurados
- Deserialización simple de datos
- Lectura de formatos binarios propietarios
- Comunicación entre aplicaciones Java

DATAOUTPUTSTREAM - ESCRITURA DE TIPOS PRIMITIVOS

Descripción avanzada

DataOutputStream envuelve cualquier OutputStream añadiendo capacidad para escribir tipos primitivos de Java en formato binario portable. Es el complemento de DataInputStream.

Arquitectura:

DataOutputStream → FileOutputStream → Sistema Operativo

Proceso interno:

1. Método writeXXX() convierte el tipo primitivo a bytes
2. Bytes se escriben en el OutputStream subyacente
3. Formato big-endian garantiza portabilidad entre plataformas

Constructor:

```
// Envuelve cualquier OutputStream
DataOutputStream(OutputStream out)
```

Ejemplo de creación:

```
DataOutputStream dos = new DataOutputStream(
   new FileOutputStream("datos.bin")
);
```

Métodos de escritura tipada:

1. **void writeBoolean(*boolean v*)**
   - Escribe 1 byte: 1 para true, 0 para false
2. **void writeByte(*int v*)**
   - Escribe 1 byte (*byte bajo del int*)
3. **void writeShort(int v)**
   - Escribe 2 bytes (*short como int*)
4. **void writeInt(*int v*)**
   - Escribe 4 bytes

```
dos.writeInt(12345);
```

5. **void writeLong(*long v*)**
   - Escribe 8 bytes

```
dos.writeLong(123456789L);
```

6. **void writeFloat(*float v*)**
   - Escribe 4 bytes en formato IEEE 754
7. **void writeDouble(*double v*)**
   - Escribe 8 bytes en formato IEEE 754

```
dos.writeDouble(3.14159);
```

8. **void writeUTF(*String s*)**
   - Escribe string con prefijo de longitud
   - Formato: 2 bytes (*longitud*) + bytes UTF-8

```
dos.writeUTF("Hola Mundo");
```

9. void writeChars(*String s*)

- Escribe cada char como 2 bytes
- No incluye información de longitud

10. void writeBytes(*String s*)

- Escribe byte bajo de cada char
- Pierde caracteres no-ASCII

Métodos de control:

1. void flush()

- Vacía buffer al OutputStream subyacente

```
dos.flush(); // Garantiza escritura inmediata
```

2. int size()

- Retorna número de bytes escritos hasta ahora

Patrón de uso eficiente:

```
// EFICIENTE - escribir múltiples valores y un solo flush
dos.writeInt(100);
dos.writeDouble(99.99);
dos.writeUTF("Producto");
dos.flush(); // Una sola vez al final
// INEFICIENTE - flush después de cada escritura
dos.writeInt(100);
dos.flush();
dos.writeDouble(99.99);
dos.flush();
```

Diseño de registro binario estructurado:

```
// Estructura de registro: int id + String nombre + double precio
public void escribirProducto(int id, String nombre, double precio)
    throws IOException {
    dos.writeInt(id);
    dos.writeUTF(nombre);
    dos.writeDouble(precio);
}
// Lectura correspondiente
public Producto leerProducto() throws IOException {
    int id = dis.readInt();
    String nombre = dis.readUTF();
    double precio = dis.readDouble();
    return new Producto(id, nombre, precio);
}
```

Cálculo de tamaño de registro:

```
// int: 4 bytes
// String UTF: 2 bytes (longitud) + n bytes (contenido variable)
// double: 8 bytes
// Total: variable según longitud del String
```

Casos de uso:

- Persistencia simple de objetos
- Archivos de configuración binarios
- Protocolos de red personalizados
- Almacenamiento eficiente de datos numéricos

JDBC - CONECTIVIDAD CON BASES DE DATOS

Descripción completa

JDBC (*Java Database Connectivity*) es una API estándar de Java que permite conectar aplicaciones Java con bases de datos relacionales. Proporciona un conjunto de interfaces y clases para ejecutar consultas SQL y manipular datos.

Características distintivas:

- **Independencia de base de datos**: mismo código para MySQL, PostgreSQL, Oracle, etc.
- **Ejecución de SQL**: permite ejecutar cualquier sentencia SQL
- **Gestión de transacciones**: control de commits y rollbacks
- **Manejo de resultados**: iteración sobre conjuntos de datos

Arquitectura de JDBC:

Aplicación Java

↓

API JDBC (*java.sql*)

↓

Driver JDBC (*específico de BD*)

↓

Base de Datos

Componentes principales:

1. **DriverManager**
    - Gestiona los drivers disponibles
    - Establece conexiones con bases de datos
2. **Connection**
    - Representa una sesión con la base de datos
    - Permite crear Statements y gestionar transacciones

3. Statement / PreparedStatement

- Ejecuta sentencias SQL
- PreparedStatement permite parámetros y mejor rendimiento

4. ResultSet

- Contiene los resultados de una consulta
- Permite iterar sobre las filas devueltas

Paso 1: Cargar el driver (opcional desde JDBC 4.0)

```
// Ya no es necesario en versiones modernas, se carga automáticamente
// Class.forName("com.mysql.cj.jdbc.Driver");
```

Paso 2: Establecer conexión

```
// URL de conexión: jdbc:tipo_bd://host:puerto/nombre_bd
String url = "jdbc:mysql://localhost:3306/mi_base_datos";
String usuario = "root";
String password = "mipassword";
Connection conn = DriverManager.getConnection(url, usuario, password);
```

Formatos de URL según base de datos:

- MySQL: jdbc:mysql://localhost:3306/basedatos
- PostgreSQL: jdbc:postgresql://localhost:5432/basedatos
- H2 (*en memoria*): jdbc:h2:mem:testdb
- SQLite: jdbc:sqlite:path/to/database.db

Paso 3: Crear Statement

```
Statement stmt = conn.createStatement();
```

Paso 4: Ejecutar consultas

Consulta SELECT (*executeQuery*):

```
String sql = "SELECT id, nombre, edad FROM usuarios";
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) {
   int id = rs.getInt("id");
   String nombre = rs.getString("nombre");
   int edad = rs.getInt("edad");
   System.out.println(id + ": " + nombre + " - " + edad);
}
rs.close();
```

Operaciones INSERT/UPDATE/DELETE (*executeUpdate*):

```
String sql = "INSERT INTO usuarios (nombre, edad) VALUES ('Juan', 25)";
int filasAfectadas = stmt.executeUpdate(sql);
System.out.println("Filas insertadas: " + filasAfectadas);
```

Paso 5: Cerrar recursos

```
// Cerrar en orden inverso a la apertura
rs.close();     // ResultSet
stmt.close();   // Statement
conn.close();   // Connection
```

PreparedStatement - Sentencias preparadas

PreparedStatement es más seguro y eficiente que Statement. Permite parámetros y previene inyección SQL.

Ventajas:

- Previene inyección SQL
- Mejor rendimiento al reutilizar
- Código más limpio y legible

Creación y uso:

```
String sql = "INSERT INTO usuarios (nombre, edad, email) VALUES (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(sql);
// Asignar valores a los parámetros (índice empieza en 1)
pstmt.setString(1, "María");
pstmt.setInt(2, 30);
pstmt.setString(3, "maria@email.com");
int filas = pstmt.executeUpdate();
```

Métodos setXXX disponibles:

- setInt(*int index, int value*)
- setString(*int index, String value*)
- setDouble(*int index, double value*)
- setBoolean(*int index, boolean value*)
- setDate(*int index, java.sql.Date value*)
- setNull(*int index, int sqlType*)

Reutilización de PreparedStatement:

```
PreparedStatement pstmt = conn.prepareStatement(
   "INSERT INTO productos (nombre, precio) VALUES (?, ?)"
);
// Primera inserción
pstmt.setString(1, "Laptop");
```

pstmt.setDouble(2, 999.99);

pstmt.executeUpdate();

// Segunda inserción (mismo statement)

pstmt.setString(1, "Mouse");

pstmt.setDouble(2, 19.99);

pstmt.executeUpdate();

pstmt.close();

Lectura de ResultSet

Métodos de navegación:

- boolean next(): avanza a la siguiente fila, retorna false al final
- boolean previous(): retrocede a la fila anterior (*requiere ResultSet scrollable*)
- void beforeFirst(): posiciona antes de la primera fila
- void afterLast(): posiciona después de la última fila

Métodos de lectura por nombre de columna:

int id = rs.getInt("id");

String nombre = rs.getString("nombre");

double precio = rs.getDouble("precio");

java.sql.Date fecha = rs.getDate("fecha_registro");

Métodos de lectura por índice (*empieza en 1*):

int id = rs.getInt(1);

String nombre = rs.getString(2);

double precio = rs.getDouble(3);

Verificar valores NULL:

```
double salario = rs.getDouble("salario");
if (rs.wasNull()) {
    System.out.println("Salario es NULL");
}
```

Gestión de transacciones

```
try {
    // Deshabilitar auto-commit
    conn.setAutoCommit(false);
    // Ejecutar múltiples operaciones
    stmt.executeUpdate("INSERT INTO cuentas (id, saldo) VALUES (1, 1000)");
    stmt.executeUpdate("UPDATE cuentas SET saldo = saldo - 100 WHERE id = 1");
    stmt.executeUpdate("UPDATE cuentas SET saldo = saldo + 100 WHERE id = 2");
    // Confirmar transacción
    conn.commit();
    System.out.println("Transacción completada");
} catch (SQLException e) {
    // Revertir en caso de error
    conn.rollback();
    System.out.println("Transacción revertida");
} finally {
    // Restaurar auto-commit
    conn.setAutoCommit(true);
}
```

Manejo de excepciones robusto

```
Connection conn = null;
PreparedStatement pstmt = null;
```

```
ResultSet rs = null;
try {
   conn = DriverManager.getConnection(url, user, pass);
   pstmt = conn.prepareStatement("SELECT * FROM usuarios WHERE edad > ?");
   pstmt.setInt(1, 18);
   rs = pstmt.executeQuery();
   while (rs.next()) {
      System.out.println(rs.getString("nombre"));
   }
} catch (SQLException e) {
   System.err.println("Error SQL: " + e.getMessage());
   e.printStackTrace();
} finally {
   // Cerrar recursos en orden inverso
   try { if (rs != null) rs.close(); } catch (SQLException e) {}
   try { if (pstmt != null) pstmt.close(); } catch (SQLException e) {}
   try { if (conn != null) conn.close(); } catch (SQLException e) {}
}
```

Try-with-resources (*Java 7*+)

```
String url = "jdbc:mysql://localhost:3306/mibd";
String sql = "SELECT * FROM usuarios";
try (Connection conn = DriverManager.getConnection(url, "user", "pass");
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql)) {
   while (rs.next()) {
      System.out.println(rs.getString("nombre"));
   }

} catch (SQLException e) {
```

```
    e.printStackTrace();
}
// Cierre automático garantizado
```

Metadata de la base de datos

```
DatabaseMetaData meta = conn.getMetaData();
System.out.println("Driver: " + meta.getDriverName());
System.out.println("Versión: " + meta.getDriverVersion());
System.out.println("URL: " + meta.getURL());
System.out.println("Usuario: " + meta.getUserName());
```

Metadata de ResultSet

```
ResultSetMetaData rsmd = rs.getMetaData();
int numColumnas = rsmd.getColumnCount();
for (int i = 1; i <= numColumnas; i++) {
    System.out.println("Columna " + i + ": " + rsmd.getColumnName(i) +
                       " - Tipo: " + rsmd.getColumnTypeName(i));
}
```

Casos de uso JDBC:

- Aplicaciones de escritorio con base de datos local
- Sistemas de gestión empresarial
- Herramientas de administración de BD
- Aplicaciones que requieren control total sobre SQL

Limitaciones de JDBC puro:

- Código verboso y repetitivo
- Gestión manual de conexiones
- Mapeo manual objeto-relacional

- SQL embebido en código Java
- Solución: usar JPA/Hibernate para proyectos grandes (*fuera del alcance de esta práctica*)

PROPERTIES - ARCHIVOS DE CONFIGURACIÓN

Descripción completa

La clase Properties (*java.util.Properties*) es una estructura especializada de Java que permite almacenar y recuperar pares clave-valor, típicamente para configuraciones de aplicaciones. Extiende Hashtable y está optimizada para trabajar con archivos de texto planos.

Características distintivas:

- **Persistencia**: puede guardar/cargar desde archivos
- **Formato texto**: archivos legibles y editables manualmente
- **Claves y valores String**: todo se maneja como cadenas
- **Jerarquía**: soporte para propiedades por defecto
- **Comentarios**: permite documentar el archivo

Formato de archivo properties:

 # Comentario: configuración de la aplicación

 # Los comentarios empiezan con # o !

 # Propiedades básicas

db.host=localhost

db.port=3306

db.name=mi_base_datos

db.user=admin

db.password=secret123

 # Espacios y caracteres especiales

app.title=Mi Aplicación

```
app.version=1.0.0
app.debug=true
# Paths (usar / o \\)
app.data.path=C:/datos/aplicacion
```

Creación y carga:

```
import java.util.Properties;
import java.io.*;
// Crear objeto Properties
Properties config = new Properties();
// Cargar desde archivo
try (FileInputStream fis = new FileInputStream("config.properties")) {
    config.load(fis);
} catch (IOException e) {
    e.printStackTrace();
}
```

Lectura de propiedades:

```
// Obtener valor (null si no existe)
String host = config.getProperty("db.host");
// Obtener con valor por defecto
String puerto = config.getProperty("db.port", "3306");
int port = Integer.parseInt(puerto);
// Verificar si existe una clave
boolean existe = config.containsKey("db.user");
```

Escritura de propiedades:

```
// Establecer valor
config.setProperty("app.language", "es");
```

```
config.setProperty("app.theme", "dark");

// Eliminar propiedad

config.remove("app.debug");
```

Guardar en archivo:

```
// Guardar con comentario

try (FileOutputStream fos = new FileOutputStream("config.properties")) {

    config.store(fos, "Configuración de la aplicación");

} catch (IOException e) {

    e.printStackTrace();

}
```

Resultado en el archivo:

```
#Configuración de la aplicación

#Sun Oct 26 22:00:00 CET 2025

app.language=es

app.theme=dark

db.host=localhost

db.port=3306
```

Properties con valores por defecto:

```
// Crear properties con valores por defecto

Properties defaultProps = new Properties();

defaultProps.setProperty("db.host", "localhost");

defaultProps.setProperty("db.port", "3306");

defaultProps.setProperty("app.debug", "false");

// Crear properties que usa los defaults

Properties config = new Properties(defaultProps);
```

```
// Si no existe, usa el valor por defecto
String host = config.getProperty("db.host"); // "localhost" (del default)
config.setProperty("db.host", "192.168.1.100"); // Sobrescribe
String nuevoHost = config.getProperty("db.host"); // "192.168.1.100"
```

Listar todas las propiedades:

```
// Iterar sobre todas las claves
for (String key : config.stringPropertyNames()) {
    String value = config.getProperty(key);
    System.out.println(key + " = " + value);
}
// Imprimir directamente (incluye defaults)
config.list(System.out);
```

Formato XML (*alternativo*):

```
// Guardar en formato XML
try (FileOutputStream fos = new FileOutputStream("config.xml")) {
    config.storeToXML(fos, "Configuración XML", "UTF-8");
}
// Cargar desde XML
try (FileInputStream fis = new FileInputStream("config.xml")) {
    config.loadFromXML(fis);
}
```

Archivo XML generado:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Configuración XML</comment>
```

```
<entry key="db.host">localhost</entry>
<entry key="db.port">3306</entry>
</properties>
```

Patrón de uso típico: Clase de Configuración

```
public class Configuracion {
    private static Properties props;
    static {
        props = new Properties();
        try (InputStream is = Configuracion.class
                .getResourceAsStream("/config.properties")) {
            props.load(is);
        } catch (IOException e) {
            // Usar valores por defecto
            props.setProperty("db.host", "localhost");
            props.setProperty("db.port", "3306");
        }
    }
    public static String get(String key) {
        return props.getProperty(key);
    }
    public static String get(String key, String defaultValue) {
        return props.getProperty(key, defaultValue);
    }
    public static int getInt(String key, int defaultValue) {
        String value = props.getProperty(key);
        return value != null ? Integer.parseInt(value) : defaultValue;
    }
    public static boolean getBoolean(String key, boolean defaultValue) {
        String value = props.getProperty(key);
```

```
        return value != null ? Boolean.parseBoolean(value) : defaultValue;
    }
}
// Uso
String host = Configuracion.get("db.host");
int puerto = Configuracion.getInt("db.port", 3306);
boolean debug = Configuracion.getBoolean("app.debug", false);
```

Integración con JDBC:

```
// config.properties
// jdbc.url=jdbc:mysql://localhost:3306/mibd
// jdbc.user=admin
// jdbc.password=secret
Properties config = new Properties();
config.load(new FileInputStream("config.properties"));
String url = config.getProperty("jdbc.url");
String user = config.getProperty("jdbc.user");
String pass = config.getProperty("jdbc.password");
Connection conn = DriverManager.getConnection(url, user, pass);
```

Casos de uso específicos:

- Configuración de conexiones a bases de datos
- Parámetros de aplicación (*idioma, tema, rutas*)
- Configuración de servidores y puertos
- Credenciales (*aunque mejor usar variables de entorno para producción*)

COMPARATIVA Y CRITERIOS DE SELECCIÓN

Tabla de comparación rápida:

Clase	Tipo de datos	Uso principal	Mejor para
FileInputStream	Bytes	Lectura binaria	Imágenes, archivos binarios
FileOutputStream	Bytes	Escritura binaria	Guardar datos binarios
DataInputStream	Primitivos	Lectura tipada	Datos estructurados binarios
DataOutputStream	Primitivos	Escritura tipada	Persistencia de primitivos
JDBC	Relacional	Base de datos	Datos relacionales, consultas SQL
Properties	Texto clave-valor	Configuración	Parámetros de aplicación

Árbol de decisión para elegir tecnología:

1. ¿Necesitas almacenar datos relacionales con consultas complejas?

JDBC con base de datos

2. ¿Necesitas configuración simple editable manualmente?

Properties

3. ¿Necesitas leer/escribir tipos primitivos de Java?

DataInputStream/DataOutputStream

4. ¿Necesitas manejar archivos binarios puros (imágenes, PDFs)?

FileInputStream/FileOutputStream

Comparativa: Persistencia de datos

Escenario: Guardar información de un empleado

Opción 1: DataOutputStream (*binario*)

dos.writeInt(empleado.getId());

dos.writeUTF(empleado.getNombre());

dos.writeDouble(empleado.getSalario());

- Ventaja: Compacto, rápido
- Desventaja: No legible por humanos, formato propietario

Opción 2: JDBC (*base de datos*)

pstmt.setInt(1, empleado.getId());

pstmt.setString(2, empleado.getNombre());

pstmt.setDouble(3, empleado.getSalario());

pstmt.executeUpdate();

- Ventaja: Consultas SQL, integridad referencial, concurrencia
- Desventaja: Requiere servidor de BD, más complejo

Opción 3: Properties (*texto*)

props.setProperty("empleado.id", String.valueOf(empleado.getId()));

props.setProperty("empleado.nombre", empleado.getNombre());

props.setProperty("empleado.salario", String.valueOf(empleado.getSalario()));

- Ventaja: Legible, editable manualmente
- Desventaja: Todo es String, no escalable para múltiples registros

Rendimiento comparativo (*orientativo*):

Operación	**FileStream**	**DataStream**	**JDBC**	**Properties**
Lectura 1MB	Muy rápido	Muy rápido	Rápido	Medio
Escritura 1MB	Muy rápido	Muy rápido	Rápido	Medio
Búsqueda	Secuencial	Secuencial	Indexada	N/A
Escalabilidad	Baja	Baja	Alta	Muy baja

MEJORES PRÁCTICAS Y OPTIMIZACIONES

1. Gestión de recursos con try-with-resources:

```
// ✓ CORRECTO - cierre automático garantizado
try (FileInputStream fis = new FileInputStream("archivo.dat");
    DataInputStream dis = new DataInputStream(fis)) {
   int numero = dis.readInt();
} catch (IOException e) {
   e.printStackTrace();
}
// ✗ INCORRECTO - posible leak de recursos
DataInputStream dis = new DataInputStream(new FileInputStream("archivo.dat"));
int numero = dis.readInt();
dis.close(); // ¿Qué pasa si hay excepción antes?
```

2. Buffer para archivos grandes:

```
// ✓ CORRECTO - usar BufferedInputStream/BufferedOutputStream
try (BufferedInputStream bis = new BufferedInputStream(
      new FileInputStream("grande.dat"));
    DataInputStream dis = new DataInputStream(bis)) {
   // Lectura eficiente con buffer de 8192 bytes
}
// ✗ MENOS EFICIENTE - sin buffer
try (DataInputStream dis = new DataInputStream(
      new FileInputStream("grande.dat"))) {
   // Cada read() puede ser una llamada al SO
}
```

3. Reutilización de PreparedStatement:

```
// ✓ CORRECTO - compilar una vez, ejecutar múltiples veces
PreparedStatement pstmt = conn.prepareStatement(
    "INSERT INTO productos VALUES (?, ?)");
for (Producto p : productos) {
    pstmt.setInt(1, p.getId());
    pstmt.setString(2, p.getNombre());
    pstmt.executeUpdate();
}
pstmt.close();
// ✗ INEFICIENTE - crear Statement en cada iteración
for (Producto p : productos) {
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO productos VALUES (" +
        p.getId() + ", '" + p.getNombre() + "')");
    stmt.close();
}
```

4. Batch processing para múltiples inserciones:

```
PreparedStatement pstmt = conn.prepareStatement(
    "INSERT INTO logs (fecha, mensaje) VALUES (?, ?)");
for (LogEntry log : logs) {
    pstmt.setDate(1, log.getFecha());
    pstmt.setString(2, log.getMensaje());
    pstmt.addBatch();
}
int[] resultados = pstmt.executeBatch();
conn.commit();
```

5. Manejo correcto de transacciones:

```
Connection conn = null;
try {
    conn = DriverManager.getConnection(url, user, pass);
    conn.setAutoCommit(false);
    // Múltiples operaciones
    stmt1.executeUpdate(...);
    stmt2.executeUpdate(...);
    conn.commit();
} catch (SQLException e) {
    if (conn != null) {
        try {
            conn.rollback();
        } catch (SQLException ex) {}
    }
    e.printStackTrace();
} finally {
    if (conn != null) {
        try {
            conn.setAutoCommit(true);
            conn.close();
        } catch (SQLException e) {}
    }
}
```

6. Cargar Properties desde classpath:

```
// ✓ CORRECTO - funciona en JAR también
Properties props = new Properties();
try (InputStream is = MiClase.class
    .getResourceAsStream("/config.properties")) {
  props.load(is);
}
// ✗ PROBLEMÁTICO - ruta absoluta, no funciona en JAR
Properties props = new Properties();
try (FileInputStream fis = new FileInputStream(
    "C:\\proyecto\\config.properties")) {
  props.load(fis);
}
```

7. Validación y conversión de Properties:

```
public class Config {
  private Properties props;
  public int getIntProperty(String key, int defaultValue) {
    String value = props.getProperty(key);
    if (value == null) return defaultValue;
    try {
      return Integer.parseInt(value.trim());
    } catch (NumberFormatException e) {
      System.err.println("Valor inválido para " + key + ": " + value);
      return defaultValue;
    }
  }
  public double getDoubleProperty(String key, double defaultValue) {
    String value = props.getProperty(key);
```

```
    if (value == null) return defaultValue;
    try {
        return Double.parseDouble(value.trim());
    } catch (NumberFormatException e) {
        return defaultValue;
    }
  }
}
```

8. Pool de conexiones simple (*conceptual*):

```
// En aplicaciones reales usar HikariCP, pero conceptualmente:
public class ConexionDB {
  private static Connection conn = null;
  public static Connection getConexion() throws SQLException {
    if (conn == null || conn.isClosed()) {
        String url = "jdbc:mysql://localhost:3306/mibd";
        conn = DriverManager.getConnection(url, "user", "pass");
    }
    return conn;
  }
  public static void cerrar() {
    if (conn != null) {
        try { conn.close(); } catch (SQLException e) {}
    }
  }
}
```

EJEMPLOS DE INTRODUCCIÓN

Ejemplo 1: FileInputStream – Lectura básica de bytes

Se abre un flujo de bytes sobre el archivo "datos.bin" y, byte a byte, se lee hasta alcanzar el fin de archivo (*read() devuelve –1*). Cada byte leído se muestra por consola como un valor numérico. El bloque try-with-resources garantiza el cierre automático del FileInputStream y captura posibles errores de E/S. Resultado: se imprime todo el contenido del archivo byte por byte en formato numérico.

Ejemplo 2: FileOutputStream – Escritura básica de bytes

Se crea o sobrescribe "salida.bin" y se escribe un array de bytes directamente. El FileOutputStream envía los bytes al archivo; al cerrar (*try-with-resources*) flush() asegura que todos los datos pendientes se escriban en disco. Resultado: el archivo de salida contiene exactamente los bytes definidos en el array.

Ejemplo 3: DataInputStream/DataOutputStream – Lectura y escritura de tipos primitivos

Se utiliza DataOutputStream para escribir diferentes tipos primitivos (*int, double, String UTF*) en un archivo binario. Luego DataInputStream lee los datos en el mismo orden. El orden de lectura debe coincidir exactamente con el orden de escritura. Resultado: demostración de escritura y lectura tipada de datos primitivos.

Ejemplo 4: JDBC – Conexión y consulta básica

Se establece una conexión con una base de datos MySQL, se crea un Statement y se ejecuta una consulta SELECT. Los resultados se recorren con un ResultSet y se muestran por consola. Finalmente todos los recursos se cierran en orden inverso. Resultado: lista de usuarios de la base de datos mostrada en consola.

Ejemplo 5: Properties – Carga y uso de configuración

Se crea un objeto Properties y se carga desde un archivo "config.properties". Se leen diferentes propiedades con valores por defecto. Si el archivo no existe, se crean propiedades por defecto y se guardan. Resultado: demostración de carga, lectura y guardado de propiedades de configuración.

EJERCICIOS OBLIGATORIOS

Ejercicio 1: Gestor de Inventario Binario

Objetivo: Crear un sistema que guarde y lea productos desde un archivo binario usando DataInputStream/DataOutputStream.

Firmas de funciones:

```
/**
 * Escribe un producto en el archivo binario
 * @param archivo ruta del archivo donde guardar
 * @param producto objeto Producto a guardar
 * @throws IOException si hay error al escribir
 */
public static void escribirProducto(String archivo, Producto producto) throws IOException

/**
 * Lee todos los productos del archivo binario
 * @param archivo ruta del archivo a leer
 * @return lista de productos leídos
 * @throws IOException si hay error al leer
 */
public static List<Producto> leerProductos(String archivo) throws IOException

/**
 * Añade un producto al final del archivo (modo append)
 * @param archivo ruta del archivo
 * @param producto producto a añadir
 * @throws IOException si hay error
 */
public static void agregarProducto(String archivo, Producto producto) throws IOException
```

Clases requeridas:

```
class Producto {
  private int id;
  private String nombre;
  private double precio;
  private int stock;
  // Constructor, getters y setters
}
```

Casos de uso:

Entrada (*código*):

```
Producto p1 = new Producto(1, "Laptop", 999.99, 10);
Producto p2 = new Producto(2, "Mouse", 19.99, 50);
escribirProducto("inventario.dat", p1);
agregarProducto("inventario.dat", p2);
```

Salida por consola:

```
Producto guardado: Laptop
Producto añadido: Mouse
```

Lectura:

```
List<Producto> productos = leerProductos("inventario.dat");
for (Producto p : productos) {
  System.out.println(p);
}
```

Salida:

```
ID: 1, Nombre: Laptop, Precio: 999.99, Stock: 10
ID: 2, Nombre: Mouse, Precio: 19.99, Stock: 50
```

Buenas prácticas:

- Usar DataInputStream/DataOutputStream para tipos primitivos
- Try-with-resources para gestión automática de recursos
- Mantener orden de lectura = orden de escritura
- Validar que el archivo existe antes de leer
- Manejar EOFException al leer hasta el final

Ejercicio 2: Sistema de Usuarios con JDBC

Objetivo: Implementar un CRUD completo de usuarios usando JDBC con PreparedStatement.

Firmas de funciones:

```
/**
 * Crea la tabla usuarios si no existe
 * @param conn conexión a la base de datos
 * @throws SQLException si hay error al crear
 */
public static void crearTabla(Connection conn) throws SQLException

/**
 * Inserta un nuevo usuario en la base de datos
 * @param conn conexión activa
 * @param nombre nombre del usuario
 * @param email email del usuario
 * @param edad edad del usuario
 * @return ID generado del usuario insertado
 * @throws SQLException si hay error
 */
public static int insertarUsuario(Connection conn, String nombre, String email, int edad)
```

```
    throws SQLException
/**
 * Busca usuarios por nombre (búsqueda parcial)
 * @param conn conexión activa
 * @param nombre fragmento de nombre a buscar
 * @return lista de usuarios encontrados
 * @throws SQLException si hay error
 */
public static List<Usuario> buscarPorNombre(Connection conn, String nombre)
    throws SQLException
/**
 * Actualiza el email de un usuario
 * @param conn conexión activa
 * @param id ID del usuario
 * @param nuevoEmail nuevo email
 * @return true si se actualizó, false si no existe
 * @throws SQLException si hay error
 */
public static boolean actualizarEmail(Connection conn, int id, String nuevoEmail)
    throws SQLException
/**
 * Elimina un usuario por ID
 * @param conn conexión activa
 * @param id ID del usuario a eliminar
 * @return true si se eliminó, false si no existía
 * @throws SQLException si hay error
 */
public static boolean eliminarUsuario(Connection conn, int id) throws SQLException
```

Clase requerida:

```
class Usuario {
    private int id;
    private String nombre;
    private String email;
    private int edad;
    // Constructor, getters y setters
}
```

Casos de uso:

Código:

```
String url = "jdbc:mysql://localhost:3306/testdb";
Connection conn = DriverManager.getConnection(url, "root", "password");
crearTabla(conn);
int id1 = insertarUsuario(conn, "Juan Pérez", "juan@email.com", 25);
int id2 = insertarUsuario(conn, "María García", "maria@email.com", 30);
List<Usuario> usuarios = buscarPorNombre(conn, "Juan");
for (Usuario u : usuarios) {
    System.out.println(u);
}
actualizarEmail(conn, id1, "juan.nuevo@email.com");
eliminarUsuario(conn, id2);
conn.close();
```

Salida por consola:

```
Tabla 'usuarios' creada
Usuario insertado con ID: 1
Usuario insertado con ID: 2
Usuarios encontrados:
  ID: 1, Nombre: Juan Pérez, Email: juan@email.com, Edad: 25
```

Email actualizado para usuario ID: 1

Usuario eliminado: ID 2

Buenas prácticas:

- Usar PreparedStatement para prevenir inyección SQL
- Cerrar recursos con try-with-resources
- Usar transacciones para operaciones múltiples
- Validar parámetros antes de ejecutar SQL
- Manejar SQLException con mensajes descriptivos

Ejercicio 3: Configurador de Aplicación con Properties

Objetivo: Crear un sistema de configuración que lea, modifique y guarde parámetros usando Properties, con validación y valores por defecto.

Firmas de funciones:

```
/**
 * Carga la configuración desde archivo o crea una por defecto
 * @param archivo ruta del archivo de configuración
 * @return objeto Properties cargado
 * @throws IOException si hay error de lectura
 */
public static Properties cargarConfiguracion(String archivo) throws IOException

/**
 * Obtiene una propiedad como String con valor por defecto
 * @param props objeto Properties
 * @param clave clave de la propiedad
 * @param valorDefecto valor si no existe
 * @return valor de la propiedad o valorDefecto
```

```
 */
public static String getString(Properties props, String clave, String valorDefecto)
/**
 * Obtiene una propiedad como int con validación
 * @param props objeto Properties
 * @param clave clave de la propiedad
 * @param valorDefecto valor si no existe o es inválido
 * @return valor int de la propiedad
 */
public static int getInt(Properties props, String clave, int valorDefecto)
/**
 * Obtiene una propiedad como boolean
 * @param props objeto Properties
 * @param clave clave de la propiedad
 * @param valorDefecto valor si no existe
 * @return valor boolean de la propiedad
 */
public static boolean getBoolean(Properties props, String clave, boolean valorDefecto)
/**
 * Guarda la configuración en archivo
 * @param props objeto Properties a guardar
 * @param archivo ruta del archivo destino
 * @param comentario comentario para el archivo
 * @throws IOException si hay error de escritura
 */
public static void guardarConfiguracion(Properties props, String archivo, String
comentario)
    throws IOException
/**
 * Muestra todas las propiedades por consola
```

** @param props objeto Properties*

**/*

public static void mostrarConfiguracion(Properties props)

Variables requeridas:

- Configuración de base de datos (*host, puerto, nombre, usuario, password*)
- Configuración de aplicación (*título, versión, debug, idioma*)
- Configuración de interfaz (*tema, tamaño_fuente*)

Casos de uso:

Código:

```
Properties config = cargarConfiguracion("app.properties");
// Leer configuración
String dbHost = getString(config, "db.host", "localhost");
int dbPort = getInt(config, "db.port", 3306);
boolean debug = getBoolean(config, "app.debug", false);
System.out.println("=== Configuración Actual ===");
mostrarConfiguracion(config);
// Modificar configuración
config.setProperty("app.idioma", "es");
config.setProperty("ui.tema", "oscuro");
config.setProperty("db.port", "3307");
guardarConfiguracion(config, "app.properties", "Configuración de Mi Aplicación");
```

Salida por consola:

```
Configuración cargada: app.properties
=== Configuración Actual ===
db.host = localhost
db.port = 3306
db.name = mi_base_datos
```

db.user = admin

app.titulo = Mi Aplicación

app.version = 1.0.0

app.debug = false

app.idioma = en

ui.tema = claro

ui.tamano_fuente = 12

Configuración guardada: app.properties

Archivo generado (*app.properties*):

#Configuración de Mi Aplicación

#Sun Oct 26 22:00:00 CET 2025

app.debug=false

app.idioma=es

app.titulo=Mi Aplicación

app.version=1.0.0

db.host=localhost

db.name=mi_base_datos

db.port=3307

db.user=admin

ui.tamano_fuente=12

ui.tema=oscuro

Buenas prácticas:

- Proporcionar valores por defecto para todas las propiedades
- Validar tipos al convertir (*NumberFormatException*)
- Usar try-catch para lectura/escritura de archivos
- Documentar el archivo con comentarios claros
- Separar propiedades por categorías (*db, app, ui*)

EJERCICIOS OPCIONALES

Ejercicio Opcional 1: Exportador de Base de Datos a Archivo Binario

Objetivo: Leer datos de una tabla JDBC y exportarlos a un archivo binario usando DataOutputStream, luego poder importarlos de vuelta.

Firmas de funciones:

```
/**
 * Exporta todos los productos de la base de datos a archivo binario
 * @param conn conexión JDBC activa
 * @param archivo ruta del archivo destino
 * @return número de productos exportados
 * @throws SQLException si hay error de BD
 * @throws IOException si hay error de archivo
 */
public static int exportarProductos(Connection conn, String archivo)
    throws SQLException, IOException
/**
 * Importa productos desde archivo binario a la base de datos
 * @param conn conexión JDBC activa
 * @param archivo ruta del archivo fuente
 * @return número de productos importados
 * @throws SQLException si hay error de BD
 * @throws IOException si hay error de archivo
 */
public static int importarProductos(Connection conn, String archivo)
    throws SQLException, IOException
```

Ejemplo de uso:

Código:

```
String url = "jdbc:mysql://localhost:3306/inventario";
Connection conn = DriverManager.getConnection(url, "root", "admin");
// Exportar
int exportados = exportarProductos(conn, "backup_productos.dat");
System.out.println("Productos exportados: " + exportados);
// Limpiar tabla (simulación de pérdida de datos)
Statement stmt = conn.createStatement();
stmt.executeUpdate("DELETE FROM productos");
// Importar
int importados = importarProductos(conn, "backup_productos.dat");
System.out.println("Productos importados: " + importados);
conn.close();
```

Salida por consola:

```
Exportando productos...
Producto exportado: ID=1, Nombre=Laptop
Producto exportado: ID=2, Nombre=Mouse
Producto exportado: ID=3, Nombre=Teclado
Productos exportados: 3
Importando productos...
Producto importado: ID=1, Nombre=Laptop
Producto importado: ID=2, Nombre=Mouse
Producto importado: ID=3, Nombre=Teclado
Productos importados: 3
```

Resultado esperado: Sistema funcional de backup/restore de datos de BD a archivo binario.

Ejercicio Opcional 2: Migrador de Configuración Properties a Base de Datos

Objetivo: Leer un archivo Properties y migrar todas las configuraciones a una tabla de base de datos, con posibilidad de sincronización bidireccional.

Firmas de funciones:

```
/**
 * Migra todas las propiedades de archivo a base de datos
 * @param archivo ruta del archivo Properties
 * @param conn conexión JDBC
 * @return número de propiedades migradas
 * @throws IOException si hay error al leer archivo
 * @throws SQLException si hay error de BD
 */
public static int migrarPropertiesABD(String archivo, Connection conn)
    throws IOException, SQLException
/**
 * Exporta configuración de base de datos a archivo Properties
 * @param conn conexión JDBC
 * @param archivo ruta del archivo destino
 * @return número de propiedades exportadas
 * @throws SQLException si hay error de BD
 * @throws IOException si hay error al escribir
 */
public static int exportarBDaProperties(Connection conn, String archivo)
    throws SQLException, IOException
/**
 * Sincroniza: actualiza BD con valores de Properties que hayan cambiado
 * @param archivo ruta del archivo Properties
 * @param conn conexión JDBC
 * @return número de propiedades actualizadas
```

```
 * @throws IOException si hay error al leer
 * @throws SQLException si hay error de BD
 */
public static int sincronizarPropiedades(String archivo, Connection conn)
    throws IOException, SQLException
```

Ejemplo de uso:

Archivo (*config.properties*):

```
db.host=localhost
db.port=3306
app.nombre=Mi App
app.version=2.0
```

Código:

```
String url = "jdbc:mysql://localhost:3306/config_db";
Connection conn = DriverManager.getConnection(url, "root", "admin");
// Migrar de archivo a BD
int migradas = migrarPropertiesABD("config.properties", conn);
System.out.println("Propiedades migradas a BD: " + migradas);
// Modificar en BD
PreparedStatement pstmt = conn.prepareStatement(
    "UPDATE configuracion SET valor = ? WHERE clave = ?");
pstmt.setString(1, "3307");
pstmt.setString(2, "db.port");
pstmt.executeUpdate();
// Exportar de BD a archivo
int exportadas = exportarBDaProperties(conn, "config_exportado.properties");
System.out.println("Propiedades exportadas a archivo: " + exportadas);
conn.close();
```

Salida por consola:

Creando tabla 'configuracion'...

Migrando propiedades a BD...

db.host = localhost

db.port = 3306

app.nombre = Mi App

app.version = 2.0

Propiedades migradas a BD: 4

Exportando configuración de BD a archivo...

Propiedades exportadas a archivo: 4

Resultado esperado: Sistema completo de migración y sincronización entre Properties y base de datos.

Ejercicio Opcional 3: Analizador de Archivos Binarios con Reporte

Objetivo: Crear una herramienta que analice archivos binarios generados con DataOutputStream y genere un reporte detallado de su estructura y contenido.

Firmas de funciones:

```
/**
 * Analiza un archivo binario y genera reporte de su estructura
 * @param archivo ruta del archivo a analizar
 * @return objeto Reporte con la información del análisis
 * @throws IOException si hay error al leer
 */
public static Reporte analizarArchivoBinario(String archivo) throws IOException

/**
 * Intenta detectar el tipo de dato en la posición actual
 * @param dis DataInputStream posicionado
```

```
 * @return String con el tipo detectado
 * @throws IOException si hay error
 */
private static String detectarTipoDato(DataInputStream dis) throws IOException
/**
 * Guarda el reporte en un archivo de texto
 * @param reporte objeto Reporte
 * @param archivo ruta del archivo destino
 * @throws IOException si hay error al escribir
 */
public static void guardarReporte(Reporte reporte, String archivo) throws IOException
/**
 * Muestra el reporte por consola con formato
 * @param reporte objeto Reporte
 */
public static void mostrarReporte(Reporte reporte)
```

Clase requerida:

```
class Reporte {
    private String nombreArchivo;
    private long tamañoBytes;
    private List<ElementoDato> elementos;
    private int totalInts;
    private int totalDoubles;
    private int totalStrings;
    // Constructor, getters y setters
}
class ElementoDato {
    private int posicion;
    private String tipo;
```

```
    private String valor;
    // Constructor, getters y setters
}
```

Ejemplo de uso:

Crear archivo binario:

```
DataOutputStream dos = new DataOutputStream(new FileOutputStream("datos.dat"));
dos.writeInt(100);
dos.writeUTF("Producto A");
dos.writeDouble(99.99);
dos.writeBoolean(true);
dos.writeInt(200);
dos.close();
```

Analizar:

```
Reporte reporte = analizarArchivoBinario("datos.dat");
mostrarReporte(reporte);
guardarReporte(reporte, "reporte_datos.txt");
```

Salida por consola:

```
=== Reporte de Análisis de Archivo Binario ===
Archivo: datos.dat
Tamaño: 29 bytes
```

Estructura detectada:

```
 [Pos 0-3]   INT:     100
 [Pos 4-14]  UTF:     "Producto A" (10 caracteres)
 [Pos 15-22] DOUBLE:  99.99
 [Pos 23]    BOOLEAN: true
 [Pos 24-27] INT:     200
```

Resumen:

Enteros (int): 2

Decimales (double): 1

Cadenas (UTF): 1

Booleanos: 1

Total elementos: 5

Resultado esperado: Herramienta completa de análisis de archivos binarios con generación de reportes detallados.

FORMATO DE ENTREGA

Requisitos del código:

- **JavaDoc** en todas las clases y métodos públicos
- **Comentarios explicativos** en lógica compleja
- **Nombres descriptivos** de variables y métodos
- **Manejo apropiado de excepciones** con try-catch y try-with-resources
- **Cierre de recursos** garantizado
- **Código compilable y ejecutable** sin errores
- **Paquete apropiado**: todos los ejercicios en package AccesoDatosBinarios