PRÁCTICA DE ACCESO A FICHEROS EN JAVA

FUNDAMENTOS DEL SISTEMA I/O DE JAVA

Jerarquía de clases

Java I/O se basa 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 segundo grupo, diseñadas específicamente para manejar caracteres Unicode de forma eficiente.

Conceptos clave:

- **Buffer:** memoria intermedia que reduce accesos costosos al disco
- **Encoding/Charset:** forma de convertir bytes a caracteres (*UTF-8, ISO-8859-1, etc.*)
- **Stream:** flujo de datos secuencial
- **File pointer:** posición actual de lectura/escritura en un archivo

---

FILEREADER - LECTURA BÁSICA DE CARACTERES

Descripción técnica

FileReader es una clase de conveniencia que extiende InputStreamReader, específicamente diseñada para leer archivos de caracteres. Internamente, abre un FileInputStream y lo envuelve con conversión de bytes a caracteres.

Arquitectura interna:

FileReader → InputStreamReader → FileInputStream → Sistema Operativo

Constructores detallados:

// Básico - usa codificación del sistema

FileReader(String fileName) throws FileNotFoundException

```
// Con objeto File - equivalente al anterior
FileReader(File file) throws FileNotFoundException
// Java 11+ - especifica codificación (RECOMENDADO)
FileReader(String fileName, Charset charset) throws IOException
```

Métodos principales con detalles:

1. **int read()**
    - **Retorna:** valor Unicode del carácter (*0-65535*) o -1 al EOF
    - Bloquea hasta que hay un carácter disponible
    - **Ejemplo de uso:**

```
int c;
while ((c = reader.read()) != -1) {
   System.out.print((char) c);
}
```

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

```
char[] buffer = new char[1024];
int charsRead = reader.read(buffer);
```

3. **int read(*char[] cbuf, int offset, int length*)**
    - Lee hasta length caracteres en cbuf[offset...offset+length-1]
    - Permite reutilizar arrays grandes leyendo en segmentos

```
char[] bigBuffer = new char[4096];
int read = reader.read(bigBuffer, 1000, 500); // Lee 500 chars en posición 1000
```

Limitaciones importantes:

- **Sin buffer:** cada read() puede generar una llamada al sistema operativo
- **Codificación implícita:** usa Charset.defaultCharset() que varía según la JVM y SO
- **No thread-safe:** no debe compartirse entre hilos sin sincronización

Cuándo usar FileReader:

- Archivos pequeños (< *1MB*) donde el rendimiento no es crítico
- Lecturas puntuales de configuración
- Prototipado rápido
- **Evitar para:** archivos grandes, procesamiento intensivo, aplicaciones multiplataforma

---

FILEWRITER - ESCRITURA BÁSICA DE CARACTERES

Descripción técnica

FileWriter extiende OutputStreamWriter y proporciona una interfaz conveniente para escribir caracteres en archivos. Maneja la conversión de caracteres Unicode a bytes según la codificación especificada.

Arquitectura interna:

FileWriter → OutputStreamWriter → FileOutputStream → Sistema Operativo

Constructores con opciones:

// Sobrescribe archivo existente

FileWriter(String fileName) throws IOException

// Control de modo append

FileWriter(String fileName, boolean append) throws IOException

// Con objeto File

FileWriter(File file) throws IOException

FileWriter(File file, boolean append) throws IOException

// Java 11+ - Control de codificación

FileWriter(String fileName, Charset charset) throws IOException

FileWriter(String fileName, Charset charset, boolean append) throws IOException

Métodos de escritura detallados:

1. **void write(*int c*)**
    - Escribe un único carácter (*toma los 16 bits bajos del int*)

writer.write(65); *// Escribe 'A'*

writer.write('ñ'); *// Escribe 'ñ' (Unicode correcto)*

2. **void write(*String str*)**
    - Escribe la cadena completa de una vez
    - Más eficiente que escribir carácter por carácter

writer.write("Línea completa con acentos ñáéíóú");

3. **void write(*char[] cbuf*)**
    - Escribe todo el array de caracteres

char[] datos = {'H', 'o', 'l', 'a'};

writer.write(datos);

4. **void write(*char[] cbuf, int off, int len*)**
    - Escribe segmento específico del array

char[] buffer = "Hola Mundo".toCharArray();

writer.write(buffer, 5, 5); *// Escribe "Mundo"*

5. **void write(*String str, int off, int len*)**
    - Escribe substring específico

writer.write("Buenos días", 7, 4); *// Escribe "días"*

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 BufferedWriter
- flush() es costoso, úsalo solo cuando necesites garantizar escritura inmediata

---

BUFFEREDREADER - LECTURA EFICIENTE CON BUFFER

Descripción avanzada

BufferedReader implementa el patrón Decorator, envolviendo cualquier Reader y añadiendo capacidades de buffering. Mantiene un array interno de caracteres que rellena de golpe, reduciendo drásticamente las llamadas I/O.

Arquitectura con buffering:

BufferedReader[buffer 8192 chars] → FileReader → FileInputStream → SO

Ventajas del buffering:

- **Reduce I/O:** en lugar de 1000 llamadas read(), hace 1 read(8192) + accesos a memoria
- **Mejor rendimiento:** acceso a memoria vs. acceso a disco (*diferencia de microsegundos*)
- **Lectura por líneas:** readLine() optimizada para archivos de texto

Constructores y configuración:

```
// Buffer por defecto (8192 caracteres = 16KB aprox)
BufferedReader(Reader in)
// Buffer personalizado
BufferedReader(Reader in, int bufferSize)
```

Tamaños de buffer recomendados:

- Archivos pequeños (< *100KB*): buffer por defecto (*8192*)
- Archivos medianos (*100KB - 10MB*): 16384 - 32768 caracteres
- Archivos grandes (> *10MB*): 65536 caracteres o más
- Memoria limitada: reducir a 4096 o 2048

Métodos optimizados:

1. **String readLine()**
    - Lee una línea completa (*hasta \n, \r\n, o \r*)
    - Retorna null al final del archivo
    - Excluye el terminador de línea del resultado

```
String line;
while ((line = reader.readLine()) != null) {
   System.out.println("Línea: " + line);
}
```

2. **int read()**
    - Mismo comportamiento que FileReader pero con buffer
    - Primero consume el buffer, luego rellena desde el archivo
3. **int read(char[] cbuf, int offset, int length)**
    - Lee eficientemente desde buffer interno
    - Combina datos del buffer con lecturas directas si es necesario
4. **Stream<String> lines() (*Java 8+*)**
    - Retorna un Stream para procesamiento funcional
    - **¡IMPORTANTE!:** debe cerrarse explícitamente

```
try (BufferedReader reader = Files.newBufferedReader(path)) {
   reader.lines()
        .filter(line -> line.contains("ERROR"))
        .forEach(System.out::println);
}
```

Métodos de control:

- **void mark(int readAheadLimit):** marca posición actual para volver después
- **void reset():** vuelve a la posición marcada
- **boolean markSupported():** siempre retorna true para BufferedReader
- **long skip(*long n*):** salta n caracteres eficientemente

Casos de uso específicos:

- **Análisis de logs:** procesamiento línea por línea de archivos grandes
- **Parsing de CSV/TSV:** lectura estructurada de datos tabulares
- **Configuración:** archivos properties, INI, etc.
- **Streaming:** procesamiento de archivos que no caben en memoria

---

BUFFEREDWRITER - ESCRITURA EFICIENTE CON BUFFER

Descripción avanzada

BufferedWriter envuelve cualquier Writer añadiendo un buffer de escritura. Acumula caracteres en memoria y los escribe en chunks grandes, optimizando el rendimiento especialmente para múltiples escrituras pequeñas.

Arquitectura con buffering:

BufferedWriter[buffer 8192 chars] → FileWriter → FileOutputStream → SO

Proceso interno de buffering:

1. write() añade datos al buffer interno
2. Si el buffer se llena, se vacía automáticamente al Writer subyacente
3. flush() o close() fuerzan el vaciado inmediato

Constructores:

```
// Buffer estándar de 8192 caracteres
BufferedWriter(Writer out)
// Buffer personalizado (útil para optimización específica)
BufferedWriter(Writer out, int bufferSize)
```

Métodos de escritura optimizados:

1. **void write(*String str*)**
    - Escribe directamente en buffer interno
    - Si la cadena es mayor que el buffer, la escribe directamente

```
writer.write("Esta línea se almacena en buffer");
```

2. **void newLine()**
    - Escribe el separador de línea correcto según el SO
    - **Windows:** \r\n, **Unix/Linux:** \n, **Mac clásico:** \r
    - Portabilidad garantizada entre sistemas operativos

```
writer.write("Primera línea");
writer.newLine();
writer.write("Segunda línea");
```

3. **void write(*int c*)**
    - Añade carácter al buffer
    - Eficiente para construcción carácter por carácter

Métodos de control del buffer:

1. **void flush()**
    - Vacía buffer al Writer subyacente inmediatamente
    - **Cuándo usar:** antes de operaciones críticas, logging en tiempo real
    - **Costo:** operación relativamente cara, no abusar

```
writer.write("Log crítico");
writer.flush(); // Garantiza escritura inmediata
```

2. **void close()**
    - Hace flush() automáticamente y cierra el stream
    - Try-with-resources garantiza ejecución incluso con excepciones

Optimizaciones avanzadas:

- **Tamaño de buffer dinámico:** para archivos muy grandes, usar buffers de 32KB o 64KB
- **Batch writing:** agrupar múltiples write() antes de flush()
- **Combinación con StringBuilder:** construir líneas complejas en memoria antes de escribir

Patrones de uso eficientes:

```
// INEFICIENTE - flush en cada línea
writer.write(line1); writer.flush();
writer.write(line2); writer.flush();
```

// EFICIENTE - flush solo al final

writer.write(line1); writer.newLine();

writer.write(line2); writer.newLine();

writer.flush(); *// Una sola vez*

RANDOMACCESSFILE - ACCESO ALEATORIO Y BINARIO

Descripción completa

RandomAccessFile es una clase única en el ecosistema Java I/O que no hereda de InputStream/OutputStream ni de Reader/Writer. Proporciona acceso tanto de lectura como escritura a cualquier posición del archivo mediante un puntero de archivo móvil.

Características distintivas:

- Acceso no secuencial: puedes saltar a cualquier posición
- Lectura y escritura simultánea: mismo objeto para ambas operaciones
- Operaciones a nivel de bytes: maneja datos binarios nativamente
- Puntero de archivo: posición actual mantenida internamente

Modos de apertura detallados:

1. **"r" (*Read-only*)**
    - Solo lectura, archivo debe existir
    - Intento de escritura lanza IOException
    - Más seguro para análisis de datos
2. **"rw" (*Read-Write*)**
    - Lectura y escritura, crea archivo si no existe
    - Modo más común para aplicaciones de datos
3. **"rws" (*Read-Write-Sync*)**
    - Como "rw" pero sincroniza contenido y metadata en cada escritura

- Muy lento pero garantiza persistencia inmediata
- Útil para aplicaciones críticas (bases de datos)

4. **"rwd" (*Read-Write-Data*)**
    - Como "rw" pero sincroniza solo el contenido (no metadata)
    - Compromiso entre rendimiento y seguridad

Gestión del puntero de archivo:

1. **void seek(long pos)**
    - Posiciona el puntero en el byte pos (base 0)
    - Si pos > length(), el archivo se extenderá al escribir

```
raf.seek(0);      // Ir al inicio
raf.seek(100);    // Ir al byte 100
raf.seek(raf.length()); // Ir al final
```

2. **long getFilePointer()**
    - Retorna posición actual del puntero
    - Útil para guardar posiciones y volver después
3. **long length()**
    - Tamaño actual del archivo en bytes
    - Se actualiza automáticamente al escribir

Métodos de lectura primitiva:

```
// Lectura de tipos básicos (Big-Endian)
byte readByte()         // 1 byte con signo (-128 a 127)
int readUnsignedByte()  // 1 byte sin signo (0 a 255)
short readShort()       // 2 bytes con signo
int readUnsignedShort() // 2 bytes sin signo
int readInt()           // 4 bytes con signo
long readLong()         // 8 bytes con signo
float readFloat()       // 4 bytes IEEE 754
```

```
double readDouble()      // 8 bytes IEEE 754
// Lectura de cadenas
String readLine()        // Lee hasta \n (CUIDADO: usa ISO-8859-1)
String readUTF()         // Lee string con prefijo de longitud
// Lectura de arrays
int read(byte[] b)       // Llena array, retorna bytes leídos
int read(byte[] b, int off, int len) // Lee en segmento
```

Métodos de escritura primitiva:

```
// Escritura de tipos básicos
void writeByte(int v)    // Escribe byte bajo del int
void writeShort(int v)   // Escribe 2 bytes (big-endian)
void writeInt(int v)     // Escribe 4 bytes
void writeLong(long v)   // Escribe 8 bytes
void writeFloat(float v)  // 4 bytes IEEE 754
void writeDouble(double v)// 8 bytes IEEE 754
// Escritura de cadenas
void writeBytes(String s) // Cada char → byte bajo (ASCII)
void writeUTF(String s)   // String con prefijo de longitud UTF-8
// Escritura de arrays
void write(byte[] b)     // Escribe array completo
void write(byte[] b, int off, int len) // Escribe segmento
```

Diseño de registros de longitud fija:

Ejemplo de un registro de "Empleado":

```
// Layout del registro:
// int id (4 bytes) + nombre 50 bytes + double salario (8 bytes) = 62 bytes total
public static final int ID_SIZE = 4;
public static final int NOMBRE_SIZE = 50;
```

```
public static final int SALARIO_SIZE = 8;
public static final int RECORD_SIZE = ID_SIZE + NOMBRE_SIZE + SALARIO_SIZE;
// Escribir empleado en posición N
long position = employeeId * RECORD_SIZE;
raf.seek(position);
raf.writeInt(employeeId);
// Nombre de longitud fija (padding con espacios)
String paddedName = String.format("%-50s", nombre);
raf.writeBytes(paddedName);
raf.writeDouble(salario);
```

Consideraciones de encoding:

- writeBytes() solo escribe el byte bajo de cada char (pierde acentos)
- Para texto Unicode completo: convertir manualmente a bytes UTF-8
- writeUTF() usa formato especial de Java (no UTF-8 estándar)

Casos de uso especializado:

- **Bases de datos simples:** registros de longitud fija con índices
- **Archivos de configuración binarios:** headers + datos
- **Caches en disco:** estructuras de datos persistentes
- **Modificaciones parciales:** editar archivos grandes sin reescribirlos completamente

---

COMPARATIVA Y CRITERIOS DE SELECCIÓN

Tabla de comparación rápida:

Clase	Buffer	Encoding	Acceso	Mejor para
FileReader	NO	Sistema	Secuencial	Archivos pequeños, prototipado
FileWriter	NO	Sistema	Secuencial	Escritura simple, archivos pequeños
BufferedReader	SI	Configurable	Secuencial	Procesamiento de texto, archivos grandes
BufferedWriter	SI	Configurable	Secuencial	Escritura eficiente, múltiples líneas
RandomAccessFile	SI	Binario	Aleatorio	Bases de datos, estructuras complejas

Árbol de decisión para elegir clase:

1. ¿Necesitas acceso no secuencial? → RandomAccessFile
2. ¿Archivo > 100KB o muchas operaciones I/O? → BufferedReader/BufferedWriter
3. ¿Necesitas controlar encoding? → Usar con Charset explícito
4. ¿Prototipado rápido? → FileReader/FileWriter

MEJORES PRÁCTICAS Y OPTIMIZACIONES

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

```
// CORRECTO - cierre automático garantizado
try (BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF_8)) {
    // operaciones de lectura
} catch (IOException e) {
    // manejo de errores
}
```

```
// ❌ INCORRECTO - posible leak de recursos
BufferedReader reader = Files.newBufferedReader(path);
// operaciones de lectura
reader.close(); // ¿Qué pasa si hay excepción?
```

2. Encoding explícito siempre:

```
// ✅ CORRECTO - encoding predecible
Files.newBufferedWriter(path, StandardCharsets.UTF_8)
// ❌ PROBLEMÁTICO - encoding del sistema
new FileWriter("archivo.txt")
```

3. Optimización de tamaño de buffer:

```
// Para archivos grandes
int bufferSize = 64 * 1024; // 64KB
BufferedReader reader = new BufferedReader(new FileReader(file), bufferSize);
```

4. Manejo robusto de excepciones:

```
try (BufferedWriter writer = Files.newBufferedWriter(path, StandardCharsets.UTF_8)) {
    writer.write(content);
} catch (NoSuchFileException e) {
    System.err.println("Archivo no encontrado: " + e.getFile());
} catch (AccessDeniedException e) {
    System.err.println("Sin permisos: " + e.getFile());
} catch (IOException e) {
    System.err.println("Error I/O: " + e.getMessage());
}
```

5. Alternativas modernas (NIO.2):

```
// Lectura completa de archivo pequeño
List<String> lines = Files.readAllLines(path, StandardCharsets.UTF_8);
// Streaming para archivos grandes
try (Stream<String> stream = Files.lines(path, StandardCharsets.UTF_8)) {
    stream.filter(line -> line.contains("ERROR"))
          .forEach(System.out::println);
}
// Escritura simple
Files.write(path, content.getBytes(StandardCharsets.UTF_8));
```

EJEMPLOS DE INTRODUCCIÓN

Ejemplo 1: FileReader – Lectura básica

Se abre un flujo de caracteres sobre el archivo “entrada.txt” y, carácter a carácter, se lee hasta alcanzar el fin de archivo (*read() devuelve –1*). Cada entero leído se convierte a char y se muestra por consola. El bloque try-with-resources garantiza el cierre automático del FileReader y captura posibles errores de E/S. Resultado: se imprime todo el contenido del archivo sin agrupaciones ni buffers, directamente carácter a carácter.

Ejemplo 2: FileWriter – Escritura básica

Se crea o sobrescribe “salida.txt” y se escribe de una sola vez la cadena contenido, que incluye saltos de línea embebidos. El FileWriter envía los caracteres directamente 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 las tres líneas definidas.

Ejemplo 3: BufferedReader – Lectura línea por línea

Un BufferedReader envuelve un FileReader para añadir un buffer de entrada. Con readLine() se obtiene una línea completa en cada llamada, hasta que retorna null. Dentro del bucle se numera cada línea y se imprime con su número. El buffering minimiza llamadas al sistema y el try-with-resources cierra el stream al finalizar. Resultado: cada línea del archivo “entrada.txt” aparece numerada en pantalla.

Ejemplo 4: BufferedWriter – Escritura eficiente

BufferedWriter envuelve FileWriter aportando un buffer de salida. Se recorre un array de cadenas y, por cada elemento, se escribe la línea y se invoca newLine(), que añade automáticamente el separador de línea propio del sistema operativo. Al cerrar el stream, todos los datos en buffer se vuelcan de golpe en “salida_buffer.txt”. Resultado: un archivo con cuatro líneas de texto, una por cada elemento del array.

Ejemplo 5: RandomAccessFile – Acceso aleatorio

Se abre “datos.bin” en modo lectura/escritura. Primero se escribe la palabra “INICIO” al comienzo; luego seek(20) mueve el puntero al byte 20 para escribir “MEDIO”; después seek(40) posiciona en el byte 40 y escribe “FINAL”. Para la lectura, se vuelve a seek(0) y readLine() lee hasta el primer salto de línea (*si existe*), luego se lee desde la posición 20. Finalmente, raf.length() devuelve el tamaño total del archivo en bytes. Resultado: demostración de escritura y lectura en ubicaciones arbitrarias dentro del mismo archivo sin necesidad de reescribirlo entero.

EJERCICIOS OBLIGATORIOS

Ejercicio 1: Contador de Palabras y Estadísticas

Objetivo: Leer un archivo de texto y generar estadísticas completas sobre su contenido.

Firmas de funciones:

```
/**
 * Lee un archivo y cuenta palabras, líneas y caracteres
 * @param nombreArchivo ruta del archivo a analizar
 * @return objeto EstadisticasTexto con los resultados
 * @throws IOException si hay error al leer el archivo
 */
public static EstadisticasTexto analizarArchivo(String nombreArchivo) throws IOException

/**
 * Escribe las estadísticas en un archivo de salida
 * @param estadisticas objeto con las estadísticas
 * @param archivoSalida ruta donde guardar el resultado
 * @throws IOException si hay error al escribir
 */
public static void guardarEstadisticas(EstadisticasTexto estadisticas, String archivoSalida)
throws IOException
```

Clases requeridas:

```
class EstadisticasTexto {
    private int numeroLineas;
    private int numeroPalabras;
    private int numeroCaracteres;
    private String palabraMasLarga;

    // Constructor, getters y setters
}
```

Casos de uso:

Entrada (*archivo.txt*):

```
Hola mundo
Java es genial
Programación
```

Salida por consola:

```
=== Estadísticas del archivo ===
Líneas: 3
Palabras: 5
Caracteres: 35
Palabra más larga: Programación (13 caracteres)
```

Buenas prácticas:

- Usar BufferedReader para lectura eficiente
- Try-with-resources para gestión automática de recursos
- Charset UTF-8 explícito
- Validar que el archivo existe antes de leer
- Manejar IOException adecuadamente

Ejercicio 2: Merge de Archivos con Filtrado

Objetivo: Leer múltiples archivos de texto, filtrar líneas según un criterio y combinar el resultado en un único archivo de salida.

Firmas de funciones:

```
/**
 * Combina múltiples archivos en uno solo, filtrando líneas
 * @param archivosEntrada array con las rutas de los archivos a combinar
 * @param archivoSalida ruta del archivo resultado
 * @param filtro palabra que debe contener la línea para incluirse (null = todas)
 * @return número total de líneas escritas
 * @throws IOException si hay error de lectura/escritura
 */
public static int combinarArchivos(String[] archivosEntrada, String archivoSalida, String filtro) throws IOException

/**
 * Verifica si una línea cumple el criterio de filtrado
 * @param linea línea a evaluar
 * @param filtro criterio de búsqueda (null = siempre true)
 * @return true si la línea debe incluirse
 */
private static boolean cumpleFiltro(String linea, String filtro)
```

Variables requeridas:

- Contador de líneas procesadas
- Contador de líneas escritas
- BufferedReader para cada archivo de entrada
- BufferedWriter para el archivo de salida

Casos de uso:

Entrada:

- **archivo1.txt:** "Java es poderoso\nPython también\nJava es popular"
- **archivo2.txt:** "JavaScript en web\nJava en backend"
- **Filtro:** "Java"

Salida (*combinado.txt*):

Java es poderoso

Java es popular

Java en backend

Consola:

Procesando archivo1.txt: 2 líneas coinciden

Procesando archivo2.txt: 1 línea coincide

Total: 3 líneas escritas en combinado.txt

Buenas prácticas:

- Validar que todos los archivos de entrada existen
- Usar try-with-resources anidados o múltiples
- Encoding UTF-8
- Añadir encabezados indicando origen de cada sección (opcional)
- Manejo robusto de excepciones

Ejercicio 3: Sistema de Log con Rotación

Objetivo: Implementar un sistema de logging que escriba en un archivo y lo rote cuando alcance cierto tamaño.

Firmas de funciones:

```
/**
 * Escribe un mensaje en el log con timestamp
 * @param mensaje contenido a registrar
 * @param nivel nivel del log (INFO, WARNING, ERROR)
 * @throws IOException si hay error al escribir
 */
public void escribirLog(String mensaje, NivelLog nivel) throws IOException

/**
 * Verifica si el archivo debe rotarse y ejecuta la rotación
 * @return true si se realizó la rotación
 * @throws IOException si hay error en la rotación
 */
private boolean rotarSiNecesario() throws IOException

/**
 * Obtiene el tamaño actual del archivo de log
 * @return tamaño en bytes
 */
private long obtenerTamanoLog()
```

Clases requeridas:

```
enum NivelLog {
   INFO, WARNING, ERROR
}
class SistemaLog {
   private String archivoLog;
   private long tamanoMaximo; // en bytes
   private int numeroRotacion;
   public SistemaLog(String archivoLog, long tamanoMaximo) {
      // Constructor
   }
   // Métodos anteriores
}
```

Casos de uso:

Uso:

```
SistemaLog log = new SistemaLog("app.log", 1024); // 1KB máximo
log.escribirLog("Aplicación iniciada", NivelLog.INFO);
log.escribirLog("Usuario conectado", NivelLog.INFO);
log.escribirLog("Error de conexión", NivelLog.ERROR);
// ... más mensajes hasta superar 1KB
```

Salida (*app.log*):

```
[2025-10-14 10:30:15] [INFO] Aplicación iniciada
[2025-10-14 10:30:16] [INFO] Usuario conectado
[2025-10-14 10:30:17] [ERROR] Error de conexión
```

Cuando se supera el tamaño, se renombra a app.log.1 y se crea nuevo app.log

Consola:

```
Log escrito: Aplicación iniciada
Log escrito: Usuario conectado
```

ROTACIÓN: app.log renombrado a app.log.1

Log escrito: Error de conexión

Buenas prácticas:

- Usar BufferedWriter para escritura eficiente
- flush() después de cada escritura crítica
- Try-with-resources donde sea posible
- Formato de fecha ISO 8601
- Validar parámetros en constructor
- Sincronización si se usa en entorno multi-thread (opcional)

EJERCICIOS OPCIONALES

Ejercicio Opcional 1: Parser de JSON Simple

Objetivo: Leer y escribir archivos JSON básicos sin usar librerías externas, o usando Gson si está disponible.

Firmas de funciones:

```
/**
 * Lee un archivo JSON y extrae pares clave-valor simples
 * @param archivoJson ruta del archivo JSON
 * @return Map con las claves y valores parseados
 * @throws IOException si hay error de lectura
 */
public static Map<String, String> leerJsonSimple(String archivoJson) throws IOException

/**
 * Escribe un Map como archivo JSON formateado
 * @param datos Map con los datos a escribir
 * @param archivoJson ruta del archivo de salida
 * @throws IOException si hay error de escritura
 */
public static void escribirJsonSimple(Map<String, String> datos, String archivoJson)
throws IOException
```

Ejemplo de uso:

Entrada (*config.json*):

```
{
  "host": "localhost",
  "puerto": "8080",
  "debug": "true"
}
```

Código:

```
Map<String, String> config = leerJsonSimple("config.json");
System.out.println("Host: " + config.get("host"));
config.put("version", "1.0");
escribirJsonSimple(config, "config_nuevo.json");
```

Salida por consola:

```
JSON leído: 3 propiedades
Host: localhost
JSON escrito: 4 propiedades en config_nuevo.json
```

Resultado esperado: Manejo básico de JSON con parsing manual de cadenas o usando Gson para estructuras más complejas.

Ejercicio Opcional 2: Carga de Variables de Entorno desde .env

Objetivo: Leer un archivo .env y cargar las variables en un Map que simule variables de entorno.

Firmas de funciones:

```
/**
 * Lee un archivo .env y carga las variables
 * @param archivoEnv ruta del archivo .env
 * @return Map con las variables cargadas
 * @throws IOException si hay error de lectura
 */
public static Map<String, String> cargarEnv(String archivoEnv) throws IOException
/**
 * Obtiene el valor de una variable de entorno
 * @param clave nombre de la variable
 * @param valorPorDefecto valor si la variable no existe
 * @return valor de la variable o valorPorDefecto
```

*/

public static String getEnv(String clave, String valorPorDefecto)

Ejemplo de uso:

Entrada (*.env*):

DB_HOST=localhost

DB_PORT=5432

DB_USER=admin

DB_PASSWORD=secret123

Comentario ignorado

DEBUG=true

Código:

Map<String, String> env = cargarEnv(".env");

System.out.println("Base de datos: " + env.get("DB_HOST") + ":" + env.get("DB_PORT"));

String debug = getEnv("DEBUG", "false");

Salida por consola:

Cargadas 5 variables desde .env

Base de datos: localhost:5432

Debug mode: true

Resultado esperado: Sistema funcional de carga de configuración desde archivo .env, ignorando comentarios y líneas vacías.

Ejercicio Opcional 3: Backup Incremental

Objetivo: Crear un sistema que haga backup solo de archivos modificados desde el último backup.

Firmas de funciones:

```
/**
 * Realiza backup incremental de una carpeta
 * @param carpetaOrigen ruta de la carpeta a respaldar
 * @param carpetaDestino ruta donde guardar el backup
 * @param archivoControl archivo que registra el último backup
 * @return número de archivos copiados
 * @throws IOException si hay error en el proceso
 */
public static int backupIncremental(String carpetaOrigen, String carpetaDestino, String archivoControl) throws IOException

/**
 * Lee la fecha del último backup desde el archivo de control
 * @param archivoControl ruta del archivo de control
 * @return timestamp del último backup, o 0 si no existe
 * @throws IOException si hay error de lectura
 */
private static long leerUltimoBackup(String archivoControl) throws IOException

/**
 * Copia un archivo de origen a destino
 * @param origen archivo fuente
 * @param destino archivo destino
 * @throws IOException si hay error en la copia
 */
private static void copiarArchivo(File origen, File destino) throws IOException
```

Ejemplo de uso:

Código:

```
int archivosCopiados = backupIncremental(
    "./documentos",
    "./backup",
    "./backup/.lastbackup"
);
System.out.println("Backup completado: " + archivosCopiados + " archivos");
```

Salida por consola (*primera ejecución*):

```
Iniciando backup incremental...
Último backup: nunca
Copiando: documento1.txt
Copiando: documento2.txt
Copiando: imagen.png
Backup completado: 3 archivos
Registro actualizado: 2025-10-14 10:30:00
```

Salida por consola (*segunda ejecución, solo 1 archivo modificado*):

```
Iniciando backup incremental...
Último backup: 2025-10-14 10:30:00
Copiando: documento2.txt (modificado)
Backup completado: 1 archivo
Registro actualizado: 2025-10-14 11:45:00
```

Resultado esperado: Sistema que optimiza backups copiando solo archivos nuevos o modificados, con registro persistente del último backup realizado.

FORMATO DE ENTREGA

Requisitos del código:

- JavaDoc en todas las clases y métodos públicos
- Comentarios explicativos en lógica compleja
- Nombres descriptivos de variables
- Manejo apropiado de excepciones
- Uso de try-with-resources
- Encoding UTF-8 explícito donde sea posible
- Código compilable y ejecutable sin errores