

# Unidad 1: Manejo de ficheros.

## Índice de contenidos

1	Clasificación de los ficheros.....	2
2	Propiedades del sistema. ....	2
3	Gestión de ficheros y directorios. ....	4
3.1	Clase File. ....	4
3.2	Clases Path y Paths. ....	6
3.3	Clase Files. ....	7
4	Flujos. ....	9
5	Ficheros de texto. ....	10
5.1	Escribir en ficheros de texto. ....	11
5.1.1	Carácter a carácter. ....	11
5.1.2	Cadena de caracteres. ....	12
5.1.3	Escritura de ficheros de texto con formato. ....	14
5.2	Leer de ficheros de texto. ....	17
5.2.1	Carácter a carácter. ....	17
5.2.2	Línea a línea. ....	18
5.2.3	Leer diferentes tipos de datos. ....	19
6	Ficheros binarios. ....	21
6.1	Escribir en ficheros binarios. ....	21
6.1.1	Byte a byte. ....	21
6.1.2	Byte a byte con búfer. ....	22
6.1.3	Datos primitivos y tipos de datos estructurados. ....	23
6.2	Leer de ficheros binarios. ....	24
6.2.1	Byte a byte. ....	24
6.2.2	Byte a byte con búfer. ....	25
6.2.3	Datos primitivos y tipos de datos estructurados. ....	27
7	Ficheros de configuración. ....	28
8	Acceso directo a ficheros. ....	30
9	Persistencia de objetos. ....	38
9.1	Serialización. ....	39
9.1.1	El <code>serialVersionUID</code> . ....	39
9.2	Flujos para entrada y salida de objetos. ....	39
9.3	El modificador <code>transient</code> . ....	43
9.4	La herencia en objetos serializables. ....	44
10	Ficheros XML. ....	47
10.1	¿Qué es XML? ....	47
10.1.1	Usos de XML. ....	47
10.1.2	Ficheros XML. ....	47
10.2	Acceso a ficheros XML con SAX. ....	48
10.3	Acceso a ficheros XML con DOM. ....	52
10.4	Serialización de objetos a XML. ....	54
11	Tratamiento de documentos JSON. ....	57
12	Convertir un archivo XML a JSON. ....	62
13	Control de excepciones. ....	63
13.1	Excepciones en Java. ....	63
13.2	Manejo de excepciones comprobadas. ....	63
13.3	Obtener mensaje de la excepción. ....	66
13.4	Uso obligatorio de try-catch. ....	66
13.5	Jerarquía de clases de excepciones. ....	67
13.6	Excepciones más frecuentes. ....	67
14	Depuración y documentación. ....	69
14.1	Depuración de programas con NetBeans. ....	70
14.2	Documentación. ....	71

# 1 Clasificación de los ficheros.

Los **ficheros**, también conocidos como archivos, **son fundamentales** para que las **aplicaciones puedan almacenar, recuperar y manipular datos** de manera **no volátil**. Cada fichero **tiene un nombre que lo identifica** y, en muchos casos, **una extensión** (como ".txt", ".jpg" o ".mp3") **que indica el tipo de contenido** que alberga. A diferencia de los datos en memoria, los ficheros **preservan la información incluso después de apagar el ordenador o cerrar un programa**.

La **clasificación de los ficheros** puede realizarse según diferentes criterios:

- **Según el tipo de datos almacenados:**
  - ✓ **Ficheros binarios:** contienen datos en formato binario, que no son legibles directamente por los humanos. Este tipo de fichero se utiliza comúnmente para almacenar imágenes, vídeos, archivos ejecutables, etc. Las **extensiones** como **.bin** o **.dat** suelen indicar que el fichero contiene información binaria en un formato específico, que una aplicación particular puede leer o escribir de manera especial.
  - ✓ **Ficheros de caracteres** (de texto): almacenan datos en formato de texto, lo que permite que sean leídos por humanos. Son ideales para guardar texto plano, configuraciones o registros.
- **Según el tipo de acceso:**
  - ✓ **Acceso secuencial:** los datos se leen o escriben en un orden secuencial, desde el principio hasta el final del fichero. Este método es eficiente para manejar grandes volúmenes de datos en una sola pasada.
  - ✓ **Acceso directo o aleatorio:** permite acceder a datos en cualquier posición del fichero sin necesidad de procesar el contenido secuencialmente. Es útil cuando se necesita localizar registros específicos en ficheros grandes.
- **Según la forma de apertura:**
  - ✓ **Apertura simple:**
    - **Lectura:** el fichero se abre únicamente para leer datos.
    - **Escritura:** el fichero se abre exclusivamente para escribir datos.
    - **Lectura/Escritura:** el fichero se abre para realizar tanto operaciones de lectura como de escritura.
  - ✓ **Apertura con buffering:** utiliza clases que proporcionan un búfer para leer o escribir datos en bloques, lo cual puede mejorar el rendimiento al reducir la cantidad de operaciones de entrada/salida.
- **Según el tipo de fichero:**
  - ✓ **Ficheros regulares:** almacenan datos generales y son los más comunes.
  - ✓ **Directorios:** contienen una lista de nombres de ficheros y subdirectorios, actuando como contenedores de otros ficheros.
  - ✓ **Enlaces simbólicos:** son referencias a otros ficheros o directorios en el sistema de archivos.
  - ✓ **Dispositivos especiales:** representan dispositivos de hardware, como discos duros, impresoras, etc.
- **Según el tipo de flujo de datos:**
  - ✓ **Flujo de entrada:** utilizado para leer datos desde un fichero.
  - ✓ **Flujo de salida:** utilizado para escribir datos en un fichero.

La elección del tipo de fichero adecuado dependerá de las necesidades específicas de cada aplicación.

# 2 Propiedades del sistema.

Las propiedades del sistema en **Java** **proporcionan información sobre la configuración y el entorno en el que se ejecuta una aplicación**. Estas propiedades son útiles para manejar **ficheros y directorios de manera que el código sea compatible con diferentes sistemas operativos**. java.lang.Object  
java.lang.System

En **Java**, las propiedades del sistema **se pueden acceder a través de la clase System** (en el paquete `java.lang`) **mediante el método estático getProperty(String key)**. A continuación, se describen algunas de las propiedades del sistema más relevantes para la gestión de ficheros y directorios:

- ✓ **file.separator:** proporciona el **separador de directorios del sistema de archivos**. En sistemas Windows, es "\", mientras que en sistemas Unix/Linux, es "/". También se puede utilizar la **constante File.separator** (importando la clase `java.io.File`) o el método estático `FileSystems.getDefault().getSeparator()`.

```
String separador = System.getProperty("file.separator"); // También File.separator
System.out.println("Separador de directorios: " + separador);
```

```
String directorio = "mi_carpeta", nombreFichero = "mi_archivo.txt";
String rutaCompleta = directorio + separador + nombreFichero;
System.out.println("Ruta de acceso al fichero: " + rutaCompleta);
```

- ✓ **file.encoding**: indica la **codificación de caracteres utilizada por el sistema**, lo cual es crucial al leer o escribir ficheros de texto.

```
String fileEncoding = System.getProperty("file.encoding");
System.out.println("Codificación de ficheros: " + fileEncoding);
```

- ✓ **user.home**: proporciona la **ruta al directorio principal del usuario actual**. Es útil para acceder a directorios personales o para establecer rutas de acceso a ficheros específicos en el directorio del usuario.

```
String userHome = System.getProperty("user.home");
System.out.println("Directorio de inicio del usuario: " + userHome);
```

- ✓ **user.dir**: indica el **directorio de trabajo actual de la aplicación Java**. Es útil para construir rutas de acceso relativas desde el directorio en el que se ejecuta la aplicación.

```
String userDir = System.getProperty("user.dir");
System.out.println("Directorio de trabajo actual: " + userDir);
```

- ✓ **java.io.tmpdir**: proporciona la **ubicación del directorio temporal del sistema**, ideal para crear ficheros temporales.

```
String tempDir = System.getProperty("java.io.tmpdir");
System.out.println("Directorio temporal del sistema: " + tempDir);
```

- ✓ **line.separator**: se utiliza para **obtener el carácter o secuencia de caracteres que representa un salto de línea** en el sistema operativo. En sistemas *Windows*, es "\r\n" (retorno de carro y nueva línea), mientras que en sistemas *Unix/Linux* es "\n" (un solo carácter de nueva línea). También se puede utilizar el método **System.lineSeparator()**.

```
String newLine = System.getProperty("line.separator"); // También System.lineSeparator()
String textoConSaltoDeLinea = "Línea 1" + newLine + "Línea 2";
```

Estas propiedades son **especialmente útiles para escribir código que sea compatible con múltiples sistemas operativos**.

#### Ejemplo:

```
import java.io.*;
import java.nio.file.*;

public class PropiedadesDelSistema {
    public static void main(String[] args) {
        // Obtener y mostrar algunas propiedades del sistema
        String fileSeparator = FileSystems.getDefault().getSeparator();
        String userHome = System.getProperty("user.home");
        String userDir = System.getProperty("user.dir");
        String lineSeparator = System.lineSeparator();

        System.out.println("Separador de directorios: " + fileSeparator);
        System.out.println("Directorio de inicio del usuario: " + userHome);
        System.out.println("Directorio de trabajo actual: " + userDir);
        System.out.println("Separador de línea: " + lineSeparator.replace("\n", "\\n").replace("\r", "\\r"));

        // Crear una ruta de acceso al directorio de inicio del usuario
        String filePath = userHome + fileSeparator + "miarchivo.txt";
        // Crear y escribir en un fichero con el formato adecuado para el sistema
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
            writer.write("Este es un ejemplo de línea 1" + lineSeparator);
            writer.write("Este es un ejemplo de línea 2" + lineSeparator);
            writer.write("Este es un ejemplo de línea 3" + lineSeparator);
            System.out.println("Datos escritos en el fichero: " + filePath);
        } catch (IOException e) {
            System.err.println("Error al escribir en el fichero: " + e.getMessage());
        }

        // Leer el fichero y mostrar su contenido
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String linea;
            System.out.println("Contenido del fichero:");
            while ((linea = reader.readLine()) != null)
                System.out.println(linea);
        } catch (IOException e) {
            System.err.println("Error al leer el fichero: " + e.getMessage());
        }
    }
}
```

La propiedad que permite obtener el nombre del Sistema Operativo con el que se está trabajando es **os.name**

```
}
}
```

## 3 Gestión de ficheros y directorios.

### 3.1 Clase File.

La clase `File`, presente en *Java* desde versiones anteriores a *Java 7*, forma parte del **paquete `java.io`** y se utiliza para representar de manera abstracta rutas de archivos y directorios. Permite realizar diversas operaciones sobre ellos, tales como crear, eliminar, renombrar y verificar su existencia. Es importante destacar que **esta clase no maneja el contenido de los archivos**.

A continuación, se presenta una tabla con los **constructores y algunos de los métodos** más importantes:

Constructor/Método	Descripción
<code>File(String path)</code>	Crea una instancia de <code>File</code> utilizando la ruta especificada.
<code>File(String path, String name)</code>	Crea una instancia de <code>File</code> utilizando la ruta y el nombre de fichero.
<code>File(File dir, String name)</code>	Crea una instancia de <code>File</code> utilizando un objeto <code>File</code> y un nombre de fichero.
<code>boolean canRead()</code>	Verifica si el fichero o directorio se puede leer. Devuelve <code>true</code> si es legible.
<code>boolean canWrite()</code>	Verifica si el fichero o directorio se puede escribir. Devuelve <code>true</code> si es escribible.
<code>boolean canExecute()</code>	Verifica si el fichero o directorio se puede ejecutar.
<code>boolean createNewFile()</code>	Crea un nuevo fichero vacío en la ruta especificada. Devuelve <code>true</code> si el fichero se creó.
<code>boolean delete()</code>	Elimina el fichero o directorio asociado a <code>File</code> . Devuelve <code>true</code> si se eliminó correctamente.
<code>boolean exists()</code>	Verifica si el fichero o directorio existe. Devuelve <code>true</code> si existe.
<code>String getAbsolutePath()</code>	Obtiene la ruta absoluta del fichero o directorio.
<code>String getName()</code>	Obtiene el nombre del fichero o directorio.
<code>String getParent()</code>	Obtiene la ruta del directorio padre, o <code>null</code> si no existe.
<code>String getPath()</code>	Obtiene la ruta relativa del fichero o directorio.
<code>boolean isDirectory()</code>	Verifica si el objeto <code>File</code> representa un directorio. Devuelve <code>true</code> si es un directorio.
<code>boolean isFile()</code>	Verifica si el objeto <code>File</code> representa un fichero. Devuelve <code>true</code> si es un fichero.
<code>long lastModified()</code>	Devuelve la hora en que se modificó por última vez el archivo.
<code>long length()</code>	Obtiene el tamaño del fichero en bytes.
<code>String[] list()</code>	Obtiene un array de nombres de ficheros/directorios en el directorio representado.
<code>String[] list(FilenameFilter filter)</code>	Obtiene un array de nombres de ficheros/directorios en el directorio representado que satisface el filtro.
<code>File[] listFiles()</code>	Obtiene un array de objetos <code>File</code> para los ficheros/directorios en el directorio representado.
<code>File[] listFiles(FileFilter filter)</code>	Obtiene un array de objetos <code>File</code> para los ficheros/directorios en el directorio representado que satisface el filtro.
<code>boolean mkdir()</code>	Crea el directorio representado por el objeto <code>File</code> (nombre indicado al crearlo). Devuelve <code>true</code> si se creó.
<code>boolean mkdirs()</code>	Crea el directorio representado por el objeto <code>File</code> , creando también los directorios padres si no existen. Devuelve <code>true</code> si se creó.
<code>boolean renameTo(File dest)</code>	Cambia el nombre de fichero o mueve el fichero/directorio representado a la ubicación especificada por <code>dest</code> . Devuelve <code>true</code> si se renombró/movió correctamente.
<code>boolean setLastModified(long time)</code>	Establece la marca de tiempo del último acceso/modificación del fichero. Devuelve <code>true</code> si se estableció correctamente.
<code>boolean setReadable(boolean readable)</code>	Establece los permisos de lectura del fichero/directorio. Devuelve <code>true</code> si se estableció correctamente.

<code>boolean setWritable(boolean writable)</code>	Establece los permisos de escritura del fichero/directorio. Devuelve <code>true</code> si se estableció correctamente.
<code>boolean setExecutable(boolean executable)</code>	Establece los permisos de ejecución del fichero/directorio. Devuelve <code>true</code> si se estableció correctamente.
<code>Path toPath()</code>	Obtener un objeto <code>Path</code> que representa la ubicación del archivo.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/File.html>

A partir de *Java 7*, se introdujeron **clases** y métodos más modernos para trabajar con el sistema de archivos, como **Path**, **Paths** y **Files** (en el paquete `java.nio.file`). Estas clases ofrecen una *API* más completa y flexible para operaciones con archivos y directorios.

**Ejemplo:** `import java.io.File;`  
`import java.io.IOException;`

```
public class FileEjemplo {
    public static void main(String[] args) {
        // Crear una instancia de File para representar un fichero
        File archivo = new File("archivo.txt"); // Windows C:\\directorio\\nombre_fichero.ext
                                                // Linux /directorio/nombre_fichero.ext

        // Verificar si el archivo existe
        if (archivo.exists())
            System.out.println("El archivo existe.");
        else
            System.out.println("El archivo no existe.");

        // Crear un nuevo archivo
        try {
            boolean creado = archivo.createNewFile();
            if (creado)
                System.out.println("Archivo creado correctamente.");
            else
                System.out.println("El archivo ya existe o no se pudo crear.");
        } catch (IOException e) {
            System.out.println("Ocurrió un error al crear el archivo: " + e.getMessage());
        }

        // Obtener información sobre el archivo
        System.out.println("Nombre del archivo: " + archivo.getName());
        System.out.println("Ruta absoluta: " + archivo.getAbsolutePath());
        System.out.println("Tamaño del archivo: " + archivo.length() + " bytes");

        // Eliminar el archivo
        boolean eliminado = archivo.delete();
        if (eliminado)
            System.out.println("Archivo eliminado.");
        else
            System.out.println("No se pudo eliminar el archivo o el archivo no existe.");
    }
}
```

Varias formas de crear objetos File

**Ejemplo:** obtener la lista de ficheros/directorios del directorio de trabajo actual.

```
import java.io.File;

public class ListarArchivosEjemplo {
    public static void main(String[] args) {
        // Obtener el directorio actual
        String directorioActual = System.getProperty("user.dir");
        System.out.println("Directorio de trabajo actual: " + directorioActual);

        // Crear una instancia de File para el directorio actual
        File directorio = new File(directorioActual);

        // Verificar si el objeto File representa un directorio
        if (directorio.isDirectory()) {
            // Obtener la lista de archivos y directorios en el directorio actual
            String[] listaArchivos = directorio.list();
        }
    }
}
```

`File f = new File("d:\\db");`

Directorio d:\\db

```

// Mostrar la lista de archivos y directorios
if (listaArchivos != null && listaArchivos.length > 0) {
    System.out.println("Archivos y directorios en el directorio de trabajo actual:");
    for (String archivo : listaArchivos)
        System.out.println(" - " + archivo);
} else
    System.out.println("El directorio está vacío.");
} else
    System.out.println("El path especificado no es un directorio válido.");
}
}

```

**Ejemplo:** filtro de archivos.

```

import java.io.File;
import java.io.FileFilter;

public class ListFilesConFiltro {
    public static void main(String[] args) {
        String directorio = "c:\\java"; // Directorio en el que se desea buscar archivos

        // Crea un objeto FileFilter personalizado usando una expresión lambda
        FileFilter filtro = (dir, nombreArchivo) -> nombreArchivo.endsWith(".java");
        /* --- OTRA FORMA DE HACERLO SIN UTILIZAR EXPRESIONES LAMBDA ---
        FileFilter filtro = new FileFilter() {
            @Override
            public boolean accept(File directorio, String nombreArchivo) {
                return nombreArchivo.endsWith(".java");
            }
        };
        */

        // Obtener una lista de archivos que cumplen con el criterio definido en el filtro
        File dir = new File(directorio);
        String[] archivosFiltrados = dir.list(filtro);

        // Imprimir los nombres de los archivos que cumplen con el criterio
        if (archivosFiltrados != null && archivosFiltrados.length > 0) {
            System.out.println("Archivos con extensión .java en el directorio " + directorio + ":");
            for (String nombreArchivo : archivosFiltrados)
                System.out.println(" - " + nombreArchivo);
        } else {
            System.out.println("No se encontraron archivos que cumplan con el criterio.");
        }
    }
}

```

String dir = args[0];  
System.out.println("Archivos en el directorio: " + dir);  
File f = new File(dir);

Directorio introducido desde la línea de comandos

En el código, el filtro está definido de tal manera que, si el nombre del archivo termina con una determinada extensión, devuelve true; de lo contrario, devuelve false. La función `endsWith()` de la clase `String` realiza esta tarea, como se muestra en la porción de código.

## 3.2 Clases Path y Paths.

Las **clases Path y Paths** forman parte de la API de manejo de archivos introducida en *Java 7*, y se encuentran en el **paquete java.nio.file**:

- ✓ La **clase Path** representa una ubicación en el sistema de archivos, como una ruta de acceso a un archivo o directorio. **Se utiliza para representar y manipular rutas de acceso de manera eficiente y portátil en diferentes sistemas operativos.**
- ✓ La **clase Paths** proporciona **métodos estáticos para crear instancias de objetos Path**. Estos **objetos Path pueden luego ser utilizados con la clase Files** y otros métodos de la API de manejo de archivos para realizar operaciones como lectura, escritura, copia, movimiento y eliminación de archivos.

En la clase `Path` de *Java*, no hay constructores públicos accesibles directamente. La clase `Path` se diseñó para que sus instancias sean creadas a través de métodos estáticos de la clase `Paths`, como `Paths.get()` o mediante métodos de la clase `FileSystems`, como `FileSystems.getDefault().getPath()`.

**Métodos importantes de la clase Path:**

Método	Descripción
<code>static Path of(String first, String... more)</code>	Crea un objeto <code>Path</code> a partir de una cadena de rutas.
<code>boolean isAbsolute()</code>	Comprueba si la ruta es absoluta (completa) o relativa.



<code>Path getFileName()</code>	Devuelve el nombre del fichero o directorio al final de la ruta.
<code>Path getParent()</code>	Devuelve el directorio padre de la ruta.
<code>Path getRoot()</code>	Devuelve la raíz de la ruta (si es absoluta).
<code>int getNameCount()</code>	Devuelve el número de elementos en la ruta.
<code>Path getName(int index)</code>	Devuelve el elemento en la posición especificada de la ruta.
<code>Path subpath(int beginIndex, int endIndex)</code>	Devuelve una subruta que comienza en <code>beginIndex</code> (incluido) y termina en <code>endIndex</code> (sin incluir).
<code>boolean startsWith(Path other)</code>	Comprueba si la ruta comienza con otra ruta dada.
<code>boolean endsWith(Path other)</code>	Comprueba si la ruta termina con otra ruta dada.
<code>Path resolve(Path other)</code>	Combina la ruta actual con otra ruta para crear una nueva ruta.
<code>Path normalize()</code>	Normaliza la ruta eliminando componentes redundantes.
<code>File toFile()</code>	Convierte la ruta a un objeto <code>File</code> .
<code>byte[] toByteArray()</code>	Lee el contenido del fichero referenciado como un array de bytes.

**Métodos importantes de la clase `Paths`:**

Método	Descripción
<code>static Path get(String first, String...)</code>	Crea un objeto <code>Path</code> a partir de una cadena de rutas.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/file/Path.html> y <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/file/Paths.html>

**Ejemplo:**

```
import java.nio.file.*; // Importar las clases Path y Paths

public class PathEjemplo {
    public static void main(String[] args) {
        // Crear una ruta usando Paths.get()
        Path path = Paths.get("C:", "Users", "Usuario", "Documents", "archivo.txt");

        // Mostrar información sobre la ruta
        System.out.println("Ruta completa: " + path); // path.toString()
        System.out.println("Nombre del archivo: " + path.getFileName());
        System.out.println("Directorio padre: " + path.getParent());
        System.out.println("Número de elementos en la ruta: " + path.getNameCount());

        // Iterar a través de los componentes de la ruta
        System.out.println("Componentes de la ruta:");
        for (int i = 0; i < path.getNameCount(); i++)
            System.out.println("  - " + (i + 1) + ": " + path.getName(i));

        // Comprobar si la ruta es absoluta
        System.out.println("¿Ruta absoluta? " + (path.isAbsolute() ? "SI" : "NO"));

        // Resolviendo una ruta relativa
        Path relativePath = Paths.get("subdirectorio", "archivo2.txt");
        Path resolvedPath = path.resolve(relativePath);
        System.out.println("Ruta resuelta: " + resolvedPath);

        // Comprobar si una ruta comienza con otra ruta
        boolean startsWith = path.startsWith(Paths.get("C:", "Users"));
        System.out.println("¿La ruta comienza con 'C:\\\\Users'? " + (startsWith ? "SI" : "NO"));
    }
}
```

### 3.3 Clase `Files`.

La clase `Files` fue introducida en *Java 7* como parte del paquete `java.nio.file`. Ofrece una serie de **métodos estáticos que permiten realizar diversas operaciones con ficheros y directorios de manera más versátil y eficiente** en comparación con la clase `File`.

A continuación, se presenta una tabla con algunos de los **métodos** más importantes:

Método	Descripción
<code>static Path createFile(Path path, FileAttribute&lt;?&gt;... attrs)</code>	Crea un nuevo fichero en la ubicación especificada.

<code>static Path copy(Path source, Path target, CopyOption... options)</code>	Copia un fichero o directorio desde la ruta de origen a la ruta de destino.
<code>static Path move(Path source, Path target, CopyOption... options)</code>	Mueve un fichero o directorio desde la ruta de origen a la ruta de destino.
<code>static byte[] readAllBytes(Path path)</code>	Lee todos los bytes de un fichero y los devuelve como un array de bytes.
<code>static List&lt;String&gt; readAllLines(Path path, Charset cs)</code>	Lee todas las líneas de texto de un fichero y las devuelve como una lista de cadenas de texto.
<code>static Path write(Path path, byte[] bytes, OpenOption... options)</code>	Escribe un array de bytes en un fichero.
<code>static Path write(Path path, Iterable&lt;? extends CharSequence&gt; lines, Charset cs, OpenOption... options)</code>	Escribe una lista de líneas de texto en un fichero.
<code>static Path writeString(Path path, CharSequence text, OpenOption... options)</code>	Escribe una secuencia de caracteres en un fichero.
<code>static boolean isRegularFile(Path path, LinkOption... options)</code>	Comprueba si el camino especificado representa un fichero.
<code>static boolean isDirectory(Path path, LinkOption... options)</code>	Comprueba si la ruta especificada representa un directorio.
<code>static boolean exists(Path path, LinkOption... options)</code>	Comprueba si la ruta especificada existe en el sistema de archivos.
<code>static void delete(Path path)</code>	Elimina el fichero o directorio asociado a Path.
<code>static boolean deleteIfExists(Path path)</code>	Elimina el fichero o directorio asociado a Path si existe.
<code>static boolean isReadable(Path path)</code>	Comprueba si el fichero o directorio es legible.
<code>static boolean isWritable(Path path)</code>	Comprueba si el fichero o directorio es escribible.
<code>static boolean isExecutable(Path path)</code>	Comprueba si el fichero es ejecutable.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/file/Files.html>

```

Ejemplo: import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;

public class FilesEjemplo {
    public static void main(String[] args) {
        // Definir las rutas de acceso de origen y destino
        Path origen = Paths.get("origen.txt");
        Path destino = Paths.get("destino.txt");
        Path nuevoDestino = Paths.get("otro_destino.txt");

        try {
            // Crear un nuevo fichero en el directorio actual
            if (!Files.exists(origen)) {
                Files.createFile(origen);
                System.out.println("Fichero de origen creado.");
            }

            // Escribir datos en el fichero
            String contenido = "Este es el contenido del fichero.";
            Files.writeString(origen, contenido);
            System.out.println("Datos escritos en el fichero de origen.");

            // Leer los bytes del fichero
            String contenidoLeido = Files.readString(origen, StandardCharsets.UTF_8);
            System.out.println("Contenido leído del fichero: " + contenidoLeido);

            // Copiar el fichero al directorio de destino
            Files.copy(origen, destino, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("Fichero copiado con éxito.");

            // Mover el fichero a otro directorio
            Files.move(destino, nuevoDestino, StandardCopyOption.REPLACE_EXISTING);
            System.out.println("Fichero movido con éxito.");

            // Comprobar si el fichero de origen es un fichero regular
            boolean esFichero = Files.isRegularFile(origen, LinkOption.NOFOLLOW_LINKS);
            System.out.println("¿El fichero de origen es un fichero regular? " + esFichero);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```



```

// Comprobar si el nuevo destino es un fichero
boolean esDirectorio = Files.isDirectory(nuevoDestino, LinkOption.NOFOLLOW_LINKS);
System.out.println("¿El nuevo destino es un directorio? " + esDirectorio);

// Eliminar el fichero de origen y el nuevo destino
Files.deleteIfExists(origen);
Files.deleteIfExists(nuevoDestino);
System.out.println("Fichero de origen y nuevo destino eliminados.");
} catch (IOException e) {
    System.err.println("Se produjo un error: " + e.getMessage());
}
}
}

```



## TAREA

1. Crear un programa que muestre información detallada sobre un fichero o directorio: nombre, ruta completa, tipo (fichero o directorio), tamaño (caso de ser un fichero) y permisos de lectura/escritura/ejecución.
2. Crear un programa que copie un fichero de una ubicación a otra. El programa debe solicitar al usuario que introduzca la ruta y nombre del fichero de origen, y la ruta y nombre del fichero de destino. Verificar si el fichero de origen existe antes de proceder con la copia.
3. (Opcional) Crear una función que busque ficheros con una extensión específica dentro de un directorio y sus subdirectorios, mostrando la ruta completa de los ficheros encontrados. Hacer uso de la recursividad.

## 4 Flujos.

En *Java*, un "flujo" se refiere a la transferencia secuencial de datos entre una fuente y un destino. Los flujos son esenciales para las operaciones de entrada y salida de datos y se utilizan para leer y escribir información en distintos contextos. Hay dos tipos principales de flujos según su propósito:



- ✓ **Flujos de entrada (Input Streams):** se utilizan para leer datos de una fuente, como un archivo, el teclado o un socket de red. Proporcionan métodos para capturar datos y transferirlos al programa.
- ✓ **Flujos de salida (Output Streams):** se utilizan para escribir datos desde el programa hacia un destino, como un archivo, la pantalla o un socket de red. Ofrecen métodos para enviar los datos desde el programa a su destino.

**Categorías principales de flujos según la representación de la información:**

- ✓ **Flujos de bytes (Byte Streams):** diseñados para manejar datos binarios. Permiten leer y escribir bytes individuales sin tener en cuenta la codificación de caracteres.
- ✓ **Flujos de caracteres (Character Streams):** utilizados para manejar datos de texto, tienen en cuenta la codificación de caracteres, como UTF-8 o ASCII, y son más adecuados para trabajar con texto.

Característica	Flujo de Bytes	Flujo de Caracteres
Representación de datos	Maneja datos binarios y sin procesar.	Trabaja con datos de texto.
Tamaño del dato	Se mide en bytes (8 bits).	Se mide en caracteres (16 bits).
Codificación de caracteres	No interpreta ni gestiona la codificación.	Interpreta y gestiona la codificación de caracteres.
Uso común	Para ficheros binarios (imágenes, ejecutables).	Para ficheros de texto y datos textuales.
Conversión de codificación	No realiza conversiones automáticas.	Puede realizar conversiones automáticas según la codificación.

La elección entre flujos de bytes y flujos de caracteres depende del tipo de datos con el que se trabaja.

**Enfoques para leer y escribir datos en ficheros según el acceso:**

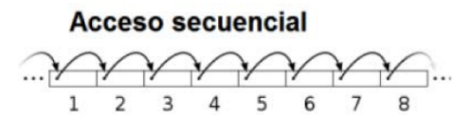
- ✓ **Acceso secuencial:** los datos se leen o escriben de manera secuencial, es decir, en orden, uno después del otro. Para acceder a un dato específico, primero se deben leer todos los datos anteriores. No permite inserciones en posiciones intermedias.

- ✓ **Acceso directo o aleatorio:** permite leer o escribir datos en cualquier posición del fichero sin necesidad de pasar por todos los datos previos.

### Flujos comunes para trabajar con ficheros en Java:

#### ➤ Acceso secuencial:

- Flujos de **entrada** (*Input Streams*):
  - Basados en **caracteres**:
    - **FileReader**: lee caracteres de un fichero de texto.
    - **BufferedReader**: proporciona un búfer para la lectura más eficiente de ficheros de texto.
    - **Scanner**: analiza y extrae diferentes tipos de datos (enteros, flotantes, cadenas, etc.).
  - Basados en **bytes**:
    - **FileInputStream**: lee bytes de un fichero, ideal para ficheros binarios.
    - **BufferedInputStream**: ofrece un búfer para la lectura eficiente de datos binarios.
    - **DataInputStream**: lee datos primitivos (int, float, char, boolean, etc.) en formato binario.
    - **ObjectInputStream**: lee objetos serializados desde un fichero.
- Flujos de **salida** (*Output Streams*):
  - Basados en **caracteres**:
    - **FileWriter**: escribe caracteres en un fichero de texto.
    - **BufferedWriter**: proporciona un búfer para la escritura más eficiente de texto.
    - **PrintWriter**: escribe texto con un formato específico.
  - Basados en **bytes**:
    - **FileOutputStream**: escribe bytes en un fichero, adecuado para ficheros binarios.
    - **BufferedOutputStream**: ofrece un búfer para la escritura eficiente de datos binarios.
    - **DataOutputStream**: escribe datos primitivos en formato binario.
    - **ObjectOutputStream**: escribe objetos serializados en un fichero.



#### ➤ Acceso directo o aleatorio:

- **RandomAccessFile**: permite operaciones de lectura y escritura de acceso aleatorio en ficheros de texto o binarios. Esta clase permite abrir archivos en distintos modos de acceso (*r*, *w*, *rw*, ...) y trabajar tanto con ficheros binarios como de caracteres.

## 5 Ficheros de texto.

Los ficheros de texto **contienen información legible por humanos y se almacenan en formato texto** (sin formato), generalmente en forma de caracteres ASCII o Unicode. Estos archivos **son utilizados para almacenar datos simples y estructurados**, como configuraciones, registros, código fuente, documentos, etc. Pueden ser editados directamente con cualquier editor de texto plano, como *Bloc de notas*, *Notepad++*, etc.

Una de las principales ventajas de los ficheros de texto es su **portabilidad entre distintos lenguajes de programación, sistemas operativos y arquitecturas de hardware**. Sin embargo, **tienden a requerir más espacio en disco en comparación con los archivos binarios**, ya que almacenan la información como texto legible.

El bloque **try-with-resources** es una característica introducida en *Java 7* que **permite manejar automáticamente el cierre de recursos**, como archivos, conexiones a *BD* o flujos de datos, de manera segura y eficiente. Este bloque **elimina la necesidad de escribir explícitamente un bloque finally para cerrar los recursos después de su uso**, reduciendo la posibilidad de errores y mejorando la legibilidad del código.

Su **estructura básica** es:

```
try (/* Declaración de recursos */) {
    // Código que utiliza los recursos
} catch (/* Excepciones */) {
    // Manejo de excepciones
}
```

Los **recursos que se declaran dentro del paréntesis del try deben implementar la interfaz AutoCloseable** (o su subinterfaz *Closeable*). Esto incluye muchos tipos de clases utilizadas para el manejo de recursos, como *FileReader*, *FileWriter*, *InputStream*, *OutputStream*, *Connection*, etc.

El uso de **try-with-resources** no solo facilita la gestión de recursos, sino que también **minimiza el riesgo de fugas de recursos**, asegurando que se liberen correctamente una vez que el bloque de código haya terminado de ejecutarse, incluso si se produce una excepción.

## 5.1 Escribir en ficheros de texto.

### 5.1.1 Carácter a carácter.

```
java.lang.Object
java.io.Writer
java.io.OutputStreamWriter
java.io.FileWriter
```

La grabación de caracteres en un fichero de texto se puede realizar utilizando la clase `FileWriter` del paquete `java.io`. El constructor de esta clase se encarga de abrir el fichero, y requiere que se indique como parámetro el fichero que se va a utilizar. Este parámetro puede ser un `String` que especifique la ruta y el nombre del fichero, o bien un objeto `File` que represente el fichero.

Si el fichero no existe, se crea automáticamente. Es importante tener en cuenta que, si el fichero ya existía, su contenido previo será borrado. Para evitar que el contenido existente se elimine cada vez que se abra el fichero, se puede pasar un segundo parámetro de tipo boolean al constructor que indica si los datos deben añadirse al final del archivo existente:

```
fw = new FileWriter(nombreFichero, true);
```

La operación de escritura se realiza utilizando el método `write`, que permite escribir caracteres en el fichero de texto. Los métodos `write(...)` pueden generar una excepción `IOException` (por ejemplo, en casos de disco lleno, permisos insuficientes, etc.).

Al finalizar las operaciones sobre el fichero, se debe cerrar el flujo para asegurar que todos los datos se guarden correctamente en el dispositivo de almacenamiento (disco duro, SSD, etc.). Esto se logra usando el método `close`. Es recomendable utilizar un bloque `finally` para asegurarse de que el flujo se cierre, incluso en el caso de excepciones, o bien emplear un bloque `try-with-resources`.

En algunos casos, es útil asegurar que los datos se guarden en el dispositivo de almacenamiento antes de cerrar el fichero. Para ello, se puede utilizar el método `flush`, que fuerza el volcado de los datos sin cerrar el fichero.

Tabla con los constructores y métodos importantes de la clase `FileWriter`:

Constructor/Método	Descripción
<code>FileWriter(File file)</code>	Crea un objeto <code>FileWriter</code> para escribir en el archivo especificado.
<code>FileWriter(String path)</code>	Crea un objeto <code>FileWriter</code> para escribir en el archivo con la ruta especificada.
<code>FileWriter(File file, boolean append)</code>	Crea un objeto <code>FileWriter</code> que permite agregar contenido al archivo si <code>append</code> es <code>true</code> . Si el modo <code>append</code> está establecido en <code>true</code> , <code>write()</code> no sobrescribe los datos existentes en el archivo y los nuevos caracteres escritos se agregan al final del mismo.
<code>FileWriter(String path, boolean append)</code>	Crea un objeto <code>FileWriter</code> que permite agregar contenido al archivo con la ruta especificada y según el valor de <code>append</code> .
<code>void write(int c)</code>	Escribe el carácter especificado en el archivo.
<code>void write(char[] cbuf)</code>	Escribe un array de caracteres.
<code>void write(String str)</code>	Escribe una cadena de caracteres.
<code>void write(String str, int off, int len)</code>	Escribe una porción de la cadena en el archivo, comenzando en el índice <code>off</code> y escribiendo <code>len</code> caracteres.
<code>Writer append(char c)</code>	Añade un carácter al final del archivo.
<code>void flush()</code>	Limpia el búfer y asegura que todos los datos se escriban en el archivo.
<code>void close()</code>	Cierra el flujo, liberando los recursos asociados.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/FileWriter.html>

```
Ejemplo: import java.io.FileWriter;
import java.io.IOException;

public class FileWriter1 {
    public static void main(String[] args) {
        String fileName = "archivo.txt"; // Definir el nombre del archivo
        // Uso de try-with-resources para manejo automático de recursos
        try (FileWriter writer = new FileWriter(fileName)) { // Crear el flujo de salida
            // Escribir caracteres usando varios métodos
            writer.write('H');
            writer.write("ola, ");
        }
    }
}
```

```

writer.write(new char[] { 'm', 'u', 'n', 'd', 'o' });
writer.write("! **");
writer.append('*');

// Agregar una nueva línea usando el separador de línea del sistema
writer.write(System.lineSeparator());

// Usar flush para asegurarse de que los datos se escriban en el archivo
writer.flush();

System.out.println("Se ha escrito en el archivo correctamente.");
} catch (IOException e) {
    System.err.println("Ocurrió un error al escribir en el archivo: " + e.getMessage());
}
}
}

```

Convierte un *String* en un array de caracteres que se escriben de uno en uno en el fichero

Ejemplo: agregar información a un fichero existente.

```

import java.io.FileWriter;
import java.io.IOException;

public class FileWriter2 {
    public static void main(String[] args) {
        String fileName = "archivo.txt";
        FileWriter writer = null;

        try {
            // El segundo parámetro "true" habilita el modo de adición, evitando que se
            // sobrescriba el contenido existente
            writer = new FileWriter(fileName, true);
            writer.write("Esta es una nueva línea agregada al archivo.");
            writer.write(System.lineSeparator()); // Nueva línea
            writer.flush();
            System.out.println("Se ha agregado información al archivo.");
        } catch (IOException e) {
            // Manejo de excepciones de entrada/salida, por ejemplo, errores de escritura
            System.err.println("Ocurrió un error al agregar información al archivo: " + e.getMessage());
        } finally {
            // El bloque finally se ejecuta siempre, tanto si se lanza una excepción como si no
            try {
                if (writer != null)
                    writer.close(); // Asegura el cierre del flujo para liberar recursos
            } catch (IOException e) {
                // Manejo de posibles excepciones al intentar cerrar el flujo
                System.err.println("No se pudo cerrar el flujo: " + e.getMessage());
            }
        }
    }
}

```

FileWriter fic = new FileWriter(fichero); //crear el flujo de salida  
String cadena = "Esto es una prueba con FileWriter";  
char[] cad = cadena.toCharArray(); //convierte un String en array de caracteres  
for(int i=0; i<cad.length; i++)  
 fic.write(cad[i]); //se va escribiendo un carácter  
fic.write(cad); //escribir un array de caracteres

Escribe el array de caracteres al completo

En este ejemplo, el constructor `FileWriter(fileName, true)` se utiliza para abrir el archivo en modo de adición, lo que significa que la información se agregará al final del archivo en lugar de sobrescribirlo. Se utiliza un bloque `try-catch-finally` para asegurar que el flujo se cierre.

Además, se ha utilizado un bloque `try-catch-finally` para manejar el flujo de datos y asegurar que el `FileWriter` se cierre correctamente, incluso si ocurre una excepción. No obstante, también se podría haber utilizado la estructura `try-with-resources`, que simplifica el manejo de recursos y garantiza automáticamente el cierre del `FileWriter` al finalizar el bloque `try`, eliminando la necesidad del bloque `finally`.

### 5.1.2 Cadena de caracteres.

La clase `BufferedWriter` proporciona una capa de búfer que mejora el rendimiento al escribir grandes cantidades de datos en un archivo. Se suele utilizar junto con `FileWriter` para optimizar la escritura en archivos. Los constructores de `BufferedWriter` están diseñados para envolver un flujo de escritura existente, mejorando el rendimiento mediante el uso de un búfer intermedio.

```

java.lang.Object
java.io.Writer
java.io.BufferedWriter
BufferedWriter fichero = new BufferedWriter(
    new FileWriter("FichTextol.txt"));

```

El método `write(String str)` permite escribir una cadena de caracteres al final de un fichero de texto, mientras que el método `newLine()` añade un salto de línea, facilitando la estructuración del contenido.

Si se desea agregar información a un archivo existente sin sobrescribir su contenido, es importante utilizar un constructor adecuado, como `FileWriter` con el parámetro `append` configurado en `true`, y luego emplear los

métodos de escritura apropiados para lograr el comportamiento deseado.

A continuación se muestra una tabla con los **constructores y algunos de los métodos más importantes** de la clase `BufferedWriter`:

Constructor/Método	Descripción
<code>BufferedWriter(Writer out)</code>	Crea un objeto <code>BufferedWriter</code> que utiliza el flujo de escritura especificado como base.
<code>BufferedWriter(Writer out, int bufferSize)</code>	Crea un objeto <code>BufferedWriter</code> con el flujo de escritura especificado y un tamaño de búfer personalizado.
<code>void write(int c)</code>	Escribe un carácter.
<code>void write(char[] cbuf)</code>	Escribe un array de caracteres.
<code>void write(char[] cbuf, int off, int len)</code>	Escribe una porción del array de caracteres.
<code>void write(String str)</code>	Escribe una cadena de caracteres.
<code>void write(String str, int off, int len)</code>	Escribe una porción de la cadena de caracteres.
<code>void newLine()</code>	Escribe una nueva línea.
<code>void flush()</code>	Limpia el búfer y asegura que los datos se escriban en el flujo de salida.
<code>void close()</code>	Cierra el flujo, liberando los recursos asociados.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/BufferedWriter.html>

Ejemplo:

```
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class FileCreationAndWriteEjemplo {
    public static void main(String[] args) {
        String fileName = "texto.txt"; // Nombre del archivo
        String contenido = "Este es el contenido del archivo."; // Contenido a escribir

        try { // Manejo de la creación y escritura del archivo
            File file = new File(fileName);

            if (file.createNewFile()) { // Crear un nuevo archivo si no existe
                System.out.println("Archivo creado: " + fileName);
            } else {
                clearFileContent(file); // Limpiar contenido del archivo existente
                System.out.println("Contenido del archivo existente borrado.");
            }

            writeToFile(file, contenido); // Escribir contenido en el archivo
            System.out.println("Contenido escrito en el archivo.");
        } catch (IOException e) {
            System.err.println("Error al manipular el archivo: " + e.getMessage());
        }
    }

    // Método para limpiar el contenido de un archivo
    private static void clearFileContent(File file) throws IOException {
        try (FileWriter fw = new FileWriter(file)) {
            fw.write(""); // Escribir una cadena vacía para borrar el contenido
        }
    }

    // Método para escribir contenido en un archivo utilizando BufferedWriter
    private static void writeToFile(File file, String contenido) throws IOException {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(file, true))) {
            writer.write(contenido);
            writer.newLine(); // Añadir una nueva línea después del contenido
        }
    }
}
```

En el ejemplo, se verifica si el archivo existe y se crea uno nuevo si no está presente. Si el archivo ya existe, se limpia su contenido utilizando `FileWriter` antes de escribir el nuevo contenido con `BufferedWriter`. Esta separación de acciones permite una gestión más clara y modular del archivo, asegurando que cualquier contenido previo sea eliminado antes de agregar el nuevo, sin sobrescribir el archivo por completo.



### 5.1.3 Escritura de ficheros de texto con formato.

Para escribir en ficheros de texto con un formato específico, se pueden utilizar diversas clases y técnicas. Una de las formas más comunes es **emplear las clases `FileWriter` y `BufferedWriter` junto con `PrintWriter`** para escribir líneas de texto formateadas.

`java.lang.Object`  
`java.io.Writer`  
`java.io.PrintWriter`

La **clase `PrintWriter`**, que hereda de `Writer`, **ofrece métodos como `print(arg)`, `println([arg])` y `printf(formato, args)`**, similares a los de `System.out`. Estos métodos **permiten imprimir datos de cualquier tipo** (char, boolean, int, etc.) en un fichero, con o sin salto de línea, y con un formato específico.

A continuación, se presenta una tabla con los **constructores y algunos métodos** importantes de la clase `PrintWriter`:

Constructor/Método	Descripción
<code>PrintWriter(File file)</code>	Crea un objeto <code>PrintWriter</code> a partir de un objeto <code>File</code> .
<code>PrintWriter(File file, String charset)</code>	Crea un objeto <code>PrintWriter</code> a partir de un objeto <code>File</code> , especificando un conjunto de caracteres (charset) para la codificación.
<code>PrintWriter(String fileName)</code>	Crea un <code>PrintWriter</code> a partir de un <code>String</code> .
<code>PrintWriter(String fileName, String charset)</code>	Crea un <code>PrintWriter</code> a partir de un <code>String</code> , especificando un conjunto de caracteres (charset) para la codificación.
<code>PrintWriter(OutputStream out)</code>	Crea un <code>PrintWriter</code> a partir de un objeto <code>OutputStream</code> .
<code>PrintWriter(OutputStream out, boolean autoFlush)</code>	Crea un <code>PrintWriter</code> a partir de un objeto <code>OutputStream</code> , con opción de auto flush.
<code>PrintWriter(Writer out)</code>	Crea un <code>PrintWriter</code> a partir de un objeto <code>Writer</code> .
<code>PrintWriter(Writer out, boolean autoFlush)</code>	Crea un <code>PrintWriter</code> a partir de un objeto <code>Writer</code> , con opción de auto flush.
<code>void print(boolean b)</code>	Escribe un valor booleano.
<code>void print(char c)</code>	Escribe un carácter.
<code>void print(int i)</code>	Escribe un valor entero.
<code>void print(double d)</code>	Escribe un valor de punto flotante.
<code>void print(String s)</code>	Escribe una cadena de caracteres.
<code>void print(Object obj)</code>	Escribe la representación de cadena de un objeto.
<code>void println()</code>	Escribe una línea en blanco.
<code>void println(boolean x)</code>	Escribe un valor booleano y una línea en blanco.
<code>void println(char x)</code>	Escribe un carácter y una línea en blanco.
<code>void println(int x)</code>	Escribe un valor entero y una línea en blanco.
<code>void println(double x)</code>	Escribe un valor de punto flotante y una línea en blanco.
<code>void println(String x)</code>	Escribe una cadena de caracteres y una línea en blanco.
<code>void println(Object x)</code>	Escribe la representación de cadena de un objeto y una línea en blanco.
<code>void flush()</code>	Limpia el búfer y escribe cualquier dato pendiente.
<code>void close()</code>	Cierra el <code>PrintWriter</code> , liberando recursos.
<code>PrintWriter append(CharSequence csq)</code>	Agrega la secuencia de caracteres <code>csq</code> a la salida.
<code>PrintWriter append(char c)</code>	Agrega el carácter <code>c</code> a la salida.
<code>void printf(String format, Object... args)</code>	Escribe una cadena de formato <code>format</code> utilizando los argumentos <code>args</code> y la envía a la secuencia de salida.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/PrintWriter.html>

**Ejemplo:**

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class EjemploPrintWriter {
    public static void main(String[] args) {
        String archivoSalida = "output.txt";
        // Uso de try-with-resources para manejo automático de recursos
        try (PrintWriter writer = new PrintWriter(new FileWriter(archivoSalida))) {
```





- ✓ %t1: hora en formato 12h.
- ✓ %tM: minutos (dos dígitos)
- ✓ %tp: indicador am/pm.
- ✓ %tm: mes en dígitos.
- ✓ %tD: fecha en formato %tm%td%ty.
- ✓ %u: decimal sin signo.
- ✓ %x o %X: entero hexadecimal.
- ✓ %: imprime el carácter %.

**Especificaciones avanzadas** para controlar precisión, ancho del campo, alineación, ...:

- ✓ %.3f: muestra un número de punto flotante con tres decimales.
- ✓ %10s: cadena con un ancho de 10 caracteres.
- ✓ %-10s: cadena con ancho de 10 caracteres alineada a la izquierda.
- ✓ %5d: entero con un ancho de 5 caracteres.
- ✓ %05d: entero con ancho de 5 caracteres, rellenando con ceros.

*Más información sobre los formatos que se pueden utilizar con printf:*

<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Formatter.html#syntax>

#### Ejemplo:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class EscribirFicheroConFormato2 {
    public static void main(String[] args) {
        String archivoSalida = "salida.txt";
        String nombre = "Juan Moreno";
        int edad = 30;
        double altura = 1.75;

        try (PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter(archivoSalida)))) {
            pw.printf("Nombre: %s\n", nombre);
            pw.printf("Edad: %d\n", edad);
            pw.printf("Altura: %.2f\n", altura);
            pw.printf("Altura en notación científica: %.2e\n", altura);
            pw.printf("Altura en notación científica con letras mayúsculas: %.2E\n", altura);
            pw.printf("Edad en hexadecimal: %x\n", edad); // Cambiado "Altura" a "Edad"
            pw.printf("Edad en octal: %o\n", edad); // Cambiado "Altura" a "Edad"
            pw.printf("¿Es mayor de edad? %b\n", edad >= 18);
            pw.printf("Carácter: %c\n", 'A');
            pw.printf("Cadena con ancho de campo de 20 caracteres: %20s\n", nombre);
            pw.printf("Cadena alineada a la izquierda con ancho de campo de 20 caracteres: %-20s\n", nombre);
            pw.printf("Entero con ancho de campo de 5 caracteres: %5d\n", edad);
            pw.printf("Entero rellenado con ceros con ancho de campo de 5 caracteres: %05d\n", edad);

            System.out.println("Archivo de salida creado con éxito.");
        } catch (IOException e) {
            System.err.println("Error al escribir en el archivo: " + e.getMessage());
        }
    }
}
```

```
Nombre: Juan Moreno
Edad: 30
Altura: 1,75
Altura en notación científica: 1,75e+00
Altura en notación científica con letras mayúsculas: 1,75E+00
Altura en hexadecimal: 1e
Altura en octal: 36
¿Es mayor de edad? true
Carácter: A
Cadena con ancho de campo de 20 caracteres: Juan Moreno
Cadena alineada a la izquierda con ancho de campo de 20 caracteres: Juan Moreno
Entero con ancho de campo de 5 caracteres: 30
Entero rellenado con ceros con ancho de campo de 5 caracteres: 00030
```

#### Ejemplo:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class EscribirFicheroConFormato3 {
    // Definir una constante para el formato de las líneas
    private static final String FORMATO_LINEA = "%-10s %8.2f %6d\n";

    public static void main(String[] args) {
        String archivoSalida = "salida.txt";

        // Manejo de recursos con try-with-resources
        try (PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter(archivoSalida)))) {
```

Artículo	Precio/Kg.	Nº Kg.
Tomates	1,50	10
Judías	3,75	2
Pimientos	1,85	6

```

        pw.println("Artículo      Precio/Kg.      Nº Kg.");
        pw.println("-----");
        pw.printf(FORMATO_LINEA, "Tomates", 1.5, 10);
        pw.printf(FORMATO_LINEA, "Judías", 3.75, 2);
        pw.printf(FORMATO_LINEA, "Pimientos", 1.85, 6);

        System.out.println("Archivo de salida creado con éxito.");
    } catch (IOException e) {
        System.err.println("Error al escribir en el archivo: " + e.getMessage());
    }
}
}

```

## 5.2 Leer de ficheros de texto.

```

java.lang.Object
java.io.Reader
    java.io.InputStreamReader
        java.io.FileReader

```

### 5.2.1 Carácter a carácter.

Para leer un fichero de texto carácter a carácter **se debe utilizar la clase FileReader**. El **constructor** de esta clase requiere como parámetro el archivo que se desea leer, y si este no se encuentra, se lanzará una excepción **FileNotFoundException**.

**FileReader** proporciona el **método read**, que **permite leer sucesivamente un carácter** desde el inicio del archivo, **devolviendo el código del carácter como un valor entero**. Si se alcanza el final del fichero, **read retorna -1**.

Para leer un archivo de texto de manera iterativa, carácter por carácter, **se puede utilizar un bucle while**:

```

while ((caracter = fileReader.read()) != -1) {
    // Operaciones con cada carácter leído
}

```

Dado que **read()** puede generar una excepción **IOException**, es necesario **envolver las operaciones de lectura dentro de una estructura try-catch**.

**Constructores y métodos importantes de FileReader:**

Constructor/Método	Descripción
<code>FileReader(File file)</code>	Crea un nuevo <code>FileReader</code> para leer el archivo especificado por el objeto <code>File</code> .
<code>FileReader(String fileName)</code>	Crea un nuevo <code>FileReader</code> para leer el archivo especificado por su nombre.
<code>int read()</code>	Lee el siguiente carácter y devuelve su valor como un entero, o -1 si se alcanza el final del archivo.
<code>int read(char[] cbuf)</code>	Lee caracteres en el arreglo proporcionado y devuelve la cantidad leída, o -1 si se alcanza el final del archivo.
<code>int read(char[] cbuf, int off, int len)</code>	Lee hasta <code>len</code> caracteres en el arreglo <code>cbuf</code> desde la posición <code>off</code> . Devuelve la cantidad leída o -1 si se llega al final del archivo.
<code>long skip(long n)</code>	Omite <code>n</code> caracteres en el flujo de entrada y devuelve el número de caracteres realmente omitidos.
<code>void close()</code>	Cierra el flujo de lectura. Es importante cerrar el <code>FileReader</code> cuando ya no se necesita.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/FileReader.html>

**Ejemplo:** lectura del fichero de texto "archivo.txt" que debe encontrarse en la carpeta del proyecto.

```

import java.io.FileReader;
import java.io.IOException;
import java.io.FileNotFoundException;

public class LeerArchivoCaracterACaracter {
    public static void main(String[] args) {
        String filePath = "archivo.txt";

        // Bloque try-with-resources para asegurar el cierre automático del FileReader
        try (FileReader reader = new FileReader(filePath)) {
            int charCode;

            while ((charCode = reader.read()) != -1) // Lectura carácter a carácter hasta el final
                System.out.print((char) charCode); // Convertir y mostrar cada carácter
        }
    }
}

```

```

    } catch (FileNotFoundException e) {
        System.err.println("Error: el archivo no fue encontrado - " + e.getMessage());
    } catch (IOException e) {
        System.err.println("Error al leer el archivo: " + e.getMessage());
    }
}
}

```

En este ejemplo, el `FileReader` se utiliza dentro de un bloque `try-with-resources`, lo que garantiza que el archivo se cerrará automáticamente al finalizar el bloque, sin necesidad de un bloque `finally`.

### 5.2.2 Línea a línea.

```

java.lang.Object
java.io.Reader
java.io.BufferedReader

```

Para leer un fichero de texto línea a línea, se utilizan la clase **`BufferedReader`** en combinación con **`FileReader`**. Esta combinación permite una lectura más eficiente del archivo.

#### Pasos para leer un archivo línea a línea:

1. **Crear una instancia de `FileReader` y `BufferedReader`:** primero, se crea una instancia de `FileReader` usando la ruta del archivo. Luego, se envuelve este `FileReader` en un `BufferedReader` para una lectura más rápida y eficiente.
2. **Leer línea a línea:** se utiliza un **bucle `while` junto con el método `readLine()`** del `BufferedReader` para leer el archivo línea a línea. Este método **devuelve `null` al llegar al final del archivo**.

**Constructores y métodos importantes de `BufferedReader`:**

Constructor/Método	Descripción
<code>BufferedReader(Reader reader)</code>	Crea un nuevo <code>BufferedReader</code> que lee caracteres de un <code>Reader</code> especificado.
<code>BufferedReader(Reader reader, int size)</code>	Crea un nuevo <code>BufferedReader</code> con un tamaño de búfer especificado para mejorar la eficiencia de la lectura.
<code>int read()</code>	Lee un solo carácter y devuelve su valor como entero, o -1 si se alcanza el final del flujo.
<code>int read(char[] cbuf, int off, int len)</code>	Lee caracteres en el búfer <code>cbuf</code> desde la posición <code>off</code> , hasta un máximo de <code>len</code> . Devuelve el número de caracteres leídos o -1 si se alcanza el final.
<code>String readLine()</code>	Lee una línea de texto completa y devuelve la línea leída como una cadena, o <code>null</code> si se llega al final.
<code>boolean ready()</code>	Verifica si el flujo está listo para ser leído.
<code>void close()</code>	Cierra el flujo y libera los recursos asociados.
<code>Stream&lt;String&gt; lines()</code>	Devuelve un flujo (stream) cuyas líneas se leen de <code>BufferedReader</code> .

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/BufferedReader.html>

Ejemplo: leer un archivo de texto línea a línea.

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class EjemploBufferedReader {
    public static void main(String[] args) {
        String rutaArchivo = "archivo.txt";

        // Uso de try-with-resources para asegurar el cierre del BufferedReader
        try (BufferedReader bufferLector = new BufferedReader(new FileReader(rutaArchivo))) {
            String linea;
            int numeroLinea = 1;

            System.out.println("--- Lectura de líneas ---");
            while ((linea = bufferLector.readLine()) != null) // Leer el archivo línea a línea
                System.out.printf("Línea %d: %s\n", numeroLinea++, linea);
        } catch (IOException e) {
            System.err.printf("Error al leer el archivo '%s': %s\n", rutaArchivo, e.getMessage());
        }
    }
}

```

Ejemplo: procesar archivos CSV simples.

```

import java.io.BufferedReader;

```

```

import java.io.FileReader;
import java.io.IOException;

public class EjemploLecturaCSV {
    public static void main(String[] args) {
        String rutaArchivo = "archivo.csv";

        // Bloque try-with-resources para asegurar el cierre del BufferedReader
        try (BufferedReader bufferLector = new BufferedReader(new FileReader(rutaArchivo))) {
            String linea;

            // Leer y procesar el encabezado (primera línea)
            if ((linea = bufferLector.readLine()) != null) {
                String[] camposEncabezado = linea.split(",");
                System.out.println("Encabezado:");
                System.out.println(String.join(" | ", camposEncabezado)); // Imprimir encabezado
            }

            // Leer y procesar los registros
            System.out.println("\nRegistros:");
            while ((linea = bufferLector.readLine()) != null) {
                String[] campos = linea.split(",");
                System.out.println(String.join(" | ", campos)); // Imprimir cada registro
            }
        } catch (IOException e) {
            System.err.printf("Error al leer el archivo '%s': %s\n", rutaArchivo, e.getMessage());
        }
    }
}

```

Nombre,Edad,Ciudad,Profesión  
Juan Pérez,30,Madrid,Ingeniero  
Ana Gómez,25,Barcelona,Abogada  
Luis García,35,Sevilla,Doctor

### 5.2.3 Leer diferentes tipos de datos.



Para leer diferentes tipos de datos desde un archivo en *Java*, se puede utilizar la clase `java.lang.Object` **Scanner** junto con **FileReader**. `Scanner` facilita la lectura y el análisis de diversos tipos de datos como enteros, números decimales, cadenas, etc. desde una secuencia de entrada. `java.util.Scanner`

Tabla con los constructores y métodos más utilizados de la clase `Scanner` en *Java* para trabajar con ficheros:

Constructor/Método	Descripción	Ejemplo
<code>Scanner(File source)</code>	Crea un <code>Scanner</code> que lee datos de un archivo.	<code>Scanner sc = new Scanner(new File("data.txt"));</code>
<code>Scanner(InputStream source)</code>	Crea un <code>Scanner</code> que lee datos de un flujo de entrada.	<code>Scanner sc = new Scanner(System.in);</code>
<code>Scanner(String source)</code>	Crea un <code>Scanner</code> que lee datos de una cadena de texto.	<code>Scanner sc = new Scanner("123 456 789");</code>
<code>boolean hasNext()</code>	Devuelve <code>true</code> si hay otro token disponible.	<code>while (sc.hasNext()) {...}</code>
<code>boolean hasNextLine()</code>	Devuelve <code>true</code> si hay otra línea disponible.	<code>if (sc.hasNextLine()) {...}</code>
<code>String next()</code>	Devuelve el siguiente token como una cadena de texto.	<code>String word = sc.next();</code>
<code>String nextLine()</code>	Lee y devuelve la siguiente línea completa.	<code>String line = sc.nextLine();</code>
<code>int nextInt()</code>	Devuelve el siguiente token como un entero.	<code>int num = sc.nextInt();</code>
<code>double nextDouble()</code>	Devuelve el siguiente token como un número de punto flotante.	<code>double d = sc.nextDouble();</code>
<code>boolean hasNextInt()</code>	Devuelve <code>true</code> si el siguiente token puede interpretarse como un entero.	<code>if (sc.hasNextInt()) {...}</code>
<code>boolean hasNextDouble()</code>	Devuelve <code>true</code> si el siguiente token puede interpretarse como un número de punto flotante.	<code>if (sc.hasNextDouble()) {...}</code>
<code>long nextLong()</code>	Devuelve el siguiente token como un número largo (long).	<code>long l = sc.nextLong();</code>
<code>float nextFloat()</code>	Devuelve el siguiente token como un número flotante simple (float).	<code>float f = sc.nextFloat();</code>
<code>short nextShort()</code>	Devuelve el siguiente token como un número corto (short).	<code>short s = sc.nextShort();</code>
<code>byte nextByte()</code>	Devuelve el siguiente token como un byte.	<code>byte b = sc.nextByte();</code>
<code>boolean nextBoolean()</code>	Devuelve el siguiente token como un valor booleano ( <code>true</code> o <code>false</code> ).	<code>boolean bool = sc.nextBoolean();</code>
<code>boolean hasNextLong()</code>	Devuelve <code>true</code> si el siguiente token puede interpretarse como un número largo (long).	<code>if (sc.hasNextLong()) {...}</code>
<code>boolean hasNextFloat()</code>	Devuelve <code>true</code> si el siguiente token puede interpretarse como un número flotante simple (float).	<code>if (sc.hasNextFloat()) {...}</code>
<code>boolean hasNextShort()</code>	Devuelve <code>true</code> si el siguiente token puede interpretarse como un número corto (short).	<code>if (sc.hasNextShort()) {...}</code>
<code>boolean hasNextByte()</code>	Devuelve <code>true</code> si el siguiente token puede interpretarse como un byte.	<code>if (sc.hasNextByte()) {...}</code>
<code>void close()</code>	Cierra el <code>Scanner</code> y libera cualquier recurso asociado.	<code>sc.close();</code>

<b>void</b> <b>useDelimiter</b> (Pattern pattern)	Establece un delimitador personalizado para los tokens usando un patrón de expresión regular.	<code>sc.useDelimiter(",");</code>
<b>Scanner</b> <b>useLocale</b> (Locale locale)	Establece un Locale que afecta el reconocimiento de números y cadenas.	<code>sc.useLocale(Locale.US);</code>
<b>String</b> <b>findInLine</b> (Pattern pattern)	Busca un patrón en la línea actual y devuelve la primera ocurrencia coincidente.	<code>sc.findInLine("\\d+");</code>
<b>String</b> <b>findWithinHorizon</b> (Pattern pattern, int horizon)	Busca un patrón dentro de un rango y devuelve el primer match encontrado.	<code>sc.findWithinHorizon("\\d{3}", 100);</code>
<b>void</b> <b>reset</b> ()	Resetea el escáner a su estado inicial, incluyendo el delimitador y el Locale.	<code>sc.reset();</code>

Para **configurar el Locale en español** en un objeto Scanner, se debe utilizar el método `useLocale(Locale locale)`. En *Java*, se puede especificar un Locale para el idioma español de la siguiente manera:

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

public class ScannerLocaleEjemplo {
    public static void main(String[] args) {
        try (Scanner sc = new Scanner(new File("datos.txt"))) {
            sc.useLocale(Locale.of("es", "ES"));

            while (sc.hasNext()) { // Leer y procesar datos del archivo
                if (sc.hasNextDouble()) { // Verificar si el siguiente token es un número de punto flotante
                    double numero = sc.nextDouble();
                    System.out.println("Número leído: " + numero);
                } else {
                    String texto = sc.next(); // Leer y mostrar el siguiente token como texto
                    System.out.println("Texto leído: " + texto);
                }
            }
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Scanner.html>

**Ejemplo:**

```
import java.io.FileReader;
import java.io.IOException;
import java.util.InputMismatchException;
import java.util.Scanner;

public class ScannerFileReaderEjemplo {
    public static void main(String[] args) {
        String rutaArchivo = "salida.txt"; // Ruta del archivo

        // Bloque try-with-resources para asegurar el cierre automático del FileReader y Scanner
        try (FileReader fr = new FileReader(rutaArchivo);
            Scanner scanner = new Scanner(fr)) {
            // Leer y analizar diferentes tipos de datos desde el archivo
            while (scanner.hasNext()) {
                if (scanner.hasNextInt()) {
                    int numeroEntero = scanner.nextInt();
                    System.out.println("Entero: " + numeroEntero);
                } else if (scanner.hasNextDouble()) {
                    double numeroDouble = scanner.nextDouble();
                    System.out.println("Double: " + numeroDouble);
                } else if (scanner.hasNextFloat()) {
                    float numeroFloat = scanner.nextFloat();
                    System.out.println("Float: " + numeroFloat);
                } else if (scanner.hasNextBoolean()) {
                    boolean valorBooleano = scanner.nextBoolean();
                    System.out.println("Booleano: " + valorBooleano);
                } else { // Si no es ninguno de los tipos anteriores, asumimos que es una cadena
                    String texto = scanner.next();
                    System.out.println("Cadena: " + texto);
                }
            }
        }
    }
}
```



```

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

```



## TAREA

4. Crear un programa que lea un archivo de texto y cuente el número total de palabras que contiene. Posteriormente, crear un nuevo archivo de texto para escribir el conteo total de palabras.
5. Crear un programa que solicite al usuario los nombres de dos archivos de texto. El programa debe copiar el contenido del primer archivo al segundo. Verificar que el primer archivo exista antes de proceder con la copia.
6. (Opcional) Crear un programa que lea un archivo de texto y permita al usuario ingresar una palabra para buscar y otra palabra para reemplazarla. El contenido modificado debe escribirse en un nuevo archivo.
7. Crear un programa que lea un archivo de texto y aplique un cifrado *César* con un desplazamiento de 5 posiciones a cada carácter del archivo. El resultado cifrado debe guardarse en un nuevo archivo.
8. Crear un programa que tome como entrada dos archivos de texto ordenados alfabéticamente y combine su contenido en un nuevo archivo, manteniendo el orden alfabético. **Nota:** no se permite el uso de listas para este ejercicio.

## 6 Ficheros binarios.

Los ficheros binarios **son archivos que almacenan información en un formato que no es legible directamente por humanos**. A diferencia de los ficheros de texto, que se pueden leer con un editor de texto, los ficheros binarios **guardan datos como secuencias de bytes**, que representan información en formato binario (ceros y unos). Estos ficheros **se utilizan para una amplia variedad de datos**, como imágenes, audio, video, ejecutables y *BD*.

**Características** de los ficheros binarios:

- ✓ **Formato no legible:** no se pueden leer directamente; se necesita un programa o aplicación específica para interpretar su contenido.
- ✓ **Estructura específica:** cada tipo de fichero binario tiene una estructura propia. Por ejemplo, los ficheros *JPEG* para imágenes tienen una estructura diferente a los ficheros *MP3* para audio. Es crucial entender esta estructura para procesarlos correctamente.
- ✓ **Compactos y eficientes:** generalmente, son más compactos y eficientes en términos de espacio de almacenamiento que los ficheros de texto.
- ✓ **Uso común en programación:** son utilizados para almacenar datos complejos como imágenes y videos. Los programas pueden leer y escribir datos binarios para realizar diversas tareas.

### 6.1 Escribir en ficheros binarios.

#### 6.1.1 Byte a byte.

```

java.lang.Object
java.io.OutputStream
java.io.FileOutputStream

```

La clase `FileOutputStream` se utiliza para escribir datos binarios en ficheros, **byte a byte**. Es útil para trabajar con imágenes, archivos de sonido, *BD* y otros datos no estructurados. Para escribir datos en formato de texto, es mejor usar clases como `FileWriter` o `BufferedWriter`.

**Constructores y métodos** comunes de `FileOutputStream`:

Constructor/Método	Descripción
<code>FileOutputStream(String name)</code>	Crea un <code>FileOutputStream</code> para abrir un archivo con el nombre especificado en modo de escritura.
<code>FileOutputStream(File file)</code>	Crea un <code>FileOutputStream</code> para abrir un archivo representado por un objeto <code>File</code> .
<code>FileOutputStream(String name, boolean append)</code>	Crea un <code>FileOutputStream</code> para abrir un archivo con el nombre especificado en modo de escritura, opcionalmente en modo de anexo si <code>append</code> es <code>true</code> .

<code>FileOutputStream(File file, boolean append)</code>	Crea un <code>FileOutputStream</code> para abrir un archivo representado por un objeto <code>File</code> , opcionalmente en modo de anexo si <code>append</code> es <code>true</code> .
<code>void write(int b)</code>	Escribe un byte en el flujo de salida.
<code>void write(byte[] b)</code>	Escribe un array de bytes en el flujo de salida.
<code>void write(byte[] b, int off, int len)</code>	Escribe un subconjunto del array de bytes, comenzando en la posición <code>off</code> y escribiendo <code>len</code> bytes.
<code>void flush()</code>	Limpia el flujo de salida, asegurando que todos los datos pendientes se escriban al archivo.
<code>void close()</code>	Cierra el <code>FileOutputStream</code> y libera los recursos asociados.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/FileOutputStream.html>

**Ejemplo:**

```
import java.io.FileOutputStream;
import java.io.IOException;

public class EjemploFileOutputStream {
    public static void main(String[] args) {
        String archivo = "archivo.bin"; // Ruta y nombre del archivo binario
        // Escribe bytes en el archivo utilizando FileOutputStream
        try (FileOutputStream fo = new FileOutputStream(archivo)) {
            // Escribe una secuencia de bytes en el archivo
            for (int i = 0; i < 100; i++)
                fo.write((byte) i); // Convierte el valor entero a byte y lo escribe
            fo.flush(); // Asegura que todos los datos pendientes se escriban en el archivo
            System.out.println("Datos escritos exitosamente en " + archivo);
        } catch (IOException e) { // Maneja errores de entrada/salida
            System.err.println("Error al escribir en el archivo: " + e.getMessage());
        }
    }
}
```

```
java.lang.Object
java.io.OutputStream
java.io.FilterOutputStream
java.io.BufferedOutputStream
```

### 6.1.2 Byte a byte con búfer.

`BufferedOutputStream` es una clase en *Java* que extiende `OutputStream` y mejora la eficiencia de la escritura en un flujo de salida mediante el uso de un búfer interno. **En lugar de escribir los datos directamente, los acumula en un búfer y los envía en bloques grandes**, reduciendo así la cantidad de operaciones de escritura y mejorando el rendimiento. Es comúnmente **utilizada con flujos como `FileOutputStream` para escribir datos de manera más rápida y eficiente**. `BufferedOutputStream` **escribe datos en formato binario**.

**Constructores y métodos** comunes de `BufferedOutputStream`:

Constructor/Método	Descripción
<code>BufferedOutputStream(OutputStream out)</code>	Crea un <code>BufferedOutputStream</code> con un búfer predeterminado de 8192 bytes, envolviendo el flujo de salida especificado.
<code>BufferedOutputStream(OutputStream out, int size)</code>	Crea un <code>BufferedOutputStream</code> con un búfer de tamaño especificado, envolviendo el flujo de salida dado.
<code>void write(int b)</code>	Escribe un solo byte en el búfer. Si el búfer está lleno, se vacía y se escribe en el flujo subyacente.
<code>void write(byte[] b)</code>	Escribe un array de bytes en el flujo de salida.
<code>void write(byte[] b, int off, int len)</code>	Escribe una porción del arreglo de bytes en el búfer, comenzando desde la posición <code>off</code> y escribiendo <code>len</code> bytes.
<code>void flush()</code>	Vacía el búfer, escribiendo todos los datos acumulados en el flujo de salida subyacente. Esto asegura que todos los datos pendientes se escriban antes de continuar.
<code>void close()</code>	Cierra el flujo de salida y asegura que todos los datos pendientes en el búfer se escriban en el flujo subyacente antes de cerrar.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/BufferedOutputStream.html>

**Ejemplo:**

```
import java.io.BufferedOutputStream;
import java.io.FileOutputStream;
```

```

import java.io.IOException;

public class EjemploBufferedOutputStream {
    public static void main(String[] args) {
        String nombreArchivo = "datos.bin";

        // Datos para escribir
        int entero = 12345;
        float decimalFloat = 3.14f;
        double decimalDouble = 2.71828;
        String texto = "Este es un ejemplo de texto.";

        try (BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(nombreArchivo))) {
            // Convertir y escribir un entero (4 bytes)
            bos.write(intToBytes(entero));
            System.out.println("Entero 12345 escrito correctamente.");

            // Convertir y escribir un número decimal (float, 4 bytes)
            bos.write(floatToBytes(decimalFloat));
            System.out.println("Decimal (float) 3.14 escrito correctamente.");

            // Convertir y escribir un número decimal (double, 8 bytes)
            bos.write(doubleToBytes(decimalDouble));
            System.out.println("Decimal (double) 2.71828 escrito correctamente.");

            // Convertir y escribir una cadena de texto (variable en longitud)
            bos.write(texto.getBytes());
            System.out.println("Texto escrito correctamente.");

            bos.flush(); // Asegurar que todos los datos se escriban en el archivo
            System.out.println("Datos escritos correctamente en el archivo " + nombreArchivo);
        } catch (IOException e) {
            System.err.println("Error al escribir en el archivo: " + e.getMessage());
        }
    }

    // Método para convertir un entero a bytes (Big Endian)
    private static byte[] intToBytes(int value) {
        return new byte[] {
            (byte) (value >>> 24),
            (byte) (value >>> 16),
            (byte) (value >>> 8),
            (byte) value
        };
    }

    // Método para convertir un float a bytes (Big Endian)
    private static byte[] floatToBytes(float value) {
        int intBits = Float.floatToIntBits(value);
        return intToBytes(intBits);
    }

    // Método para convertir un double a bytes (Big Endian)
    private static byte[] doubleToBytes(double value) {
        long longBits = Double.doubleToLongBits(value);
        return new byte[] {
            (byte) (longBits >>> 56),
            (byte) (longBits >>> 48),
            (byte) (longBits >>> 40),
            (byte) (longBits >>> 32),
            (byte) (longBits >>> 24),
            (byte) (longBits >>> 16),
            (byte) (longBits >>> 8),
            (byte) longBits
        };
    }
}

```

### 6.1.3 Datos primitivos y tipos de datos estructurados.

La clase `DataOutputStream` permite escribir datos primitivos y tipos de datos estructurados en un flujo de salida, como un archivo binario. Estos datos se almacenan en formato binario, lo que asegura que se pueden leer de manera consistente con la clase `DataInputStream`.

```

java.lang.Object
  java.io.OutputStream
    java.io.FilterOutputStream
      java.io.DataOutputStream

```

**Constructores y métodos comunes de `DataOutputStream`:**

Constructor/Método	Descripción
<code>DataOutputStream(OutputStream out)</code>	Crea un <code>DataOutputStream</code> para escribir datos primitivos en un flujo de salida especificado ( <code>OutputStream</code> ).
<code>void write(int b)</code>	Escribe un byte en el flujo de salida.
<code>void write(byte[] b)</code>	Escribe un array de bytes en el flujo de salida.
<code>void write(byte[] b, int off, int len)</code>	Escribe un subconjunto del array de bytes en el flujo de salida, comenzando en la posición <code>off</code> y escribiendo <code>len</code> bytes.
<code>void writeBoolean(boolean v)</code>	Escribe un valor booleano en el flujo de salida como un byte.
<code>void writeByte(int v)</code>	Escribe un byte en el flujo de salida.
<code>void writeBytes(String s)</code>	Escribe una cadena de texto en el flujo de salida como una secuencia de bytes.
<code>void writeChar(int v)</code>	Escribe un carácter Unicode de 2 bytes en el flujo de salida.
<code>void writeChars(String s)</code>	Escribe una cadena de texto en el flujo de salida como una secuencia de caracteres Unicode.
<code>void writeDouble(double v)</code>	Escribe un valor de punto flotante de 8 bytes en el flujo de salida.
<code>void writeFloat(float v)</code>	Escribe un valor de punto flotante de 4 bytes en el flujo de salida.
<code>void writeInt(int v)</code>	Escribe un entero de 4 bytes en el flujo de salida.
<code>void writeLong(long v)</code>	Escribe un entero largo de 8 bytes en el flujo de salida.
<code>void writeShort(int v)</code>	Escribe un entero corto de 2 bytes en el flujo de salida.
<code>void writeUTF(String str)</code>	Escribe una cadena de texto en formato UTF-8 en el flujo de salida.
<code>void flush()</code>	Limpia el flujo de salida para asegurar que todos los datos pendientes se escriban.
<code>void close()</code>	Cierra el <code>DataOutputStream</code> y libera los recursos asociados.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/DataOutputStream.html>

#### Ejemplo:

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class EjemploDataOutputStream {
    public static void main(String[] args) {
        String archivo = "datos.bin"; // Ruta y nombre del archivo binario

        try (FileOutputStream fos = new FileOutputStream(archivo);
            DataOutputStream dos = new DataOutputStream(fos)) {
            dos.writeBoolean(true); // Escribe un valor booleano
            dos.writeInt(42); // Escribe un entero
            dos.writeFloat(3.14f); // Escribe un valor de punto flotante
            dos.writeUTF("Hola, mundo!"); // Escribe una cadena en formato UTF-8

            dos.flush(); // Limpia el flujo de salida para asegurarse de que todos los datos se escriban
            System.out.println("Datos escritos en el archivo correctamente.");
        } catch (IOException e) {
            System.err.println("Error al escribir en el archivo: " + e.getMessage());
        }
    }
}
```

## 6.2 Leer de ficheros binarios.

### 6.2.1 Byte a byte.

```
java.lang.Object
  java.io.InputStream
    java.io.FileInputStream
```

La clase `FileInputStream` se usa para leer datos de archivos como un flujo de bytes, ya sea byte por byte o en bloques. Es adecuada para leer secuencialmente datos binarios como imágenes, archivos de audio o cualquier otro tipo de archivo que no sea texto simple. Los métodos `read()` devuelven el número de bytes leídos o -1 si se alcanza el final del archivo. Para archivos grandes, es mejor usar un búfer para mejorar el rendimiento.

`FileInputStream` no permite retroceder en el flujo (no soporta `mark/reset`). Si se necesita volver a leer desde

un punto anterior, será necesario cerrar y reabrir el flujo o usar una clase que soporte marcadores.

**Constructores y métodos comunes de FileInputStream:**

Constructor/Método	Descripción
<code>FileInputStream(String name)</code>	Crea un <code>FileInputStream</code> para abrir el archivo con el nombre especificado.
<code>FileInputStream(File file)</code>	Crea un <code>FileInputStream</code> para abrir el archivo representado por un objeto <code>File</code> .
<code>int read()</code>	Lee un byte del archivo y devuelve el valor del byte como un entero entre 0 y 255. Devuelve -1 si llega al final del archivo.
<code>int read(byte[] b)</code>	Lee un array de bytes del archivo y los almacena en un array <code>b</code> . Devuelve el número de bytes leídos o -1 si llega al final del archivo.
<code>int read(byte[] b, int off, int len)</code>	Lee hasta <code>len</code> bytes del archivo, comenzando en la posición <code>off</code> del array <code>b</code> . Devuelve el número de bytes leídos o -1 si llega al final del archivo.
<code>long skip(long n)</code>	Salta <code>n</code> bytes en el archivo. Devuelve el número de bytes realmente saltados.
<code>int available()</code>	Devuelve el número de bytes que se pueden leer.
<code>void close()</code>	Cierra el <code>FileInputStream</code> y libera los recursos asociados.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/FileInputStream.html>

Ejemplo:

```
import java.io.FileInputStream;
import java.io.IOException;
public class EjemploFileInputStream {
    public static void main(String[] args) {
        String archivo = "archivo.bin"; // Ruta y nombre del archivo binario a leer

        try (FileInputStream fileInput = new FileInputStream(archivo)) {
            // Lee bytes del archivo y muestra su valor
            System.out.println("Leyendo bytes del archivo:");
            int byteLeido;
            while ((byteLeido = fileInput.read()) != -1)
                System.out.print(byteLeido + " ");

            long bytesSaltados = fileInput.skip(5); // Salta 5 bytes
            System.out.println("\n\nSe han saltado " + bytesSaltados + " bytes.");

            // Lee y muestra los siguientes 10 bytes
            System.out.println("\nLeyendo los siguientes 10 bytes:");
            byte[] buffer = new byte[10];
            int bytesRead = fileInput.read(buffer);
            for (int i = 0; i < bytesRead; i++)
                System.out.print(buffer[i] + " ");
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        }
    }
}
```

```
java.lang.Object
java.io.InputStream
java.io.FilterInputStream
java.io.BufferedInputStream
```

### 6.2.2 Byte a byte con búfer.

La clase `BufferedInputStream` en *Java* mejora la eficiencia de la lectura de datos al utilizar un búfer interno para almacenar temporalmente los datos leídos de un flujo de entrada. Esto reduce la cantidad de accesos físicos a la fuente de datos, como archivos o redes, al leer grandes bloques de datos en lugar de bytes individuales. Esto permite realizar lecturas más rápidas y con menor carga en el sistema, y proporciona métodos adicionales como `mark()` y `reset()` para manejar posiciones en el flujo de manera más flexible.

**Constructores y métodos comunes de BufferedInputStream:**

Constructor/Método	Descripción
<code>BufferedInputStream(InputStream in)</code>	Crea un <code>BufferedInputStream</code> que utiliza un búfer interno para leer datos del flujo <code>in</code> .
<code>BufferedInputStream(InputStream in, int size)</code>	Crea un <code>BufferedInputStream</code> con un búfer interno de tamaño específico <code>size</code> , para leer datos del flujo <code>in</code> .
<code>int read()</code>	Lee un solo byte de datos del flujo. Devuelve el byte leído como un entero (en el rango de 0 a 255) o -1 si se ha alcanzado el final del flujo.
<code>byte[] readAllBytes()</code>	Lee todos los bytes restantes del flujo y los devuelve como un arreglo de bytes.

<code>int read(byte[] b, int off, int len)</code>	Lee hasta len bytes de datos del flujo y los almacena en el arreglo b, comenzando en la posición off. Devuelve el número de bytes leídos o -1 si se ha alcanzado el final del flujo.
<code>byte[] readNBytes(int len)</code>	Lee hasta len bytes de datos del flujo y los devuelve como un arreglo de bytes. Si el flujo alcanza el final antes de leer len bytes, el arreglo tendrá un tamaño menor al solicitado.
<code>void mark(int readlimit)</code>	Marca la posición actual en el flujo de entrada para que se pueda volver a ella después de una llamada a <code>reset()</code> , dentro del límite especificado <code>readlimit</code> .
<code>void reset()</code>	Restablece el flujo a la última posición marcada por <code>mark()</code> .
<code>long skip(long n)</code>	Omite y descarta hasta n bytes de datos del flujo. Devuelve el número de bytes realmente saltados.
<code>void close()</code>	Cierra el flujo de entrada y libera los recursos asociados.
<code>long transferTo(OutputStream out)</code>	Transfiere los bytes restantes del flujo actual al flujo de salida out. Devuelve el número de bytes transferidos.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/BufferedInputStream.html>

### Ejemplo:

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class EjemploBufferedInputStream {
    public static void main(String[] args) {
        String nombreArchivo = "datos.bin";

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(nombreArchivo))) {
            // Leer y mostrar un entero (4 bytes)
            int entero = bytesToInt(bis.readNBytes(4));
            System.out.println("Entero leído: " + entero);

            // Leer y mostrar un número decimal (float, 4 bytes)
            float decimalFloat = bytesToFloat(bis.readNBytes(4));
            System.out.println("Decimal (float) leído: " + decimalFloat);

            // Leer y mostrar un número decimal (double, 8 bytes)
            double decimalDouble = bytesToDouble(bis.readNBytes(8));
            System.out.println("Decimal (double) leído: " + decimalDouble);

            // Leer y mostrar una cadena de texto (resto del archivo)
            byte[] textoBytes = bis.readAllBytes();
            String texto = new String(textoBytes);
            System.out.println("Texto leído: " + texto);
        } catch (IOException e) {
            System.err.println("Error al leer el archivo: " + e.getMessage());
        }
    }

    // Método para convertir 4 bytes a un entero (Big Endian)
    private static int bytesToInt(byte[] bytes) {
        return ((bytes[0] & 0xFF) << 24) |
            ((bytes[1] & 0xFF) << 16) |
            ((bytes[2] & 0xFF) << 8) |
            (bytes[3] & 0xFF);
    }

    // Método para convertir 4 bytes a un float (Big Endian)
    private static float bytesToFloat(byte[] bytes) {
        int intBits = bytesToInt(bytes);
        return Float.intBitsToFloat(intBits);
    }

    // Método para convertir 8 bytes a un double (Big Endian)
    private static double bytesToDouble(byte[] bytes) {
        long longBits = ((long) (bytes[0] & 0xFF) << 56) |
            ((long) (bytes[1] & 0xFF) << 48) |
            ((long) (bytes[2] & 0xFF) << 40) |
            ((long) (bytes[3] & 0xFF) << 32) |
            ((long) (bytes[4] & 0xFF) << 24) |
            ((long) (bytes[5] & 0xFF) << 16) |
            ((long) (bytes[6] & 0xFF) << 8) |
            (bytes[7] & 0xFF);
    }
}
```



```

        return Double.longBitsToDouble(longBits);
    }
}

```

**Ejemplo:** crear una copia de un archivo de imagen.

Nota: el método `readAllBytes` de `BufferedInputStream` lee todo el contenido del flujo de entrada en un arreglo de bytes. Dado que el tamaño máximo del arreglo es de aproximadamente 2 GB (tamaño máximo teórico de un arreglo es de `Integer.MAX_VALUE`), se puede usar `readAllBytes` para leer archivos de imagen hasta ese tamaño, siempre y cuando se tenga suficiente memoria disponible para almacenar el arreglo.

```

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopiarImagenEjemplo {
    public static void main(String[] args) {
        String srcFile = "foto.jpg";
        String destFile1 = "foto1.jpg";
        String destFile2 = "foto2.jpg";

        try (BufferedInputStream bis = new BufferedInputStream(new FileInputStream(srcFile));
            BufferedOutputStream bos1 = new BufferedOutputStream(new FileOutputStream(destFile1));
            BufferedOutputStream bos2 = new BufferedOutputStream(new FileOutputStream(destFile2))) {
            // Usando readAllBytes para copiar la imagen a foto1.jpg
            byte[] imageBytes = bis.readAllBytes();
            bos1.write(imageBytes);
            System.out.println("La imagen ha sido copiada usando readAllBytes.");

            // Volver al inicio del flujo para la segunda operación
            bis.close(); // Cerrar el primer flujo
            // Reabrir el flujo para la segunda operación
            try (BufferedInputStream bis2 = new BufferedInputStream(new FileInputStream(srcFile))) {
                bis2.transferTo(bos2);
            }
            System.out.println("La imagen ha sido copiada usando transferTo.");
        } catch (IOException e) {
            System.err.println("Ocurrió un error durante la copia de la imagen: " + e.getMessage());
        }
    }
}

```

### 6.2.3 Datos primitivos y tipos de datos estructurados.

La clase `DataInputStream` facilita la **lectura de datos primitivos y cadenas de caracteres en un formato específico desde un archivo binario**. Permite leer tipos de datos de *Java* en su formato binario correspondiente, lo que simplifica la recuperación de datos estructurados.

```

java.lang.Object
java.io.InputStream
java.io.FilterInputStream
java.io.DataInputStream

```

**Constructores y métodos** comunes de `DataInputStream`:

Constructor/Método	Descripción
<code>DataInputStream(InputStream in)</code>	Crea un <code>DataInputStream</code> para leer datos primitivos desde un flujo de entrada dado ( <code>InputStream</code> ).
<code>int read(byte[] b)</code>	Lee bytes del flujo de entrada y los almacena en el array <code>b</code> . Devuelve el número de bytes leídos.
<code>boolean readBoolean()</code>	Lee un valor booleano desde el flujo de entrada.
<code>byte readByte()</code>	Lee un byte desde el flujo de entrada.
<code>char readChar()</code>	Lee un carácter Unicode de 2 bytes del flujo de entrada.
<code>double readDouble()</code>	Lee un valor de punto flotante de 8 bytes del flujo de entrada.
<code>float readFloat()</code>	Lee un valor de punto flotante de 4 bytes del flujo de entrada.
<code>void readFully(byte[] b)</code>	Lee bytes del flujo de entrada y los almacena en el array <code>b</code> hasta que se lea la longitud completa del array.
<code>void readFully(byte[] b, int off, int len)</code>	Lee bytes del flujo de entrada y los almacena en el array <code>b</code> , comenzando en la posición <code>off</code> y leyendo <code>len</code> bytes.
<code>int readInt()</code>	Lee un entero de 4 bytes del flujo de entrada.

<code>long readLong()</code>	Lee un entero largo de 8 bytes del flujo de entrada.
<code>short readShort()</code>	Lee un entero corto de 2 bytes del flujo de entrada.
<code>String readUTF()</code>	Lee una cadena de texto en formato UTF-8 del flujo de entrada.
<code>int skipBytes(int n)</code>	Salta <code>n</code> bytes en el flujo de entrada.
<code>final int readUnsignedByte()</code>	Lee un byte sin signo del flujo de entrada.
<code>final int readUnsignedShort()</code>	Lee un entero corto sin signo desde el flujo de entrada.
<code>int available()</code>	Devuelve el número de bytes que se pueden leer sin bloquearse en el archivo.
<code>void close()</code>	Cierra el <code>DataInputStream</code> .

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/DataInputStream.html>

**Ejemplo:**

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class EjemploDataInputStream {
    public static void main(String[] args) {
        try (DataInputStream di = new DataInputStream(new FileInputStream("datos.bin"))) {
            boolean valorBooleano = di.readBoolean(); // Leer y mostrar un valor booleano
            System.out.println("Valor booleano: " + valorBooleano);

            int valorEntero = di.readInt(); // Leer y mostrar un valor entero
            System.out.println("Valor entero: " + valorEntero);

            float valorFlotante = di.readFloat(); // Leer y mostrar un valor de punto flotante
            System.out.println("Valor de punto flotante: " + valorFlotante);

            String cadenaTexto = di.readUTF(); // Leer y mostrar una cadena de texto (UTF-8)
            System.out.println("Cadena de texto: " + cadenaTexto);
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        }
    }
}
```



## TAREA

9. Crear una clase Estudiante con los atributos: nombre (String), edad (int) y calificaciones (un array de double). Escribir un programa que cree objetos de tipo Estudiante y almacene sus datos en un archivo binario utilizando la clase `DataOutputStream`. El programa también debe leer los objetos Estudiante desde el archivo y mostrar sus datos en la pantalla. **Nota:** no se deben utilizar `ObjectOutputStream` ni `ObjectInputStream`.
10. Crear dos archivos binarios distintos, cada uno con diferentes tipos de datos escritos usando `FileOutputStream` y/o `DataOutputStream`. Escribir un programa que lea ambos archivos y combine sus datos en un tercer archivo binario. **Nota:** se valorará que el programa cree los archivos de entrada por sí mismo.
11. (Opcional) Crear una clase Factura que represente una factura con los siguientes atributos: número de factura (String), fecha (String), lista de productos (una lista de objetos Producto), e importe total (double). Escribir un programa que permita al usuario ingresar la información de una factura, incluyendo la lista de productos, y guarde esta factura en un archivo binario. El programa también debe recuperar la factura desde el archivo y mostrar los detalles en la pantalla. **Nota:** no se deben utilizar `ObjectOutputStream` ni `ObjectInputStream`.

## 7 Ficheros de configuración.

En *Java*, se utilizan librerías específicas para manejar ficheros de configuración, siguiendo un **formato estándar**. La librería se encarga de leer y escribir en el fichero, permitiendo al programador acceder o modificar propiedades sin preocuparse por el manejo de archivos directamente.

Un fichero de configuración tiene un **formato de texto** como este:

```
# Ejemplo Properties
```

```
# Mon Mar 24 20:50:56 CET 2023
lastAccess=1395690656760
language=ES
```

Los **comentarios comienzan con #**, y cada propiedad se define con el formato **clave=valor**. Las claves deben ser únicas en el fichero.

La **clase Properties** (del paquete `java.util`) permite manipular estos ficheros de configuración. Los ficheros pueden ser también en formato XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>Ejemplo Properties</comment>
  <entry key="lastAccess">1395690656760</entry>
  <entry key="language">ES</entry>
</properties>
```

Estos archivos **pueden ser generados directamente desde una aplicación Java** o con un editor de textos simple, para luego leer la información que contienen desde la aplicación.

**Constructores y métodos** importantes de la clase `Properties`:

Constructor/Método	Descripción
<code>Properties()</code>	Crea un objeto <code>Properties</code> vacío.
<code>Object setProperty(String key, String value)</code>	Agrega o modifica una propiedad.
<code>String getProperty(String key)</code>	Obtiene el valor de una clave.
<code>String getProperty(String key, String defaultValue)</code>	Obtiene el valor de una clave o un valor predeterminado si no existe.
<code>void load(InputStream inStream)</code>	Lee propiedades desde un flujo de entrada.
<code>void loadFromXML(InputStream in)</code>	Lee propiedades desde un documento XML.
<code>void store(OutputStream out, String comments)</code>	Escribe las propiedades en un flujo de salida con comentarios.
<code>void storeToXML(OutputStream os, String comment)</code>	Escribe las propiedades en un documento XML.
<code>Set&lt;String&gt; stringPropertyNames()</code>	Devuelve un conjunto de todas las claves.
<code>Object remove(Object key)</code>	Elimina una propiedad.
<code>boolean containsKey(Object key)</code>	Verifica si una clave existe.
<code>boolean containsValue(Object value)</code>	Verifica si un valor existe.
<code>Enumeration&lt;String&gt; keys()</code>	Devuelve todas las claves.
<code>void clear()</code>	Elimina todas las propiedades.
<code>int size()</code>	Devuelve el número de propiedades.
<code>boolean isEmpty()</code>	Verifica si el objeto está vacío.

#### Consideraciones:

- ✓ **Fechas:** deberán ser **convertidas a texto** cuando se quieran guardar y nuevamente reconvertidas a `Date` cuando se lean del fichero para usarlas.
- ✓ **boolean:** usar `String.valueOf(boolean)` para convertir a texto y `Boolean.parseBoolean(String)` para leer.
- ✓ **Números:** usar `Integer.parseInt(String)`, `Float.parseFloat(String)` y `Double.parseDouble(String)` para convertir de texto a números.

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Properties.html>

**Ejemplo:**

```
import java.util.Properties;

public class PropertiesEjemplo {
    public static void main(String[] args) {
        Properties properties = new Properties(); // Crear un objeto Properties

        // Agregar propiedades
        properties.setProperty("nombre", "Juan");
        properties.setProperty("edad", "30");
        properties.setProperty("email", "juan@ejemplo.com");
    }
}
```

```

// Obtener y mostrar una propiedad
String nombre = properties.getProperty("nombre");
System.out.println("Nombre: " + nombre);

// Obtener una propiedad con valor predeterminado
String ciudad = properties.getProperty("ciudad", "Ciudad desconocida");
System.out.println("Ciudad: " + ciudad);

// Mostrar todas las propiedades
System.out.println("\nPropiedades:");
properties.forEach((key, value) -> System.out.println(" - " + key + " = " + value));
}
}

```

**Ejemplo:**

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

public class PropertiesEjemplo2 {
    public static void main(String[] args) {
        Properties properties = new Properties(); // Crear un objeto de la clase Properties

        // Agregar propiedades
        properties.setProperty("url", "jdbc:mysql://localhost:3306/mydb");
        properties.setProperty("username", "myuser");
        properties.setProperty("password", "mypassword");

        // Guardar las propiedades en "config.properties"
        try (FileOutputStream fileOut = new FileOutputStream("config.properties")) {
            properties.store(fileOut, "Configuración de la BD");
            System.out.println("Propiedades guardadas en el archivo.");
        } catch (IOException e) {
            System.err.println("Error al guardar las propiedades: " + e.getMessage());
        }

        // Recuperar las propiedades del archivo
        try (FileInputStream fileIn = new FileInputStream("config.properties")) {
            properties.load(fileIn);
            // Mostrar propiedades recuperadas
            System.out.println("\nPropiedades recuperadas del archivo:");
            System.out.println("URL de la BD: " + properties.getProperty("url"));
            System.out.println("Nombre de usuario: " + properties.getProperty("username"));
            System.out.println("Contraseña: " + properties.getProperty("password"));
        } catch (IOException e) {
            System.err.println("Error al cargar las propiedades: " + e.getMessage());
        }
    }
}

```



## TAREA

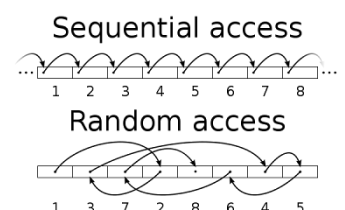
12. Crear un programa que lea un archivo de propiedades llamado "config.properties" y muestre en pantalla todos los pares clave-valor que contiene. Además, debe solicitar al usuario que ingrese una clave y un nuevo valor, para actualizar el valor correspondiente a esa clave en el archivo y guardar los cambios. Nota: comprobar si el archivo "config.properties" existe antes de intentar leerlo.

## 8 Acceso directo a ficheros.

La clase `RandomAccessFile` permite el acceso directo a cualquier posición de un archivo. Se pueden abrir archivos en modo lectura ("r") o en modo lectura y escritura ("rw"), especificándolo el modo en el segundo parámetro del constructor.

`RandomAccessFile` trata el archivo como un arreglo de bytes numerado desde cero, no como flujos. El puntero del archivo indica la posición actual para leer o escribir. Después de cada operación, el puntero se mueve automáticamente a la siguiente posición. Si se escribe más allá del final del archivo, este se extiende. Se puede leer y establecer la posición del puntero con `getFilePointer()` y `seek(pos)`.

`java.lang.Object`  
`java.io.RandomAccessFile`



`RandomAccessFile` es útil principalmente para archivos binarios y no es la opción más común para archivos de texto. Para almacenar **objetos**, se debe **convertirlos a un array de bytes mediante serialización**.

Al leer, si se alcanza el final del archivo antes de leer todos los bytes deseados, se lanza una `EOFException` (un tipo de `IOException`). Si no se puede leer un byte por otras razones, se lanza una `IOException`, como cuando el archivo está cerrado.

Modificar datos con `RandomAccessFile` requiere precaución para evitar la corrupción de datos (especialmente si cambia la longitud de los datos). Es importante conocer la estructura del archivo y el formato de los datos.

Constructores y métodos comunes de la clase `RandomAccessFile`:

Tipo	Tamaño en bytes
byte	1
short	2
int	4
long	8
float	4
double	8
boolean	1
char	2

Constructor/Método	Descripción
<code>RandomAccessFile(String name, String mode)</code>	Crea un objeto <code>RandomAccessFile</code> para abrir un archivo con el nombre y modo especificado ("r", "rw").
<code>RandomAccessFile(File file, String mode)</code>	Crea un objeto <code>RandomAccessFile</code> para el archivo representado por el objeto <code>File</code> (modos "r" o "rw").
<code>void write(byte[] b)</code>	Escribe un array de bytes.
<code>void writeBoolean(boolean v)</code>	Escribe un valor booleano (1 byte).
<code>void writeByte(int v)</code>	Escribe un byte.
<code>void writeChar(int v)</code>	Escribe un carácter Unicode (2 bytes).
<code>void writeChars(String s)</code>	Escribe una cadena.
<code>void writeDouble(double v)</code>	Escribe un valor de punto flotante (8 bytes).
<code>void writeFloat(float v)</code>	Escribe un valor de punto flotante (4 bytes).
<code>void writeInt(int v)</code>	Escribe un entero (4 bytes).
<code>void writeLong(long v)</code>	Escribe un entero largo (8 bytes).
<code>void writeShort(int v)</code>	Escribe un entero corto (2 bytes).
<code>void writeUTF(String str)</code>	Escribe una cadena usando codificación UTF-8 (independiente de la máquina).
<code>int read(byte[] b)</code>	Lee bytes en el array proporcionado <code>b</code> . Devuelve la cantidad de bytes leídos o -1 si se llega al final.
<code>boolean readBoolean()</code>	Lee un valor booleano.
<code>byte readByte()</code>	Lee un byte.
<code>char readChar()</code>	Lee un carácter Unicode (2 bytes).
<code>double readDouble()</code>	Lee un valor de punto flotante (8 bytes).
<code>float readFloat()</code>	Lee un valor de punto flotante (4 bytes).
<code>int readInt()</code>	Lee un entero (4 bytes).
<code>String readLine()</code>	Lee una línea del archivo. Devuelve null si se llega al final.
<code>long readLong()</code>	Lee un entero largo (8 bytes).
<code>short readShort()</code>	Lee un entero corto de (2 bytes).
<code>String readUTF()</code>	Lee una cadena.
<code>long getFilePointer()</code>	Devuelve la posición actual del puntero.
<code>void seek(long pos)</code>	Establece la posición del puntero (pos medido en bytes).
<code>int skipBytes(int n)</code>	Intenta avanzar el puntero <code>n</code> bytes desde la posición actual.
<code>long length()</code>	Devuelve la longitud (tamaño) del archivo en bytes.
<code>void close()</code>	Cierra el <code>RandomAccessFile</code> .

Más información en <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/RandomAccessFile.html>

Para **organizar un archivo** de datos con `RandomAccessFile`, se pueden seguir los siguientes pasos:

1. **Encabezado con el total de registros:** al principio del archivo, escribir un encabezado que contenga el número total de registros. Esto permitirá saber cuántos registros hay sin necesidad de recorrer todo el archivo.
2. **Estructura de cada registro:** definir una estructura fija para cada registro con campos específicos.
3. **Escritura de registros:** cuando se agregue un nuevo registro, primero actualizar el total de registros en el

encabezado. Luego, mover el puntero al final del archivo y escribir los datos del nuevo registro.

```
// Escribir el total de registros al principio del archivo
archivo.seek(0);
archivo.writeInt(totalRegistros);

// Agregar un nuevo registro al final
archivo.seek(archivo.length()); // Posición al final
archivo.writeInt(id);
archivo.writeUTF(nombre);
archivo.writeInt(edad);
// ...
```

4. **Lectura de registros:** para leer los registros, primero leer el total de registros desde el encabezado. Luego, recorrer los registros uno por uno, leyendo los campos en el orden en que se escribieron.

```
archivo.seek(0); // Posición al principio
int totalRegistros = archivo.readInt();
for (int i = 0; i < totalRegistros; i++) {
    int id = archivo.readInt();
    String nombre = archivo.readUTF();
    int edad = archivo.readInt();
    // Procesar los datos del registro
}
```

5. **Modificar o eliminar registros:**

- Crear un nuevo archivo temporal.
- Leer todos los registros del archivo original, excepto el que se desea modificar o eliminar, y escribirlos en el archivo temporal. Actualizar el total de registros restando uno.

```
String archivoNombre = "miarchivo.dat";
String archivoTemporalNombre = "miarchivo_temp.dat";
int registroAEliminar = 3; // ID del registro que se desea eliminar
try (RandomAccessFile archivo = new RandomAccessFile(archivoNombre, "rw");
    RandomAccessFile archivoTemporal = new RandomAccessFile(archivoTemporalNombre, "rw")) {
    int totalRegistros = archivo.readInt(); // Leer el total de registros
    // Copiar registros excepto el que se desea eliminar o modificar
    for (int i = 0; i < totalRegistros; i++) {
        int id = archivo.readInt();
        String nombre = archivo.readUTF();
        int edad = archivo.readInt();
        if (id != registroAEliminar) {
            archivoTemporal.writeInt(id);
            archivoTemporal.writeUTF(nombre);
            archivoTemporal.writeInt(edad);
        }
    }
    // Actualizar el total de registros en el archivo temporal
    archivoTemporal.seek(0);
    archivoTemporal.writeInt(totalRegistros - 1);
    // Renombrar el archivo temporal al nombre original
    File archivoOriginal = new File(archivoNombre);
    File archivoTemporal = new File(archivoTemporalNombre);
    archivoTemporal.renameTo(archivoOriginal);
    System.out.println("Archivo reconstruido correctamente.");
} catch (IOException e) {
    System.out.println("Error al reestructurar el archivo.");
}
```

Se puede utilizar un **enfoque de registros de longitud fija**. Cada registro tiene un tamaño constante y contiene información específica.

**Ideal para estructuras de datos donde los campos tienen una longitud predefinida** (por ejemplo, registros de empleados con campos como nombre, salario, fecha de ingreso, etc.).

**Para acceder a un registro específico, se calcula la posición en bytes multiplicando el número de registros por la longitud de cada registro.**

- Si se modifica un registro, agregar el registro modificado al final del archivo temporal. Sumar uno al total de registros
- Reemplazar el archivo original con el archivo temporal.

**Ejemplo:**

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class EjemploRandomAccessFile {
    public static void main(String[] args) {
        String archivo = "archivo.dat";

        try (RandomAccessFile raFile = new RandomAccessFile(archivo, "rw")) {
            // Escribir datos
```



```

        raFile.writeBoolean(true);
        raFile.writeInt(42);
        raFile.writeDouble(3.14159);

        // Leer datos
        raFile.seek(0); // Volver al inicio del archivo
        boolean valorBooleano = raFile.readBoolean();
        int valorEntero = raFile.readInt();
        double valorDouble = raFile.readDouble();

        // Mostrar los datos leídos
        System.out.println("Valor booleano: " + valorBooleano);
        System.out.println("Valor entero: " + valorEntero);
        System.out.println("Valor double: " + valorDouble);
    } catch (IOException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
}

```

**Ejemplo:**

```

import java.io.IOException;
import java.io.RandomAccessFile;

public class ModificarDatosEnArchivo {
    public static void main(String[] args) {
        String archivo = "archivo.dat";

        try (RandomAccessFile raFile = new RandomAccessFile(archivo, "rw")) {
            // Escribir datos iniciales
            raFile.writeBoolean(true);
            raFile.writeInt(42);
            raFile.writeDouble(3.14159);

            // Leer y mostrar datos iniciales
            raFile.seek(0); // Volver al inicio del archivo
            boolean valorBooleano = raFile.readBoolean();
            int valorEntero = raFile.readInt();
            double valorDouble = raFile.readDouble();

            System.out.println("Datos iniciales:");
            System.out.println("Valor booleano: " + valorBooleano);
            System.out.println("Valor entero: " + valorEntero);
            System.out.println("Valor double: " + valorDouble);

            // Modificar el valor entero
            raFile.seek(1); // Mover al inicio del valor entero (booleano ocupa 1 byte)
            raFile.writeInt(100); // Modificar el valor entero

            // Agregar texto al final del archivo
            raFile.seek(raFile.length()); // Mover al final del archivo
            String nuevoTexto = "Texto agregado";
            raFile.writeUTF(nuevoTexto);

            // Leer y mostrar datos actualizados
            raFile.seek(0); // Volver al inicio del archivo
            valorBooleano = raFile.readBoolean();
            valorEntero = raFile.readInt(); // Leer el valor entero modificado
            valorDouble = raFile.readDouble();
            String textoAgregado = raFile.readUTF(); // Leer el texto agregado

            System.out.println("\nDatos después de modificar el entero y agregar texto:");
            System.out.println("Valor booleano: " + valorBooleano);
            System.out.println("Valor entero actualizado: " + valorEntero);
            System.out.println("Valor double: " + valorDouble);
            System.out.println("Texto agregado: " + textoAgregado);
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

```

**Ejemplo:** convertir todas las “b” de un documento de texto a mayúsculas.

```

import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

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

```

```

try (RandomAccessFile archivo = new RandomAccessFile("archivo.txt", "rw")) {
    System.out.println("El tamaño del archivo es: " + archivo.length());
    byte b;
    while (true) {
        try {
            b = archivo.readByte();
            if (b == 'b') {
                archivo.seek(archivo.getFilePointer() - 1);
                archivo.writeByte('B');
            }
        } catch (EOFException e) {
            System.out.println("Todas las 'b' han sido convertidas a mayúsculas.");
            break;
        }
    }
} catch (FileNotFoundException e) {
    System.out.println("El archivo no existe.");
} catch (IOException e) {
    System.out.println("Se produjo un error con el archivo: " + e.getMessage());
}
}

```

**Ejemplo:** programa para realizar operaciones CRUD (crear, leer, actualizar y eliminar) sobre registros de empleados (id, nombre, apellidos, departamento y sueldo) usando la clase `RandomAccessFile` para el almacenamiento de datos.

El programa usa una lista de empleados en memoria en lugar de guardar cada empleado individualmente en el archivo. Esta estrategia mejora la eficiencia y facilita la manipulación de datos, siendo ideal para aplicaciones de tamaño pequeño o mediano.

El programa también utiliza un archivo de configuración llamado `config.properties`. Si el archivo existe, se lee para obtener el nombre del archivo de datos donde se guardarán los empleados. Si no existe, se utiliza el nombre de archivo predeterminado "empleados.dat". El archivo de configuración debe tener el siguiente formato:

```

# Nombre del fichero donde guardar los empleados
dataFileName=nombre_archivo.extension

```

#### ➤ GestionEmpleados.java

```

import java.io.*;
import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import java.util.Scanner;

class Utiles {
    public static final String ROJO = "\033[31m";
    public static final String VERDE = "\033[32m";
    public static final String AZUL = "\033[34m";
    public static final String MORADO = "\033[35m";
    public static final String RESET = "\033[0m";
    public static final String LIMPIAR_PANTALLA = "\033[2J";
    private static final Scanner SCANNER = new Scanner(System.in);

    public static int leerEnteroValido(String texto) {
        int numero;
        while (true) {
            System.out.print(texto);
            if (SCANNER.hasNextInt()) {
                numero = SCANNER.nextInt();
                SCANNER.nextLine(); // Consumir la nueva línea
                return numero;
            } else {
                System.out.println("Valor no válido. Por favor, ingresa un número entero.");
                SCANNER.nextLine(); // Limpiar el búfer de entrada
            }
        }
    }

    public static double leerDoubleValido(String mensaje) {
        double numero;
        while (true) {
            System.out.print(mensaje);
            if (SCANNER.hasNextDouble()) {
                numero = SCANNER.nextDouble();
                SCANNER.nextLine(); // Consumir la nueva línea
                return numero;
            }
        }
    }
}

```

```

    } else {
        System.out.println("Valor no válido. Por favor, ingresa un número decimal.");
        SCANNER.nextLine(); // Limpiar el búfer de entrada
    }
}

public static String leerTextoConLongitudMaxima(String mensaje, int longitudMaxima) {
    String texto;
    while (true) {
        System.out.print(mensaje);
        texto = SCANNER.nextLine();
        if (texto.isEmpty())
            System.out.println("El texto no puede dejarse vacío. Intenta de nuevo.");
        else if (texto.length() > longitudMaxima)
            System.out.println("El texto tiene más de " + longitudMaxima + " caracteres. Intenta de nuevo.");
        else
            return texto;
    }
}

public static void pulseIntroParaContinuar() {
    System.out.println("Presione INTRO para continuar...");
    SCANNER.nextLine(); // Esperar a que el usuario presione Enter
    System.out.print(LIMPIAR_PANTALLA); // Limpiar la pantalla
    System.out.flush();
}

public static String cargarConfiguracion(String configFileName) {
    File f = new File(configFileName);
    if (!f.exists())
        return null;

    Properties properties = new Properties();
    try (FileInputStream fis = new FileInputStream(configFileName)) {
        properties.load(fis);
        return properties.getProperty("dataFileName");
    } catch (IOException e) {
        System.out.println(ROJO + "Error al cargar la configuración: " + e.getMessage() + RESET);
        return null;
    }
}

class Empleado implements Serializable {
    private int id;
    private String nombre, apellidos, departamento;
    private double sueldo;

    public Empleado(int id, String nombre, String apellidos, String departamento, double sueldo) {
        this.id = id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.departamento = departamento;
        this.sueldo = sueldo;
    }

    public int getId() { return id; }
    public String getNombre() { return nombre; }
    public String getApellidos() { return apellidos; }
    public String getDepartamento() { return departamento; }
    public double getSueldo() { return sueldo; }

    @Override
    public String toString() {
        String format = Utiles.AZUL + "|" + Utiles.VERDE + " %-10s " + Utiles.AZUL + "|" + Utiles.VERDE + " %-15s "
            + Utiles.AZUL + "|" + Utiles.VERDE + " %-20s " + Utiles.AZUL + "|" + Utiles.VERDE + " %-15s "
            + Utiles.AZUL + "|" + Utiles.VERDE + " %-15s " + Utiles.AZUL + "|" + Utiles.RESET;
        return String.format(format, id, nombre, apellidos, departamento, sueldo);
    }

    public static void mostrarCabeceraListado(String texto) {
        System.out.println(Utiles.AZUL + texto);
        System.out.println("IDNombreApellidosDepartamentoSueldo");
        System.out.println(Utiles.RESET);
    }

    public static void mostrarFinallistado() {
        System.out.println(Utiles.AZUL +
            "IDNombreApellidosDepartamentoSueldo" + Utiles.RESET);
    }
}

class EmpleadosDAO {
    private List<Empleado> empleados;
    private String archivo;

```

```

public EmpleadosDAO(String configFileName) {
    archivo = Utiles.cargarConfiguracion(configFileName);
    if (archivo == null)
        archivo = "empleados.dat";
    empleados = new ArrayList<>();
    File f = new File(archivo);
    if (!f.exists()) {
        try (RandomAccessFile raf = new RandomAccessFile(archivo, "rw")) {
            raf.writeInt(0);
        } catch (IOException e) {
            System.out.println("Error al crear el archivo: " + e.getMessage());
        }
    } else
        cargarEmpleadosDesdeArchivo();
}

public void crearEmpleado(Empleado empleado) {
    empleados.add(empleado);
    guardarEmpleadosEnArchivo();
}

public List<Empleado> leerEmpleados() {
    return new ArrayList<>(empleados);
}

public Empleado leerEmpleadoPorID(int id) {
    return empleados.stream()
        .filter(empleado -> empleado.getId() == id)
        .findFirst()
        .orElse(null);
}

public void actualizarEmpleado(Empleado empleadoActualizado) {
    for (int i = 0; i < empleados.size(); i++) {
        if (empleados.get(i).getId() == empleadoActualizado.getId()) {
            empleados.set(i, empleadoActualizado);
            guardarEmpleadosEnArchivo();
            return;
        }
    }
}

public void eliminarEmpleado(int id) {
    empleados.removeIf(empleado -> empleado.getId() == id);
    guardarEmpleadosEnArchivo();
}

private void cargarEmpleadosDesdeArchivo() {
    try (RandomAccessFile raf = new RandomAccessFile(archivo, "r")) {
        int numEmpleados = raf.readInt();
        for (int i = 0; i < numEmpleados; i++) {
            int id = raf.readInt();
            String nombre = raf.readUTF();
            String apellidos = raf.readUTF();
            String departamento = raf.readUTF();
            double sueldo = raf.readDouble();
            empleados.add(new Empleado(id, nombre, apellidos, departamento, sueldo));
        }
    } catch (IOException e) {
        System.out.println(Utiles.ROJO + "Error al cargar empleados: " + e.getMessage() + Utiles.RESET);
    }
}

private void guardarEmpleadosEnArchivo() {
    File archivoFile = new File(archivo);
    if (archivoFile.exists())
        archivoFile.delete();

    try (RandomAccessFile raf = new RandomAccessFile(archivo, "rw")) {
        raf.writeInt(empleados.size());
        for (Empleado empleado : empleados) {
            raf.writeInt(empleado.getId());
            raf.writeUTF(empleado.getNombre());
            raf.writeUTF(empleado.getApellidos());
            raf.writeUTF(empleado.getDepartamento());
            raf.writeDouble(empleado.getSueldo());
        }
    } catch (IOException e) {
        System.out.println(Utiles.ROJO + "Error al guardar empleados: " + e.getMessage() + Utiles.RESET);
    }
}

}

public class GestionEmpleados {
    public static void main(String[] args) {
        EmpleadosDAO dao = new EmpleadosDAO("config.properties");
    }
}

```

```

int opcion;
boolean primerIngreso = true;
do {
    if (!primerIngreso)
        Utiles.pulseIntroParaContinuar();
    else {
        System.out.print(Utiles.LIMPIAR_PANTALLA);
        System.out.flush();
        primerIngreso = false;
    }

    mostrarMenu();
    opcion = Utiles.leerEnteroValido("Seleccione una opción: ");

    switch (opcion) {
        case 1 -> crearEmpleado(dao);
        case 2 -> leerTodosLosEmpleados(dao);
        case 3 -> leerEmpleadoPorID(dao);
        case 4 -> actualizarEmpleado(dao);
        case 5 -> eliminarEmpleado(dao);
        case 0 -> System.out.println(Utiles.AZUL + "Saliendo del programa." + Utiles.RESET);
        default -> System.out.println(Utiles.ROJO + "Opción no válida. Intente de nuevo." + Utiles.RESET);
    }
} while (opcion != 0);
}

private static void mostrarMenu() {
    System.out.println(Utiles.MORADO);
    System.out.println("
    _____ Menú de Empleados _____
    ");
    System.out.println("
    1.- Crear empleado.
    ");
    System.out.println("
    2.- Mostrar todos los empleados.
    ");
    System.out.println("
    3.- Buscar empleado por ID.
    ");
    System.out.println("
    4.- Actualizar empleado.
    ");
    System.out.println("
    5.- Borrar empleado.
    ");
    System.out.println("
    0.- Salir.
    ");
    System.out.println("
    ");
    System.out.print(Utiles.RESET);
}

private static void crearEmpleado(EmpleadosDAO dao) {
    int id = Utiles.leerEnteroValido("Ingresa ID: ");
    String nombre = Utiles.leerTextoConLongitudMaxima("Ingresa nombre: ", 15);
    String apellidos = Utiles.leerTextoConLongitudMaxima("Ingresa apellidos: ", 20);
    String departamento = Utiles.leerTextoConLongitudMaxima("Ingresa departamento: ", 15);
    double sueldo = Utiles.leerDoubleValido("Ingresa sueldo: ");

    Empleado empleado = new Empleado(id, nombre, apellidos, departamento, sueldo);
    dao.crearEmpleado(empleado);
    System.out.println(Utiles.VERDE + "Empleado creado con éxito." + Utiles.RESET);
}

private static void leerTodosLosEmpleados(EmpleadosDAO dao) {
    List<Empleado> empleados = dao.leerEmpleados();

    if (empleados.isEmpty())
        System.out.println(Utiles.ROJO + "No hay empleados registrados." + Utiles.RESET);
    else {
        Empleado.mostrarCabeceraListado("Lista de Empleados:");
        empleados.forEach(System.out::println);
        Empleado.mostrarFinalListado();
    }
}

private static void leerEmpleadoPorID(EmpleadosDAO dao) {
    int id = Utiles.leerEnteroValido("Ingresa ID del empleado a buscar: ");
    Empleado empleado = dao.leerEmpleadoPorID(id);

    if (empleado != null) {
        Empleado.mostrarCabeceraListado("Empleado encontrado:");
        System.out.println(empleado);
        Empleado.mostrarFinalListado();
    } else
        System.out.println(Utiles.ROJO + "Empleado no encontrado." + Utiles.RESET);
}

private static void actualizarEmpleado(EmpleadosDAO dao) {
    int id = Utiles.leerEnteroValido("Ingresa ID del empleado a actualizar: ");
    Empleado empleadoExistente = dao.leerEmpleadoPorID(id);

    if (empleadoExistente != null) {
        Empleado.mostrarCabeceraListado("Empleado actual:");
        System.out.println(empleadoExistente);
        Empleado.mostrarFinalListado();

        String nuevoNombre = Utiles.leerTextoConLongitudMaxima("Ingresa el nuevo nombre: ", 15);
        String nuevosApellidos = Utiles.leerTextoConLongitudMaxima("Ingresa los nuevos apellidos: ", 20);
        String nuevoDepartamento = Utiles.leerTextoConLongitudMaxima("Ingresa el nuevo departamento: ", 15);
        double nuevoSueldo = Utiles.leerDoubleValido("Ingresa nuevo Sueldo: ");
    }
}

```

```

Empleado empleadoActualizado = new Empleado(id, nuevoNombre, nuevosApellidos, nuevoDepartamento, nuevoSueldo);
dao.actualizarEmpleado(empleadoActualizado);
System.out.println(Utiles.VERDE + "Empleado actualizado con éxito." + Utiles.RESET);
} else
    System.out.println(Utiles.ROJO + "Empleado no encontrado." + Utiles.RESET);
}
private static void eliminarEmpleado(EmpleadosDAO dao) {
    int id = Utiles.leerEnteroValido("Ingresa el ID del empleado a eliminar: ");
    Empleado empleadoExistente = dao.leerEmpleadoPorID(id);
    if (empleadoExistente != null) {
        dao.eliminarEmpleado(id);
        System.out.println(Utiles.VERDE + "Empleado eliminado con éxito." + Utiles.RESET);
    } else
        System.out.println(Utiles.ROJO + "Empleado no encontrado." + Utiles.RESET);
}
}

```

**Nota 1:** cuando se actualizan registros en un archivo de acceso aleatorio y se cambia el tamaño de los campos, pueden surgir problemas de desplazamiento debido a las diferencias en el tamaño de los datos, lo que puede provocar errores de lectura y escritura. En el ejemplo, se soluciona este problema eliminando y recreando el archivo completo.

Para **evitar problemas cuando los campos del archivo tienen tamaños variables al modificarlos**, puede usarse la **clase StringBuffer** para **asegurar que los datos escritos tengan un tamaño fijo**. Ejemplo de cómo hacerlo al modificar un empleado en el programa:

```

// Utilizar StringBuffer para garantizar que los campos tengan el mismo tamaño
StringBuffer sbNombre = new StringBuffer(15);
sbNombre.setLength(15);
sbNombre.replace(0, nuevoNombre.length(), nuevoNombre);
StringBuffer sbApellidos = new StringBuffer(20);
sbApellidos.setLength(20);
sbApellidos.replace(0, nuevosApellidos.length(), nuevosApellidos);
StringBuffer sbDepartamento = new StringBuffer(15);
sbDepartamento.setLength(15);
sbDepartamento.replace(0, nuevoDepartamento.length(), nuevoDepartamento);

```

Si el nuevo valor es más corto, se rellena con espacios en blanco; si es más largo, se trunca para ajustarse al tamaño fijo. Para saltar de un registro a otro, se debe avanzar 112 bytes en la posición del cursor, que corresponde al tamaño del registro (4 bytes para el ID + 30 bytes para el nombre + 40 bytes para el apellido + 30 bytes para el departamento + 8 bytes para el sueldo).

Si **todos los campos de texto tienen el mismo tamaño** (utilizando StringBuffer) **y las modificaciones no cambian la longitud de los datos**, no es necesario recrear todo el archivo al modificar un registro específico. Se puede actualizar directamente el registro en su posición sin afectar a los demás.

**Nota 2:** en lugar de eliminar físicamente un registro del archivo, se puede implementar un **borrado lógico**, que **marca el registro como "eliminado" o "inactivo" sin eliminarlo realmente**. Esto es útil para mantener un historial de registros o para permitir la recuperación de datos eliminados en el futuro. El ejemplo proporcionado implementa un borrado físico, que requiere reconstruir el archivo para eliminar el registro y evitar espacios vacíos.



## TAREA

13. Crear un programa que permita al usuario editar una línea específica en un archivo de texto utilizando RandomAccessFile. El programa debe solicitar al usuario el número de línea que desea modificar y el nuevo contenido para esa línea, reemplazando el contenido existente en la línea indicada.
14. Diseñar un sistema de gestión de clientes utilizando la clase RandomAccessFile. Cada cliente tener campos como nombre, CIF, teléfono, e-mail, persona de contacto e importe adeudado. El programa debe permitir agregar, buscar, actualizar y eliminar clientes, además de generar informes según diferentes criterios.

## 9 Persistencia de objetos.

La persistencia de objetos **permite almacenar objetos y sus estados** (valores de sus atributos) **en un medio de almacenamiento** como archivos, BD o memoria **para recuperarlos y usarlos posteriormente**. En Java, hay varias **formas de lograr la persistencia de objetos**:



- ✓ **Serialización en archivos binarios:** convierte objetos en secuencias de bytes que se pueden guardar en un archivo o transmitir a través de la red.
- ✓ **Archivos de texto:** los objetos pueden guardarse en formatos de texto plano (.txt), CSV, XML o JSON.
- ✓ **JPA (Java/Jakarta Persistence API):** permite el mapeo objeto-relacional (ORM) y la persistencia de objetos en BD relacionales.
- ✓ **Hibernate:** framework ORM que facilita la persistencia en BD relacionales.
- ✓ **BD NoSQL:** herramientas como *MongoDB* o *Redis* permiten almacenar objetos en formato JSON o BSON usando bibliotecas específicas.

La elección de la técnica depende de los requisitos de la aplicación y la infraestructura utilizada, cada enfoque tiene sus ventajas y desventajas.

**En este punto se va a tratar la serialización en archivos binarios, en próximas unidades se verán otras formas.**

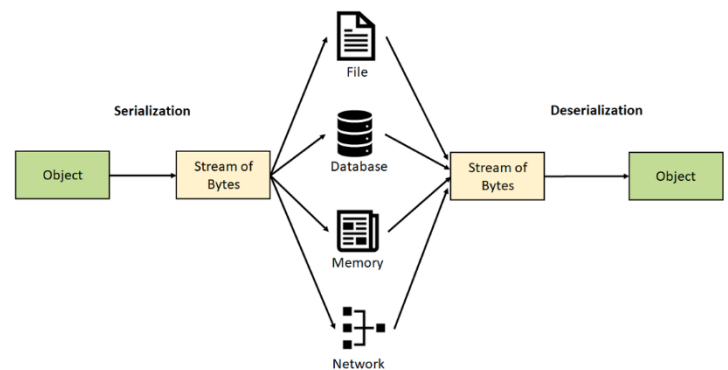
## 9.1 Serialización.

La **serialización** convierte un objeto en una secuencia de bytes para almacenarlo o enviarlo. La **deserialización**, por otro lado, **reconstruye el objeto a partir de esa secuencia**. En *Java*, los objetos deben implementar la interfaz `Serializable` para ser serializables:

```
import java.io.*;
public interface Serializable {}
```

Si un objeto tiene referencias a otros objetos, todos ellos también deben ser serializables. Si no, se lanzará una excepción `NotSerializableException`.

Para que un objeto sea serializable, **todas sus variables de instancia han de ser serializables**. Todos los tipos primitivos en *Java* son serializables por defecto (igual que los arrays y otros muchos tipos estándar).



### 9.1.1 El serialVersionUID.

El `serialVersionUID` es un **campo especial en Java que controla la compatibilidad de versiones al serializar y deserializar objetos**. Cuando un objeto se serializa, *Java* guarda el `serialVersionUID` junto con la secuencia de bytes. Al deserializar, *Java* compara el `serialVersionUID` guardado en el flujo con el de la clase actual. Si no coinciden, se lanza una excepción `InvalidClassException`, indicando que la clase ha cambiado de manera incompatible con la versión previamente serializada.

Este campo actúa como un identificador de versión para la clase, y cualquier cambio en su definición (incluso cambios menores como agregar o eliminar métodos, cambiar la visibilidad de atributos o modificar el orden de los campos) puede alterar el `serialVersionUID` generado automáticamente por *Java*. Esto hace que la clase nueva no sea compatible con las versiones anteriores, lo que puede provocar errores al intentar deserializar objetos antiguos.

Por eso, es **recomendable declarar manualmente el serialVersionUID** en la clase **para evitar estos problemas y mantener el control explícito de la compatibilidad**. Un ejemplo de su declaración sería:

```
private static final long serialVersionUID = 123456789L;
```

Asignar un `serialVersionUID` fijo asegura que pequeños cambios en la clase no alteren la compatibilidad de versiones, permitiendo que la deserialización siga funcionando correctamente, incluso después de actualizar la clase.

## 9.2 Flujos para entrada y salida de objetos.

En *Java*, los flujos de objetos **permiten la entrada y salida de objetos mediante la serialización y deserialización**. La serialización convierte los objetos en una secuencia de bytes para escribirlos en un flujo de salida, y la deserialización reconstruye los objetos a partir de esa secuencia de bytes al leerlos desde un flujo de entrada.

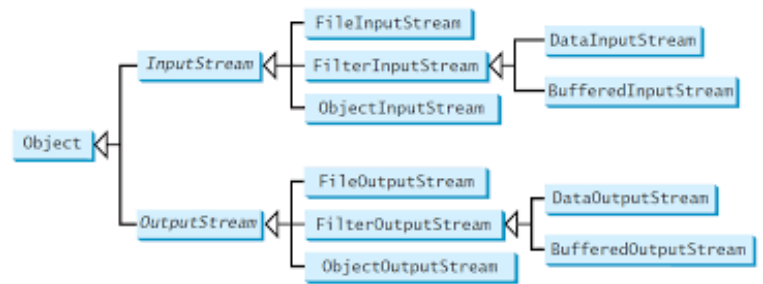
Para **manejar objetos en flujos**, se utilizan `ObjectOutputStream` y `ObjectInputStream`, que son subclases de `OutputStream` e `InputStream`, respectivamente. Estos flujos **proporcionan métodos para leer y escribir objetos**,

devolviendo y recibiendo instancias de la clase `Object`.

Para **serializar un objeto y escribirlo en un flujo**, se utiliza el método `void writeObject(Object o)` de `ObjectOutputStream`.

Para **deserializar un objeto desde un flujo**, se usa el método `Object readObject()` de `ObjectInputStream`, que requiere hacer un cast al tipo de objeto esperado. Ejemplo:

```
Perro p = (Perro) miStream.readObject();
```



**Constructores y métodos importantes de la clase `ObjectOutputStream`:**

Constructor/Método	Descripción
<code>ObjectOutputStream(OutputStream out)</code>	Crea un <code>ObjectOutputStream</code> que escribe objetos en un <code>OutputStream</code> .
<code>ObjectOutputStream(OutputStream out, boolean append)</code>	Crea un <code>ObjectOutputStream</code> que escribe objetos en un <code>OutputStream</code> , con opción de anexar datos al flujo existente.
<code>void writeBoolean(boolean val)</code>	Escribe un valor booleano en el archivo.
<code>void writeByte(int val)</code>	Escribe un byte en el archivo.
<code>void writeChar(int val)</code>	Escribe un carácter Unicode (2 bytes).
<code>void writeDouble(double val)</code>	Escribe un valor de punto flotante (8 bytes).
<code>void writeFloat(float val)</code>	Escribe un valor de punto flotante (4 bytes).
<code>void writeInt(int val)</code>	Escribe un entero (4 bytes).
<code>void writeLong(long val)</code>	Escribe un entero largo (8 bytes).
<code>void writeShort(int val)</code>	Escribe un entero corto (2 bytes).
<code>void writeUTF(String str)</code>	Escribe una cadena usando codificación UTF-8 modificada.
<code>void writeObject(Object obj)</code>	Escribe un objeto serializable.
<code>void write(int val)</code>	Escribe un byte de datos.
<code>void write(byte[] buf)</code>	Escribe un array de bytes.
<code>void flush()</code>	Limpia el búfer y envía cualquier dato pendiente al flujo de salida.
<code>void close()</code>	Cierra el <code>ObjectOutputStream</code> , liberando recursos.

**Constructores y métodos importantes de la clase `ObjectInputStream`:**

Constructor/Método	Descripción
<code>ObjectInputStream(InputStream in)</code>	Crea un <code>ObjectInputStream</code> que lee objetos desde un <code>InputStream</code> .
<code>boolean readBoolean()</code>	Lee un valor booleano.
<code>byte readByte()</code>	Lee un byte.
<code>char readChar()</code>	Lee un carácter Unicode (2 bytes).
<code>double readDouble()</code>	Lee un valor de punto flotante (8 bytes).
<code>float readFloat()</code>	Lee un valor de punto flotante (4 bytes).
<code>int readInt()</code>	Lee un entero (4 bytes).
<code>long readLong()</code>	Lee un entero largo (8 bytes).
<code>short readShort()</code>	Lee un entero corto (2 bytes).
<code>String readUTF()</code>	Lee una cadena usando codificación UTF-8 modificada.
<code>Object readObject()</code>	Lee un objeto serializado. Si se llega al final del archivo se lanza la excepción <code>EOFException</code> .
<code>int read()</code>	Lee un byte de datos.
<code>int read(byte[] buf)</code>	Lee un array de bytes y lo almacena en el búfer especificado.
<code>int available()</code>	Devuelve el número de bytes disponibles para leer sin bloquear.
<code>void close()</code>	Cierra el <code>ObjectInputStream</code> , liberando recursos.
<code>long skip(long n)</code>	Salta <code>n</code> bytes en el flujo de entrada.

**Estrategias para detectar el final del archivo:**

- ✓ **Captura de excepción `EOFException`:** se lanza cuando se alcanza el final del archivo.

```
try {
    while (true) {
        Object obj = ois.readObject();
        // Procesar el objeto leído
    }
} catch (EOFException e) {
    System.out.println("Fin del archivo.");
}
```

Las **colecciones** en *Java* (por ejemplo, `List<>`) también son un objeto, por lo que es posible leer/escribir todos sus elementos en una única operación.

- ✓ **Usar una colección:** almacenar objetos en una lista, escribir la lista completa y leerla de una sola vez.

```
List<MiObjeto> listaObjetos = new ArrayList<>();
listaObjetos.add(objeto1);
listaObjetos.add(objeto2);
// ... Agregar más objetos a la lista
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("miarchivo.dat"));
oos.writeObject(listaObjetos);
oos.close();
// ...
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("miarchivo.dat"));
List<MiObjeto> listaLeida = (List<MiObjeto>) ois.readObject();
ois.close();
for (MiObjeto obj : listaLeida) // Ahora se puede trabajar con la lista leída
    // Procesar cada objeto
```

- ✓ **Contador de objetos:** escribir la cantidad de objetos al inicio del archivo y luego usarla como contador en un bucle.

```
int cantidadObjetos = 10;
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("miarchivo.dat"));
oos.writeInt(cantidadObjetos); // Escribir la cantidad al principio
for (int i = 0; i < cantidadObjetos; i++)
    oos.writeObject(objeto); // Escribir los objetos aquí
oos.close();
// ...
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("miarchivo.dat"));
int cantidadLeida = ois.readInt(); // Leer la cantidad al principio
for (int i = 0; i < cantidadLeida; i++) {
    Object obj = ois.readObject(); // Leer los objetos aquí
    // Procesar el objeto leído
}
ois.close();
```

- ✓ **Escribir null al final:** al escribir objetos, agregar null. Durante la lectura, null indica el fin.

```
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("miarchivo.dat"));
oos.writeObject(objeto1);
oos.writeObject(objeto2);
// ...
oos.writeObject(null); // Escribir null al final
oos.close();
// ...
ObjectInputStream ois = new ObjectInputStream(new FileInputStream("miarchivo.dat"));
try {
    while (true) {
        Object obj = ois.readObject();
        if (obj == null) {
            System.out.println("Fin del archivo.");
            break;
        }
        // Procesar el objeto leído
    }
} catch (EOFException e) {
    System.out.println("Fin del archivo.");
}
ois.close();
```

Más información en

<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/ObjectOutputStream.html> y  
<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/ObjectInputStream.html>

**Ejemplo:**

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

public class EjemploSerial {
    public static void main(String[] args) {
        // Serialización de objetos
        try (FileOutputStream archivo = new FileOutputStream("prueba.dat");
             ObjectOutputStream salida = new ObjectOutputStream(archivo)) {
            salida.writeObject("Hoy es: ");
            salida.writeObject(new Date());
        } catch (IOException e) {
            System.out.println("Error al escribir en el archivo: " + e.getMessage());
        }

        // Deserialización de objetos
        try (FileInputStream archivo = new FileInputStream("prueba.dat");
             ObjectInputStream entrada = new ObjectInputStream(archivo)) {
            String hoy = (String) entrada.readObject();
            Date fecha = (Date) entrada.readObject();
            System.out.println(hoy + fecha);
        } catch (FileNotFoundException e) {
            System.out.println("No se pudo abrir el archivo: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println("Clase no encontrada durante la deserialización: " + e.getMessage());
        }
    }
}
```

Como se puede comprobar en el ejemplo, en un **ObjectStream** pueden coexistir diferentes tipos de datos.

**Ejemplo:** utilizar flujos de objetos para entrada y salida de objetos *Persona*.

➤ *Persona.java*

```
import java.io.Serializable;

public class Persona implements Serializable {
    private static final long serialVersionUID = 123456789L; // Número de versión personalizado
    private String nombre;
    private int edad;

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

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    @Override
    public String toString() {
        return "Persona { " + "nombre = '" + nombre + '\'" + ", edad = " + edad + " }";
    }
}
```

Si se necesita un enfoque de "acceso directo" a los datos serializados en un archivo binario, se podría considerar el uso de soluciones de almacenamiento de objetos como **ObjectDB** o **Db4o**. Ambas, **ObjectDB** (licencia comercial para su versión completa) y **db4o** (de código abierto y es gratuito tanto para proyectos comerciales como no comerciales), son **BD** orientadas a objetos.

➤ *EscrituraDePersonas.java*

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class EscrituraDePersonas {
    public static void main(String[] args) {
        // Serialización de objetos Persona
        try (FileOutputStream fileOut = new FileOutputStream("personas.obj");
             ObjectOutputStream objectOut = new ObjectOutputStream(fileOut)) {
            // Crear y escribir los objetos en el archivo
            objectOut.writeObject(new Persona("Juan", 18));
            objectOut.writeObject(new Persona("Carlos", 22));
        }
    }
}
```

```

        System.out.println("Objetos escritos en el archivo.");
    } catch (IOException e) {
        System.out.println("Error al escribir en el archivo: " + e.getMessage());
    }
}
}

```

➤ **LecturaDePersonas.java**

```

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class LecturaDePersonas {
    public static void main(String[] args) {
        // Deserialización de objetos Persona
        try (FileInputStream fileIn = new FileInputStream("personas.obj");
            ObjectInputStream objectIn = new ObjectInputStream(fileIn)) {
            // Leer los objetos desde el archivo
            Persona persona1 = (Persona) objectIn.readObject();
            Persona persona2 = (Persona) objectIn.readObject();

            // Mostrar los objetos leídos
            System.out.println(persona1);
            System.out.println(persona2);
        } catch (IOException e) {
            System.out.println("Error al leer el archivo: " + e.getMessage());
        } catch (ClassNotFoundException e) {
            System.out.println("Clase no encontrada al deserializar un objeto: " + e.getMessage());
        }
    }
}

```

## 9.3 El modificador transient.

El modificador **transient** se utiliza en la declaración de los campos de una clase para indicar que esos campos **no deben ser serializados**. Es decir, los campos marcados como **transient** se excluyen de la serialización y su valor no se incluye en la representación serializada del objeto.

Los **atributos transitorios** son aquellos que no se deben guardar cuando los objetos se convierten en una **secuencia de bytes**. Para marcar un atributo como transitorio, basta con **declararlo con la palabra clave transient**. Por ejemplo:

```

public class Usuario implements Serializable {
    private String nombre;
    private transient String contrasena; // Atributo transitorio
    // Resto de atributos y métodos...
}

```

**Razones para usar transient:**

- ✓ **Datos temporales o volátiles:** si un campo almacena datos que no son relevantes para la persistencia a largo plazo, marcarlo como **transient** evita que esos datos se serialicen.
- ✓ **Campos calculados o derivados:** si un campo se calcula o deriva de otros campos dentro de la clase y no necesita ser almacenado, se puede marcar como **transient**.
- ✓ **Campos no serializables:** referencias a recursos externos, flujos de entrada/salida o conexiones de red no son serializables y deben marcarse como **transient**.
- ✓ **Información sensible:** para proteger datos sensibles como contraseñas, se recomienda marcarlos como **transient**.

Ejemplo:

➤ **Cliente.java**

```

import java.io.Serializable;

public class Cliente implements Serializable {
    private static final long serialVersionUID = 1L; // Compatibilidad con versiones futuras
    private String nombre;
    private transient String password; // Atributo transitorio

    public Cliente(String nombre, String password) {
        this.nombre = nombre;
    }
}

```

```

        this.password = password;
    }
    @Override
    public String toString() {
        return (password == null ? "(No disponible)" : password) + " - " + nombre;
    }
}

```

➤ SerializacionClientes.java

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializacionClientes {
    private static final String FILE_NAME = "cliente.obj"; // Nombre del archivo

    public static void main(String[] args) {
        Cliente cliente = new Cliente("Juan Moreno", "Dam1234.!");

        // Serialización
        try (ObjectOutputStream salida = new ObjectOutputStream(new FileOutputStream(FILE_NAME))) {
            salida.writeObject("Datos del cliente\n");
            salida.writeObject(cliente);
        } catch (IOException e) {
            System.out.println("Error durante la serialización: " + e.getMessage());
        }

        // Deserialización
        try (ObjectInputStream entrada = new ObjectInputStream(new FileInputStream(FILE_NAME))) {
            String mensaje = (String) entrada.readObject();
            Cliente clienteLeido = (Cliente) entrada.readObject();
            System.out.println("-----");
            System.out.println(mensaje + " " + clienteLeido);
            System.out.println("-----");
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Error durante la deserialización: " + e.getMessage());
        }
    }
}

```

## 9.4 La herencia en objetos serializables.

Para serializar objetos en una jerarquía de clases, solo la clase base (padre) debe implementar la interfaz `Serializable`. Las clases derivadas (hijas) heredan el `serialVersionUID` de la clase base. Si se desea personalizar el `serialVersionUID` en una clase derivada, debe declararse explícitamente en esa clase.

Durante la deserialización, primero se llama al constructor de la clase base y luego a los constructores de las clases derivadas. Por lo tanto, hay que asegurarse de que las clases derivadas tengan constructores adecuados para inicializar sus atributos.

Se puede personalizar la lógica de serialización y deserialización en las clases derivadas usando los métodos `writeObject` y `readObject`.

```

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Usuario implements Serializable {
    private static final long serialVersionUID = 123456789L;
    private String nombre;
    private transient String contrasena; // Atributo transitorio

    public Usuario (String nombre, String contrasena) {
        this.nombre = nombre;
        this.contrasena = contrasena;
    }

    // Getters y setters

    @Override
    public String toString() {

```



```

        return "Usuario {" + "nombre='" + nombre + '\'' + ", contrasena='" + contrasena + '\'' + '}';
    }
    // Personalización de la lógica de serialización
    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject(); // Llama al método predeterminado
        // Agregar lógica adicional si es necesario
        System.out.println("Objeto Usuario serializado.");
    }
    // Personalización de la lógica de deserialización
    private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
        in.defaultReadObject(); // Llama al método predeterminado
        // Agregar lógica adicional si es necesario
        System.out.println("Objeto Usuario deserializado.");
    }
}

```

**Ejemplo:** herencia de la clase `Figura` que implementa la interfaz `Serializable`. De esta clase heredan la clase `Circulo` y `Rectangulo`, ninguna de ellas implementa la interfaz `Serializable`.

➤ `Figura.java`

```

import java.io.Serializable;

public abstract class Figura implements Serializable {
    protected int x;
    protected int y;
    public Figura(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public abstract double area();
}

```

➤ `Circulo.java`

```

public class Circulo extends Figura {
    private static final double PI = 3.1416;
    protected double radio;
    public Circulo(int x, int y, double radio) {
        super(x, y);
        this.radio = radio;
    }
    @Override
    public double area() {
        return PI * radio * radio;
    }
}

```

➤ `Rectangulo.java`

```

public class Rectangulo extends Figura {
    protected double ancho, alto;
    public Rectangulo(int x, int y, double ancho, double alto) {
        super(x, y);
        this.ancho = ancho;
        this.alto = alto;
    }
    @Override
    public double area() {
        return ancho * alto;
    }
}

```

➤ `HerenciaSerializable.java`

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class HerenciaSerializable {
    public static void main(String[] args) {
        Figura fig1 = new Rectangulo(10, 15, 30, 60);
        Figura fig2 = new Circulo(12, 19, 60);
    }
}

```

La **interfaz `Serializable`** es una interfaz maker, es decir, **no tiene miembros ni métodos asociados**. En el ejemplo se puede observar que **declarar una clase como serializable no implica añadir ninguna funcionalidad adicional**.

**Excepciones más comunes** que se pueden dar al serializar/deserializar objetos en *Java*:

- **`IOException`**: problema general de entrada/salida durante la serialización/deserialización.
- **`ClassNotFoundException`**: no se encuentra la clase al deserializar un objeto.
- **`InvalidClassException`**: incompatibilidad entre la versión de la clase serializada y deserializada.
- **`NotSerializableException`**: la clase o algún campo no implementa `Serializable`.
- **`OptionalDataException`**: datos primitivos inesperados en un flujo de objetos.

```

try (ObjectOutputStream salida = new ObjectOutputStream(new FileOutputStream("figura.obj"));
    ObjectInputStream entrada = new ObjectInputStream(new FileInputStream("figura.obj"))) {
    salida.writeObject("Guardar un objeto de una clase hija\n");
    salida.writeObject(fig1);
    salida.writeObject(fig2);

    // Leer objetos del archivo
    String str = (String) entrada.readObject();
    Figura obj1 = (Figura) entrada.readObject();
    Figura obj2 = (Figura) entrada.readObject();

    System.out.println("-----");
    System.out.print(str);
    System.out.println(obj1.getClass().getSimpleName() + " origen ("
        + obj1.x + "," + obj1.y + ") " + " - área = " + obj1.area());
    System.out.println(obj2.getClass().getSimpleName() + " origen ("
        + obj2.x + "," + obj2.y + ") " + " - área = " + obj2.area());
    System.out.println("-----");
} catch (IOException | ClassNotFoundException e) {
    System.out.println("Error: " + e.getMessage());
}
}
}

```

**Ejemplo:** programa completo que implementa un CRUD (crear, leer, actualizar y eliminar) de empleados utilizando ObjectOutputStream y ObjectInputStream para serializar/deserializar objetos en un archivo.

- GestionEmpleados2.java → Realizar una copia de GestionEmpleados.java y modificar los siguientes métodos de la clase EmpleadosDAO. Eliminar el archivo de datos que se tenga creado con la aplicación anterior.

```

private void cargarEmpleadosDesdeArchivo() {
    try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(archivo))) {
        empleados = (List<Empleado>) ois.readObject();
    } catch (IOException | ClassNotFoundException e) {
        System.out.println(Utiles.ROJO + "Error: " + e.getMessage() + Utiles.RESET);
    }
}

private void guardarEmpleadosEnArchivo() {
    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(archivo))) {
        oos.writeObject(empleados);
    } catch (IOException e) {
        System.out.println(Utiles.ROJO + "Error: " + e.getMessage() + Utiles.RESET);
    }
}

```



## TAREA

15. (Opcional) Se desea guardar en un archivo binario las 10 notas finales de las asignaturas cursadas por un estudiante. Para ello, se debe crear un programa que solicite por teclado el nombre de las asignaturas y sus respectivas notas. Tanto el nombre de las asignaturas como las notas deben ser guardadas en el archivo. Finalmente, el programa debe leer el archivo, calcular el promedio de las notas del estudiante y mostrar el resultado en pantalla.
16. Una inmobiliaria tiene un catálogo de inmuebles disponibles para vender y alquilar. Cada inmueble tiene los siguientes atributos: identificador, dirección, ciudad, tipo (de alquiler o venta) y precio (para alquiler el precio es mensual; para venta, el precio es total). Se solicita crear un programa que permita realizar las siguientes operaciones: agregar, editar, eliminar, listar y buscar inmuebles. La opción de listar debe mostrar la información en pantalla y también guardarla en un archivo de texto con el mismo formato que se muestra en pantalla.
17. Crear una aplicación para gestionar el registro de usuarios. La aplicación debe ofrecer las siguientes funcionalidades mínimas:
  - Registro de usuarios: los usuarios pueden registrarse proporcionando un nombre de usuario y una contraseña. Los datos de los usuarios deben ser almacenados en un archivo.
  - Iniciar sesión: los usuarios pueden iniciar sesión ingresando su nombre de usuario y contraseña. Si las credenciales son correctas, se les permite acceder a la aplicación.
  - Cambio de contraseña: una vez iniciada sesión, los usuarios deben tener la opción de cambiar su contraseña.

## 10 Ficheros XML.

### 10.1 ¿Qué es XML?

**XML** (**eXtensible Markup Language** – **Lenguaje de Etiquetado Extensible**) es un lenguaje de marcado utilizado para estructurar, almacenar y transportar datos de una manera legible tanto para humanos como para máquinas. Define un conjunto de reglas para codificar documentos de texto en un formato estructurado. **Aspectos clave** de XML:

- ✓ **Lenguaje de marcado:** utiliza etiquetas para definir elementos de datos y sus relaciones. Por ejemplo, `<persona>` y `</persona>` marcan el inicio y fin de un elemento llamado "persona".
- ✓ **Anidamiento:** los elementos pueden anidarse dentro de otros, creando estructuras jerárquicas. Ejemplo:
 

```
<libro>
  <titulo>El Gran Gatsby</titulo>
  <autor>F. Scott Fitzgerald</autor>
</libro>
```
- ✓ **Atributos:** los elementos pueden tener atributos que brindan información adicional. Ejemplo:
 

```
<persona nombre="Juan" edad="30" />
```
- ✓ **Legibilidad:** diseñado para ser leído y editado con un editor de texto, útil para configuración, intercambio de datos y representación de documentos.
- ✓ **Extensible:** permite definir etiquetas y estructuras propias según las necesidades, lo que lo hace versátil y ampliamente utilizado.
- ✓ **Independiente del lenguaje:** no está ligado a ningún lenguaje de programación o plataforma específica.
- ✓ **Uso común:** se usa en diversas aplicaciones, desde configuración de software hasta intercambio de datos y representación de documentos.
- ✓ **Validación:** puede validarse con *DTD* (*Document Type Definition*) o *XSD* (*XML Schema Definition*) para asegurar que siga una estructura específica.

```
<?xml version="1.0" encoding="UTF-8"?>
<biblioteca>
  <libro>
    <titulo>La vida está en otra parte</titulo>
    <autor>Milan Kundera</autor>
    <fechaPublicacion año="1973"/>
  </libro>
  <libro>
    <titulo>Pantaleón y las visitadoras</titulo>
    <autor fechaNacimiento="28/03/1936">Mario Vargas Llosa</autor>
    <fechaPublicacion año="1973"/>
  </libro>
  <libro>
    <titulo>Conversación en la catedral</titulo>
    <autor fechaNacimiento="28/03/1936">Mario Vargas Llosa</autor>
    <fechaPublicacion año="1969"/>
  </libro>
</biblioteca>
```

#### 10.1.1 Usos de XML.

XML se utiliza en muchas aplicaciones por su capacidad para representar datos de forma estructurada. Algunos **usos comunes** son:

- ✓ **Intercambio de datos:** facilita el intercambio de datos entre aplicaciones y sistemas, independientemente del lenguaje de programación o plataforma.
- ✓ **Configuración:** almacena configuraciones de aplicaciones y sistemas, permitiendo ajustes sin modificar el código fuente.
- ✓ **Representación de documentos:** utilizado en *HTML* para páginas web y en documentos de procesamiento de texto para estructurar el contenido.
- ✓ **Servicios Web:** facilita la comunicación entre aplicaciones distribuidas mediante protocolos como SOAP, utilizando XML para enviar y recibir datos.
- ✓ **Bases de Datos:** algunas *BD* permiten almacenar datos en formato XML, facilitando su manipulación y consulta.
- ✓ **Lenguajes de marcado específicos:** crea lenguajes personalizados para dominios específicos, como *MusicXML* para la música o *AeroML* para la industria aeroespacial.
- ✓ **Publicación de contenido:** usado en *CMS* y la industria editorial para almacenar y publicar contenido.
- ✓ **Intercambio de datos en la nube:** facilita la integración de sistemas en entornos de nube.
- ✓ **Automatización de procesos empresariales:** define flujos de trabajo y reglas comerciales para automatizar tareas y procesos.

#### 10.1.2 Ficheros XML.

Los ficheros XML permiten el intercambio de información entre aplicaciones utilizando un archivo de texto

plano con etiquetas para definir los valores almacenados. Ejemplo de un fichero XML (*productos.xml*) con información de productos:

```
<?xml version="1.0" encoding="UTF-8">
<productos>
  <producto>
    <nombre>Cereales</nombre>
    <precio>3.45</precio>
  </producto>
  <producto>
    <nombre>Cola cao</nombre>
    <precio>1.45</precio>
  </producto>
  <producto>
    <nombre>Agua mineral</nombre>
    <precio>1.00</precio>
  </producto>
</productos>
```

Otro ejemplo

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE Mensaje SYSTEM "Mensaje.dtd">
<Mensaje>
  <Remitente>
    <Nombre>Anuska Esteban</Nombre>
    <Mail> anuska.informatica@gmail.com </Mail>
  </Remitente>
  <Destinatario>
    <Nombre>Pepito de los Palotes</Nombre>
    <Mail>pepito.palotes@correo.es</Mail>
  </Destinatario>
  <Texto>
    <Asunto>
      Este es el asunto del mensaje.
    </Asunto>
    <Parrafo>
      Este es mi documento con una estructura muy sencilla
      no contiene atributos ni entidades...
    </Parrafo>
  </Texto>
</Mensaje>
```

Para **procesar y manipular datos XML en Java**, se pueden usar varias librerías:

- ✓ **SAX** (*Simple API for XML*): es un analizador de eventos XML. **Lee el documento XML secuencialmente sin almacenarlo en memoria y dispara eventos** para elementos, atributos, texto, etc.
  - Ventajas: eficiente en memoria y CPU, ya que no almacena el documento completo en memoria. Ideal para archivos grandes.
  - Desventajas: no permite navegación bidireccional, solo lectura, requiere manejo de eventos y errores en tiempo real.
  - Uso recomendado: procesar grandes archivos XML eficientemente.
- ✓ **DOM** (*Document Object Model*): **crea un árbol en memoria del documento XML, permitiendo acceder y modificar cualquier parte**.
  - Ventajas: navegación bidireccional, lectura y escritura, fácil de usar para documentos pequeños.
  - Desventajas: requiere almacenar el documento completo en memoria. No es eficiente para grandes volúmenes de datos.
  - Uso recomendado: documentos XML pequeños o cuando se necesita modificar el contenido.
- ✓ **XStream**: librería que **convierte objetos Java a XML y viceversa**.
  - Ventajas: fácil de usar, soporta anotaciones para personalizar la serialización.
  - Desventajas: menos eficiente para grandes volúmenes de datos.
  - Uso recomendado: convertir objetos Java en XML y viceversa sin preocuparse por detalles de bajo nivel.

## 10.2 Acceso a ficheros XML con SAX.

SAX (*Simple API for XML*) es una **interfaz estándar en Java para procesar documentos XML**. A diferencia de DOM, que carga el documento completo en memoria, **SAX procesa el XML de manera secuencial, nodo por nodo, lo que es ideal para archivos grandes**. Sin embargo, **SAX no permite navegar hacia atrás ni modificar el documento XML**, solo ofrece una vista secuencial (impide tener una visión global del documento que se va a analizar).

SAX **trabaja mediante un enfoque basado en eventos**. El parser SAX emite eventos cuando encuentra elementos, atributos y contenido de texto en el documento. Estos **eventos son manejados por un controlador que ejecuta métodos definidos por el programador**. El parser SAX viene con el **JDK**, por lo que **no es necesario descargar librerías**.

**Eventos y métodos del controlador SAX:**

Evento	Método del manejador de eventos	Descripción
Inicio del documento	<code>startDocument()</code>	Llamado al comenzar el procesamiento del XML.
Fin del documento	<code>endDocument()</code>	Llamado al finalizar el procesamiento del XML.
Inicio de un elemento	<code>startElement(String uri, String localName, String qName, Attributes attributes)</code>	Llamado al encontrar el inicio de un elemento XML.
Fin de un elemento	<code>endElement(String uri, String localName, String qName)</code>	Llamado al encontrar el final de un elemento XML.

Contenido de texto	<code>characters(char[] ch, int start, int length)</code>	Llamado al encontrar texto dentro de un elemento.
Comentario XML	<code>comment(char[] ch, int start, int length)</code>	Llamado al encontrar un comentario XML.
Error SAX	<code>error(SAXParseException e)</code>	Llamado al producirse un error SAX.
Advertencia SAX	<code>warning(SAXParseException e)</code>	Llamado al producirse una advertencia SAX.
Error fatal SAX	<code>fatalError(SAXParseException e)</code>	Llamado al producirse un error fatal SAX.

### Pasos para crear un analizador SAX:

1. **Crear un objeto SAXParserFactory.**
2. **Crear un SAXParser** a partir del objeto anterior.
3. **Crear un controlador de eventos extendiendo la clase DefaultHandler.** Aquí se definen las acciones a realizar cuando se encuentra una etiqueta específica. Los métodos básicos a reescribir son:
  - `startElement`: apertura de etiqueta.
  - `characters`: contenido de una etiqueta.
  - `endElement`: cierre de etiqueta.
4. **Llamar al método parse desde SAXParser**, pasando el nombre del archivo XML y el controlador de eventos.

Ejemplo: documento XML y eventos SAX que se producirían durante su procesamiento.

```
<?xml version="1.0" encoding="UTF-8"?>
<libros>
  <libro>
    <titulo>El Gran Gatsby</titulo>
    <autor>F. Scott Fitzgerald</autor>
  </libro>
  <libro>
    <titulo>Cien años de soledad</titulo>
    <autor>Gabriel García Márquez</autor>
  </libro>
</libros>
```

Tabla con los eventos SAX que ocurrirían durante el procesamiento de este documento XML:

Evento	Método del Manejador de Eventos
Inicio del Documento	<code>startDocument()</code>
Inicio de un Elemento libros	<code>startElement("libros")</code>
Inicio de un Elemento libro	<code>startElement("libro")</code>
Inicio de un Elemento titulo	<code>startElement("titulo")</code>
Contenido de Texto para titulo	<code>characters("El Gran Gatsby".toCharArray(), 0, 14)</code>
Fin de un Elemento titulo	<code>endElement("titulo")</code>
Inicio de un Elemento autor	<code>startElement("autor")</code>
Contenido de Texto para autor	<code>characters("F. Scott Fitzgerald".toCharArray(), 0, 19)</code>
Fin de un Elemento autor	<code>endElement("autor")</code>
Fin de un Elemento libro	<code>endElement("libro")</code>
Inicio de un Elemento libro	<code>startElement("libro")</code>
Inicio de un Elemento titulo	<code>startElement("titulo")</code>
Contenido de Texto para titulo	<code>characters("Cien años de soledad".toCharArray(), 0, 20)</code>
Fin de un Elemento titulo	<code>endElement("titulo")</code>
Inicio de un Elemento autor	<code>startElement("autor")</code>
Contenido de Texto para autor	<code>characters("Gabriel García Márquez".toCharArray(), 0, 22)</code>
Fin de un Elemento autor	<code>endElement("autor")</code>
Fin de un Elemento libro	<code>endElement("libro")</code>
Fin de un Elemento libros	<code>endElement("libros")</code>
Fin del Documento	<code>endDocument()</code>

Ejemplo: uso de SAX básico para acceder y procesar un fichero XML.

Suponer que se tiene el siguiente archivo XML llamado "empleados.xml"

```
<?xml version="1.0" encoding="UTF-8"?>
<empleados>
  <empleado>
    <id>1</id>
    <nombre>Juan</nombre>
    <apellidos>Pérez</apellidos>
    <departamento>Ventas</departamento>
    <suelo>3000</suelo>
  </empleado>
  <empleado>
    <id>2</id>
```

```

        <nombre>María</nombre>
        <apellidos>Gómez</apellidos>
        <departamento>Recursos Humanos</departamento>
        <sueldo>3500</sueldo>
    </empleado>
</empleados>

```

Código de ejemplo de cómo utilizar SAX para acceder y procesar este archivo XML:

```

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.io.File;
import java.io.IOException;

class GestionContenido extends DefaultHandler {
    private boolean bId = false;
    private boolean bNombre = false;
    private boolean bApellidos = false;
    private boolean bDepartamento = false;
    private boolean bSueldo = false;

    @Override
    public void startDocument() {
        System.out.println("***** Comienzo del documento XML *****");
    }

    @Override
    public void endDocument() {
        System.out.println("***** Final del documento XML *****");
    }

    @Override
    public void startElement(String uri, String localName, String qName, Attributes attributes) {
        switch (qName.toLowerCase()) {
            case "id":
                bId = true;
                break;
            case "nombre":
                bNombre = true;
                break;
            case "apellidos":
                bApellidos = true;
                break;
            case "departamento":
                bDepartamento = true;
                break;
            case "sueldo":
                bSueldo = true;
                break;
        }
    }

    @Override
    public void characters(char[] ch, int start, int length) {
        String content = new String(ch, start, length).trim();

        if (bId) {
            System.out.println("ID: " + content);
            bId = false;
        } else if (bNombre) {
            System.out.println(" - Nombre: " + content);
            bNombre = false;
        } else if (bApellidos) {
            System.out.println(" - Apellidos: " + content);
            bApellidos = false;
        } else if (bDepartamento) {
            System.out.println(" - Departamento: " + content);
            bDepartamento = false;
        } else if (bSueldo) {
            System.out.println(" - Sueldo: " + content);
            bSueldo = false;
        }
    }
}

```

En SAX, DefaultHandler es una clase proporcionada por la biblioteca SAX que implementa la interfaz ContentHandler, ErrorHandler y otras interfaces relacionadas con el manejo de eventos XML. **Al extender DefaultHandler, se pueden sobrescribir los métodos que manejan los diferentes eventos XML según las necesidades.**

DefaultHandler **proporciona implementaciones por defecto de todos los métodos.** Esto significa que **se puede elegir implementar solo los métodos que son relevantes para la aplicación**, en lugar de tener que implementar todos los métodos de las interfaces, incluso si no se necesitan.



```

    }
}

@Override
public void endElement(String uri, String localName, String qName) {
    // Manejo del final de elementos si es necesario
}

@Override
public void error(SAXParseException e) throws SAXException {
    System.out.println("Error: " + e.getMessage());
}

@Override
public void warning(SAXParseException e) throws SAXException {
    System.out.println("Advertencia: " + e.getMessage());
}

}

public class EjemploSAX {
    public static void main(String[] args) {
        try {
            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser saxParser = factory.newSAXParser();
            GestionContenido handler = new GestionContenido();
            saxParser.parse(new File("empleados.xml"), handler);
        } catch (SAXException e) {
            System.out.println("Error SAX: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("Error IO: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

```

**SAXParser no implementa AutoCloseable**, lo que implica que **no puede ser utilizado directamente en un bloque try-with-resources**. SaxParser no maneja flujos directamente, por lo que **no necesita ser cerrado explícitamente**.



## TAREA

El siguiente archivo *XML* contiene una lista de libros, donde cada libro está descrito con información sobre su título, autor, *ISBN*, categoría y precio:

```

<?xml version="1.0" encoding="UTF-8"?>
<libros>
  <libro>
    <titulo>El Gran Gatsby</titulo>
    <autor>F. Scott Fitzgerald</autor>
    <isbn>978-0743273565</isbn>
    <categoria>Ficción</categoria>
    <precio>12.99</precio>
  </libro>
  <libro>
    <titulo>Cien Años de Soledad</titulo>
    <autor>Gabriel García Márquez</autor>
    <isbn>978-0307474728</isbn>
    <categoria>Realismo Mágico</categoria>
    <precio>15.50</precio>
  </libro>
  <libro>
    <titulo>1984</titulo>
    <autor>George Orwell</autor>
    <isbn>978-0451524935</isbn>
    <categoria>Distopía</categoria>
    <precio>9.99</precio>
  </libro>
</libros>

```

18. Crear un programa llamado “Contador de Etiquetas” que utilice la *API SAX* para analizar el archivo *XML* proporcionado. El objetivo del programa es contar cuántas veces aparece cada etiqueta en el archivo *XML* y mostrar el resultado en pantalla. Cada línea debe mostrar el nombre de la etiqueta y la cantidad de veces que aparece en el archivo. **Nota:** utilizar un `HashMap<String, Integer>` para almacenar el conteo de cada etiqueta. La clave será el nombre de la etiqueta y el valor será el número de veces que aparece en el *XML*.

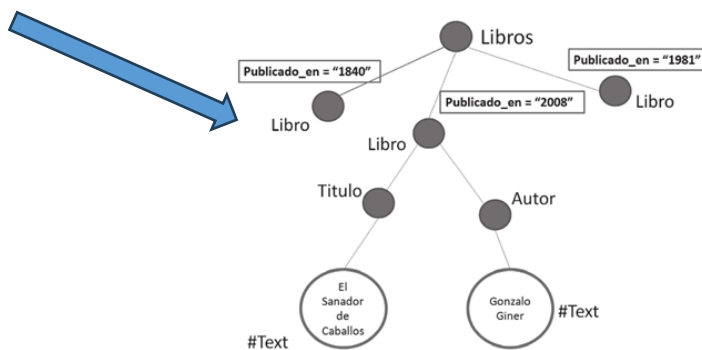
## 10.3 Acceso a ficheros XML con DOM.

**DOM (Document Object Model)** es una interfaz que permite manipular documentos XML de manera dinámica. Representa un documento XML como un árbol en memoria, donde cada elemento, atributo y texto se convierte en un nodo del árbol. Esto permite acceder y modificar el contenido del documento fácilmente utilizando la API del lenguaje de programación.

En DOM, los nodos más comunes son los de elementos y los de texto. Con DOM, se puede leer, escribir, crear, eliminar y recorrer nodos en el documento XML. Una vez que el documento está cargado en memoria como un árbol, se puede recorrer y analizar los nodos utilizando la API proporcionada.

Por ejemplo, el siguiente documento XML muestra una lista de libros, cada uno con un atributo Publicado\_en, un título y un autor:

```
<?xml version="1.0" encoding="UTF-8"?>
<Libros xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance' xsi:noNamespaceSchemaLocation='LibrosEsquema.xsd'>
  <Libro Publicado_en="1840">
    <Titulo>El Capote</Titulo>
    <Autor>Nikolai Gogol</Autor>
  </Libro>
  <Libro Publicado_en="2008">
    <Titulo>El Sanador de Caballos</Titulo>
    <Autor>Gonzalo Giner</Autor>
  </Libro>
  <Libro Publicado_en="1981">
    <Titulo>El Nombre de la Rosa</Titulo>
    <Autor>Umberto Eco</Autor>
  </Libro>
</Libros>
```



Un esquema del árbol DOM que representaría internamente el documento se muestra en la imagen y para no complicar el gráfico solo se ha desarrollado hasta el final uno de los elementos *Libro*, dejando los otros dos sin detallar.

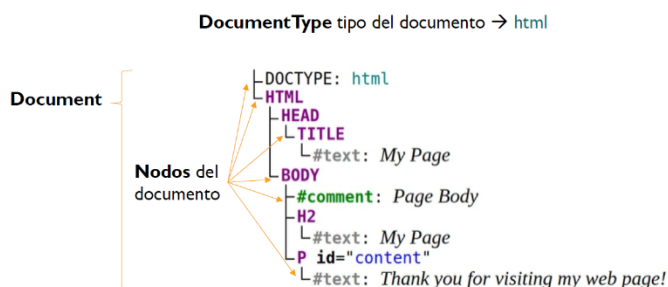
DOM permite tanto la lectura como la modificación de documentos XML. Sin embargo, puede ser ineficiente con documentos XML muy grandes debido a que el árbol se carga completamente en memoria.

En Java, la principal API para trabajar con DOM es JAXP (Java API for XML Processing). Las **clases y paquetes principales** que se utilizan son:

- ✓ `javax.xml.parsers.*`: para crear instancias de `DocumentBuilder`.
- ✓ `org.w3c.dom.*`: representa el modelo DOM según la W3C.
- ✓ `javax.xml.transform.*`: para generar archivos XML a partir del árbol DOM.

Al trabajar con DOM en Java, es útil conocer estas **interfaces principales**:

- ✓ **Document** (`org.w3c.dom.Document`): representa todo el documento XML como un árbol en memoria. Proporciona métodos para acceder a los elementos raíz y para crear nuevos elementos y nodos.
- ✓ **Element** (`org.w3c.dom.Element`): representa un elemento XML en el documento. Permite acceder a los atributos y contenidos de un elemento.
- ✓ **Node** (`org.w3c.dom.Node`): es la interfaz base para todos los nodos en el árbol DOM. Proporciona métodos para obtener información sobre el tipo de nodo y su contenido.
- ✓ **NodeList** (`org.w3c.dom.NodeList`): representa una lista de nodos. Se utiliza para acceder a los hijos de un nodo o a los nodos que coinciden con ciertos criterios.
- ✓ **Attr** (`org.w3c.dom.Attr`): representa un atributo de un elemento XML. Puede utilizarse para acceder al nombre y al valor del atributo.
- ✓ **Text** (`org.w3c.dom.Text`): representa el contenido de texto dentro de un elemento XML. Permite acceder al texto contenido en un nodo de texto.
- ✓ **Comment** (`org.w3c.dom.Comment`): representa comentarios dentro del documento XML. Permite acceder al contenido de los comentarios.



Los **Element** son los escritos en color **púrpura** (etiquetas)

**NodeList** lista de hijos de un nodo. Ej. de HTML: head y body

**Attr** atributos de las etiquetas Ej. de P es id

**CharacterData** todo el texto que no es una etiqueta. Comprenden los **comment**, los **text** y los **attr**

**Text** `CharacterData` + `Element` entremezclados

- ✓ **NamedNodeMap** (org.w3c.dom.NamedNodeMap): representa una **colección de nodos indexados por nombre**. Se utiliza para acceder a los atributos de un elemento.
- ✓ **CharacterData** (org.w3c.dom.CharacterData): se utiliza para representar **nodos de texto**, *cdata* y comentarios.
- ✓ **DocumentType** (org.w3c.dom.DocumentType): se utiliza para representar el **tipo de documento** (DOCTYPE) de un documento XML.
- ✓ **DocumentBuilderFactory** (javax.xml.parsers.DocumentBuilderFactory): no es una interfaz *DOM*, pero es importante **para crear instancias de DocumentBuilder**, que se utiliza para cargar documentos XML y crear instancias de Document.
- ✓ **DocumentBuilder** (javax.xml.parsers.DocumentBuilder): se utiliza **para crear instancias de Document a partir de documentos XML**. Proporciona métodos para analizar y cargar documentos XML en un objeto *DOM*.

Ejemplo: creación de un fichero XML a partir del fichero de empleados generado en el ejemplo del punto 8 ("empleados.dat").

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

public class CrearEmpleadosXml {
    public static void main(String[] args) {
        File inputFile = new File("empleados.dat");
        File outputFile = new File("empleados.xml");

        try (RandomAccessFile raf = new RandomAccessFile(inputFile, "r")) {
            // Crear DocumentBuilderFactory y DocumentBuilder
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document document = builder.newDocument(); // Crear un nuevo documento XML

            // Crear el nodo raíz "empleados"
            Element rootElement = document.createElement("empleados");
            document.appendChild(rootElement);

            // Leer número de empleados
            int numEmpleados = raf.readInt();
            for (int i = 0; i < numEmpleados; i++) {
                int id = raf.readInt();
                String nombre = raf.readUTF();
                String apellidos = raf.readUTF();
                String departamento = raf.readUTF();
                double sueldo = raf.readDouble();

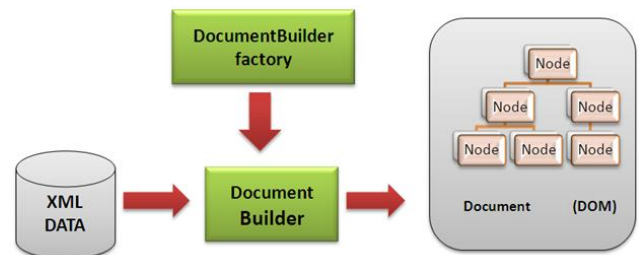
                if (id > 0) {
                    Element empleado = document.createElement("empleado");
                    rootElement.appendChild(empleado);

                    crearElemento("id", Integer.toString(id), empleado, document);
                    crearElemento("nombre", nombre.trim(), empleado, document);
                    crearElemento("apellidos", apellidos.trim(), empleado, document);
                    crearElemento("departamento", departamento.trim(), empleado, document);
                    crearElemento("sueldo", Double.toString(sueldo), empleado, document);
                }
            }

            // Convertir el documento a archivo XML
            Transformer transformer = TransformerFactory.newInstance().newTransformer();
            transformer.setOutputProperty(OutputKeys.INDENT, "yes");
            Source source = new DOMSource(document);
            Result result = new StreamResult(outputFile);
            transformer.transform(source, result);

            System.out.println("Archivo XML creado con éxito.");
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
    }

    private static void crearElemento(String nombre, String valor, Element raiz, Document document) {
        Element elemento = document.createElement(nombre);
        elemento.appendChild(document.createTextNode(valor));
        raiz.appendChild(elemento);
    }
}
```



Ejemplo: leer el documento XML "empleados.xml" y mostrar su contenido por pantalla.

```

import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.File;

public class LecturaEmpleadosXML {
    public static void main(String[] args) {
        File inputFile = new File("empleados.xml");

        try {
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document document = builder.parse(inputFile);
            document.getDocumentElement().normalize();

            System.out.printf("Elemento raíz: %s %n", document.getDocumentElement().getNodeName());

            NodeList empleados = document.getElementsByTagName("empleado");
            System.out.printf("Nodos empleado a recorrer: %d %n", empleados.getLength());

            for (int i = 0; i < empleados.getLength(); i++) {
                Node empleado = empleados.item(i);
                if (empleado.getNodeType() == Node.ELEMENT_NODE) {
                    Element elemento = (Element) empleado;
                    System.out.printf("ID = %s %n", gettextContent(elemento, "id"));
                    System.out.printf(" - Nombre = %s %n", gettextContent(elemento, "nombre"));
                    System.out.printf(" - Apellidos = %s %n", gettextContent(elemento, "apellidos"));
                    System.out.printf(" - Departamento = %s %n", gettextContent(elemento, "departamento"));
                    System.out.printf(" - Sueldo = %s %n", gettextContent(elemento, "sueldo"));
                }
            }
        } catch (Exception e) {
            System.err.println("Error: " + e.getMessage());
        }
    }

    private static String gettextContent(Element element, String tagName) {
        NodeList nodeList = element.getElementsByTagName(tagName);
        if (nodeList.getLength() > 0)
            return nodeList.item(0).getTextContent();

        return "N/A";
    }
}

```



## TAREA

19. Crear un programa que a partir del fichero de objetos Persona ("personas.obj"), que fue creado en uno de los ejemplos del punto 9, genere un documento XML usando DOM. Finalmente, el programa, debe leer el documento XML recién creado y mostrar su contenido en la pantalla.

## 10.4 Serialización de objetos a XML.

La **serialización de objetos a XML** consiste en convertir un objeto de un lenguaje de programación, como *Java*, en un formato XML para su almacenamiento o transmisión de manera estructurada y legible. Esto permite guardar objetos en archivos XML, enviar objetos a través de la red o intercambiar datos con otros sistemas que utilizan XML como formato estándar.

La **deserialización** es el proceso inverso: toma datos en formato XML y los convierte de nuevo en objetos del lenguaje de programación.

**XStream** es una biblioteca de código abierto para *Java* que facilita la serialización y deserialización de objetos a XML y viceversa. Es una opción popular por su simplicidad y flexibilidad.

Pasos para **serializar y deserializar con XStream**:

1. **Agregar la biblioteca XStream al proyecto.** Se puede descargar desde el sitio oficial de XStream <https://x-stream.github.io/download.html>. Descargar la **Binary distribution**, descomprimir e importar los archivos xstream-1.4.20.jar, xmlpull-1.1.3.1.jar y mxparser-1.2.2.jar. Alternativamente, agregar XStream como una dependencia de

```

Dependencies
> xstream-1.4.20.jar
> mxparser-1.2.2.jar
> xmlpull-1.1.3.1.jar

```

Maven o Gradle. Dependencia Maven:

```
<dependency>
  <groupId>com.thoughtworks.xstream</groupId>
  <artifactId>xstream</artifactId>
  <version>1.4.20</version>
</dependency>
```

2. **Definir la clase que se desea serializar.** No se requieren anotaciones especiales, lo que simplifica su uso. Por ejemplo, una clase Empleado puede definirse así:

```
import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.security.AnyTypePermission;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

// Clase Empleado
class Empleado {
    private int id;
    private String nombre;
    private String apellidos;
    private String departamento;
    private double sueldo;

    public Empleado(int id, String nombre, String apellidos, String departamento, double sueldo) {
        this.id = id;
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.departamento = departamento;
        this.sueldo = sueldo;
    }

    @Override
    public String toString() {
        return "Empleado{" + "id=" + id + ", nombre='" + nombre + '\'' + ", apellidos='" +
            apellidos + '\'' + ", departamento='" + departamento + '\'' + ", sueldo=" + sueldo + '}';
    }
}

// Clase para manejar la lista de empleados
class ListaEmpleados {
    private List<Empleado> empleados = new ArrayList<>();

    public void agregarEmpleado(Empleado empleado) {
        empleados.add(empleado);
    }

    public List<Empleado> obtenerEmpleados() {
        return empleados;
    }
}
```

#### PARA AMPLIAR...

**JAXB (Java Architecture for XML Binding)** es una API en Java que permite convertir objetos Java en documentos XML y viceversa (es decir, serialización y deserialización de datos entre Java y XML). En esencia, JAXB facilita la vinculación de clases Java con estructuras XML, lo que permite que los desarrolladores trabajen con datos XML utilizando objetos Java directamente, sin tener que escribir mucho código de manipulación de XML.

3. **Crear un objeto XStream y utilizar su método toXML para serializar un objeto en XML o fromXML para deserializar:**

```
// Clase principal para serializar y deserializar
public class SerializacionListaEmpleados {
    private static final String ARCHIVO_XML = "empleados.xml";

    public static void main(String[] args) {
        ListaEmpleados listaEmpleados = new ListaEmpleados();

        listaEmpleados.agregarEmpleado(new Empleado(1, "Juan", "Pérez", "Ventas", 50000.0));
        listaEmpleados.agregarEmpleado(new Empleado(2, "María", "López", "Recursos Humanos", 55000.0));
        listaEmpleados.agregarEmpleado(new Empleado(3, "Pedro", "González", "TI", 60000.0));

        serializarEmpleados(listaEmpleados);
        ListaEmpleados empleadosDeserializados = deserializarEmpleados();

        if (empleadosDeserializados != null)
            empleadosDeserializados.obtenerEmpleados().forEach(System.out::println);
    }

    // Método para serializar una lista de empleados a XML
    private static void serializarEmpleados(ListaEmpleados listaEmpleados) {
        XStream xstream = new XStream();
    }
```



```

xstream.alias("empleados", ListaEmpleados.class);
xstream.alias("empleado", Empleado.class);
xstream.addImplicitCollection(ListaEmpleados.class, "empleados");
try (FileOutputStream fos = new FileOutputStream(ARCHIVO_XML)) {
    xstream.toXML(listaEmpleados, fos);
    System.out.println("Serialización completada.");
} catch (IOException e) {
    System.out.println("Error al serializar la lista de empleados: " + e.getMessage());
}
}

// Método para deserializar una lista de empleados desde un archivo XML
private static ListaEmpleados deserializarEmpleados() {
    XStream xstream = new XStream();

    xstream.addPermission(AnyTypePermission.ANY);
    xstream.alias("empleados", ListaEmpleados.class);
    xstream.alias("empleado", Empleado.class);
    xstream.addImplicitCollection(ListaEmpleados.class, "empleados");

    try (FileInputStream fis = new FileInputStream(ARCHIVO_XML)) {
        System.out.println("Deserialización completada.");
        return (ListaEmpleados) xstream.fromXML(fis);
    } catch (IOException e) {
        System.out.println("Error al deserializar la lista de empleados: " + e.getMessage());
        return null;
    }
}
}

```

**Ejemplo:** programa completo que implementa un CRUD (crear, leer, actualizar y eliminar) de empleados utilizando la librería *XStream* para serializar/deserializar objetos en un archivo *XML*.

- Agregar la dependencia de *XStream* al proyecto.
- GestionEmpleados3.java → Realizar una copia de GestionEmpleados.java y modificar los siguientes métodos de la clase EmpleadosDAO.

```

private void cargarEmpleadosDesdeArchivo() {
    XStream xstream = new XStream();

    xstream.addPermission(AnyTypePermission.ANY); // Habilitar la deserialización
    xstream.alias("empleado", Empleado.class); // Alias para los elementos de la lista (Empleado)

    try (FileInputStream fis = new FileInputStream(archivo)) {
        empleados = (List<Empleado>) xstream.fromXML(fis);
    } catch (IOException e) {
        System.out.println(Utiles.ROJO + "Error: " + e.getMessage() + Utiles.RESET);
    }
}

private void guardarEmpleadosEnArchivo() {
    XStream xstream = new XStream();

    xstream.alias("empleado", Empleado.class); // Alias para los elementos de la lista (Empleado)
    // Serializar la lista de empleados a XML y guardarla en el archivo
    try (FileOutputStream fos = new FileOutputStream(archivo)) {
        xstream.toXML(empleados, fos);
    } catch (IOException e) {
        System.out.println(Utiles.ROJO + "Error: " + e.getMessage() + Utiles.RESET);
    }
}
}

```

- Añadir las siguientes importaciones.

```

import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.security.AnyTypePermission;

```

- Cambiar en la clase EmpleadosDAO “empleados.dat” por “empleados.xml”.

## TAREA

- 20. Definir una clase Libreria que gestione una lista de libros. Cada libro debe tener las propiedades título, autor y año de publicación. Crear un programa que permita crear una lista de libros, serializarla a un archivo *XML* y luego deserializarla para recuperar los datos.**



21. Extender el ejercicio anterior para crear un programa de gestión de una biblioteca que permita a los usuarios realizar las siguientes acciones: consultar libros, añadir nuevos libros, eliminar libros existentes y modificar la información de los libros. Además, el programa debe guardar la lista de libros en un archivo XML al cerrarse y cargar los datos guardados al iniciarse nuevamente. Esto permitirá a los usuarios mantener un registro persistente de su biblioteca y continuar gestionándola en futuras sesiones.

## 11 Tratamiento de documentos JSON.

Para **trabajar con JSON en Java**, se pueden utilizar varias **bibliotecas**. Las **más populares** son:

- ✓ **Jackson**: muy utilizada para convertir entre objetos Java y JSON. Es **rápida y eficiente**.
- ✓ **Gson**: desarrollada por Google, facilita la conversión entre objetos Java y JSON. Ideal **para casos simples**.
- ✓ **JSON.simple**: una **opción ligera y liviana** para trabajar con JSON, adecuada **para casos básicos**.
- ✓ **JSON-lib**: permite convertir entre JSON, XML, Strings, Mapas, etc., pero **no está tan actualizada**.
- ✓ **org.json**: **biblioteca simple**, incluida en Java EE, para manejar JSON.

Entre **las dos bibliotecas principales**, Jackson y Gson, la elección depende de las necesidades específicas. **Jackson es más rápido y versátil**, adecuado para aplicaciones donde el rendimiento es esencial. **Gson es más fácil de usar** y es perfecto para casos donde la simplicidad es clave.

**Pasos para trabajar con JSON en Java** utilizando la **biblioteca Jackson**:

1. **Agregar la dependencia de Jackson**. Si se usa Maven, añadir las siguientes líneas en el archivo pom.xml:

```
<dependencies>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.17.2</version> <!-- Utilizar la versión más reciente disponible -->
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.17.2</version> <!-- Utilizar la versión más reciente disponible -->
  </dependency>
</dependencies>
```

2. **Convertir JSON a un objeto Java** utilizando la clase **ObjectMapper**:

```
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JSONEjemplo1 {
    public static void main(String[] args) {
        ObjectMapper objectMapper = new ObjectMapper();
        String jsonString = "{\"nombre\": \"Juan\", \"edad\": 25}";
        try {
            Persona persona = objectMapper.readValue(jsonString, Persona.class);
            System.out.println("Nombre: " + persona.getNombre());
            System.out.println("Edad: " + persona.getEdad());
        } catch (JsonProcessingException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

class Persona {
    @JsonProperty("nombre")
    private String nombre;
    @JsonProperty("edad")
    private int edad;

    public String getNombre() { return nombre; }
    public int getEdad() { return edad; }
}
```

3. **Convertir un objeto Java a JSON**:

```
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.core.JsonProcessingException;
```

```
import com.fasterxml.jackson.databind.ObjectMapper;

public class JSONEjemplo2 {
    public static void main(String[] args) {
        ObjectMapper objectMapper = new ObjectMapper();
        Persona persona = new Persona("María", 30);

        try {
            String jsonString = objectMapper.writeValueAsString(persona);
            System.out.println(jsonString);
        } catch (JsonProcessingException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

class Persona {
    @JsonProperty("nombre")
    private String nombre;
    @JsonProperty("edad")
    private int edad;

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

#### Anotaciones útiles de Jackson:

- ✓ **@JsonProperty**: cambia el nombre de la propiedad en *JSON* sin cambiar el nombre del campo en la clase.
 

```
@JsonProperty("nombre")
private String name;
```
- ✓ **@JsonIgnore**: excluir campos de la serialización/deserialización.
 

```
@JsonIgnore
private int edad;
```
- ✓ **@JsonFormat**: define el formato para fechas.
 

```
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd HH:mm:ss")
private LocalDateTime fecha;
```
- ✓ **@JsonInclude**: controla la inclusión de propiedades nulas.
 

```
@JsonInclude(JsonInclude.Include.NON_NULL) // Opcionalmente, NON_EMPTY para propiedades vacías
public class Persona {
    private String nombre;
    private String direccion;
    // Getters y setters
}
```
- ✓ **@JsonCreator y @JsonProperty en el constructor**: personalizar la deserialización de objetos. **@JsonCreator** se coloca en el constructor que se utilizará para crear objetos a partir de *JSON* y **@JsonProperty** se utiliza para mapear los campos de *JSON* a los parámetros del constructor.
 

```
public class Persona {
    private String nombre;
    private int edad;

    @JsonCreator
    public Persona(@JsonProperty("nombre") String nombre, @JsonProperty("edad") int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Getters y setters
}
```

**Ejemplo:** anotaciones a la clase *Persona* para personalizar la serialización con *Jackson*.

```
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

class Persona {
```

```

private String nombre;
private int edad;

@JsonCreator
public Persona(@JsonProperty("nombre") String nombre, @JsonProperty("edad") int edad) {
    this.nombre = nombre;
    this.edad = edad;
}

@JsonProperty("nombre")
public String getNombre() { return nombre; }

@JsonProperty("edad")
public int getEdad() { return edad; }
}

public class JSONListaPersonas {
    private List<Persona> personas = new ArrayList<>();
    public List<Persona> getPersonas() { return personas; }
    public static void main(String[] args) {
        JSONListaPersonas listaPersonas = new JSONListaPersonas();
        listaPersonas.getPersonas().add(new Persona("Ana", 30));
        listaPersonas.getPersonas().add(new Persona("Carlos", 25));
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            objectMapper.writeValue(new File("personas.json"), listaPersonas);
            JSONListaPersonas containerLeido = objectMapper.readValue(new File("personas.json"),
                                                                    new TypeReference<>() {});
            containerLeido.getPersonas().forEach(p -> System.out.println(p.getNombre() +
                                                                    " - " + p.getEdad()));
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

```

**Ejemplo:** programa completo que implementa un CRUD (crear, leer, actualizar y eliminar) de empleados utilizando la librería *Jackson* para serializar/deserializar objetos en un archivo *JSON*.

- Agregar las dependencias de *Jackson* al proyecto.
- GestionEmpleados4.java → Realizar una copia de GestionEmpleados.java y modificar los siguientes métodos de la clase EmpleadosDAO.

```

private void cargarEmpleadosDesdeArchivo() {
    ObjectMapper objectMapper = new ObjectMapper(); // Crear un objeto ObjectMapper
    try { // Deserializar la lista de empleados desde el archivo JSON
        empleados = objectMapper.readValue(new File(archivo), new TypeReference<>() { });
    } catch (IOException e) {
        System.out.println(Utiles.ROJO + "Error: " + e.getMessage() + Utiles.RESET);
    }
}

private void guardarEmpleadosEnArchivo() {
    ObjectMapper objectMapper = new ObjectMapper(); // Crear un objeto ObjectMapper
    try { // Serializar la lista de empleados a un archivo JSON
        objectMapper.writeValue(new File(archivo), empleados);
    } catch (IOException e) {
        System.out.println(Utiles.ROJO + "Error: " + e.getMessage() + Utiles.RESET);
    }
}

```

- Añadir las siguientes importaciones.

```

import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;

```

- Añadir a la clase Empleado el constructor por defecto.

```

public Empleado() { }

```

- Cambiar en la clase EmpleadosDAO “empleados.dat” por “empleados.json”.

Documentación oficial de Jackson en <https://github.com/FasterXML/jackson-docs>

**Ejemplo:** uso de la **biblioteca Gson** para trabajar con documentos *JSON* en *Java*.

1. **Agregar la dependencia de Gson en el proyecto.** Lo más fácil es **crear un proyecto con Maven** y agregar al **archivo pom.xml** las siguientes líneas:

```
<dependencies>
  <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.11.0</version> <!-- Utilizar la versión más reciente en Maven Central -->
  </dependency>
</dependencies>
```

2. **Crear un objeto JSON utilizando la clase Gson de Gson:**

```
import com.google.gson.Gson;
import com.google.gson.JsonSyntaxException;

public class GsonEjemplo1 {
    public static void main(String[] args) {
        Gson gson = new Gson();
        String jsonString = "{\"nombre\": \"Juan\", \"edad\": 25}"; // Crear un objeto JSON

        try {
            Persona persona = gson.fromJson(jsonString, Persona.class);
            System.out.println("Nombre: " + persona.getNombre());
            System.out.println("Edad: " + persona.getEdad());
        } catch (JsonSyntaxException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

class Persona {
    private String nombre;
    private int edad;

    public String getNombre() { return nombre; }
    public int getEdad() { return edad; }
}
```

3. **Convertir un objeto Java a formato JSON:**

```
import com.google.gson.Gson;

public class GsonEjemplo2 {
    public static void main(String[] args) {
        Gson gson = new Gson();
        Persona persona = new Persona("María", 30);

        try {
            String jsonString = gson.toJson(persona);
            System.out.println(jsonString);
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

class Persona {
    private String nombre;
    private int edad;

    public Persona() { }

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

Documentación oficial de Gson en <https://github.com/google/gson>

**Ejemplo:** uso de *Gson* para serializar y deserializar una lista de personas.

```
import com.google.gson.Gson;
import java.io.FileReader;
```

```

import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

class Persona {
    private String nombre;
    private int edad;

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

    public String getNombre() { return nombre; }
    public int getEdad() { return edad; }
}

public class GsonListaPersonas {
    private List<Persona> personas = new ArrayList<>();

    public List<Persona> getPersonas() { return personas; }

    public static void main(String[] args) {
        GsonListaPersonas listaPersonas = new GsonListaPersonas();

        listaPersonas.getPersonas().add(new Persona("Ana", 30));
        listaPersonas.getPersonas().add(new Persona("Carlos", 25));

        Gson gson = new Gson();

        try (FileWriter writer = new FileWriter("personas.json")) {
            gson.toJson(listaPersonas, writer);
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }

        try (FileReader reader = new FileReader("personas.json")) {
            GsonListaPersonas listaPersonasLeido = gson.fromJson(reader, GsonListaPersonas.class);
            listaPersonasLeido.getPersonas().forEach(p -> System.out.println(p.getNombre() +
                                                                              " - " + p.getEdad()));
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

```

**Ejemplo:** uso de la biblioteca *Gson* para leer un archivo *JSON*, modificar su contenido y luego guardarlo.

Suponer que se tiene un archivo llamado "persona.json" (poner en la raíz del proyecto) con el siguiente contenido:

```
{ "nombre": "Juan", "edad": 25 }
```

Y se quiere cambiar la edad a 30. Código *Java*:

```

import com.google.gson.Gson;
import com.google.gson.JsonObject;
import com.google.gson.JsonParser;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class GsonFileEjemplo {
    public static void main(String[] args) {
        try (FileReader fileReader = new FileReader("persona.json")) {
            JsonObject jsonObject = JsonParser.parseReader(fileReader).getAsJsonObject();
            jsonObject.addProperty("edad", 30);

            try (FileWriter fileWriter = new FileWriter("persona_modificado.json")) {
                new Gson().toJson(jsonObject, fileWriter);
            }

            System.out.println("Archivo JSON modificado y guardado.");
        } catch (IOException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}

```

Este código primero lee el archivo "persona.json", luego modifica el valor de *edad* y finalmente guarda el nuevo contenido en un archivo llamado "persona\_modificado.json".

## 12 Convertir un archivo *XML* a *JSON*.

Para convertir un archivo *XML* a *JSON* en *Java* utilizando la biblioteca *Gson*, primero se debe leer el archivo *XML* y luego realizar la conversión a *JSON*.

Ejemplo: suponer que se tiene un archivo *XML* llamado "personas.xml".

```
<personas>
  <persona>
    <nombre>Juan</nombre>
    <edad>30</edad>
    <ciudad>Nueva York</ciudad>
  </persona>
  <persona>
    <nombre>Alicia</nombre>
    <edad>28</edad>
    <ciudad>Los Ángeles</ciudad>
  </persona>
</personas>
```

Crear un proyecto con *Maven* y agregar las siguientes dependencias al archivo pom.xml:

```
<dependencies>
  <dependency> <!-- Gson: Para la conversión de JSON -->
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.11.0</version> <!-- Usar la versión más reciente -->
  </dependency>
  <dependency> <!-- JSON-java (org.json): Para la conversión de JSON -->
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20240303</version> <!-- Usar la versión más reciente -->
  </dependency>
</dependencies>
```

El siguiente código muestra cómo convertir un archivo *XML* en formato *JSON*:

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Map;
import org.json.JSONObject;
import org.json.XML;

public class XmlToJsonConverter {
    public static void main(String[] args) {
        String filePath = "personas.xml"; // Ruta del archivo XML
        StringBuilder xmlContenido = new StringBuilder(); // Para almacenar el contenido del archivo

        // Leer el archivo usando BufferedReader
        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String linea;
            while ((linea = reader.readLine()) != null)
                xmlContenido.append(linea);

            // Convertir XML a un objeto JSONObject utilizando org.json
            JSONObject objetoJSON = XML.toJSONObject(xmlContenido.toString());
            // Convertir el objeto JSONObject a un Map
            Map<String, Object> jsonMap = objetoJSON.toMap();
            // Crear un objeto Gson con formato bonito (pretty printing)
            Gson gson = new GsonBuilder().setPrettyPrinting().create();
            // Convertir el Map a una cadena JSON con formato
            String salidaJSON = gson.toJson(jsonMap);
            // Imprimir el JSON resultante
            System.out.println(salidaJSON);
        } catch (IOException e) {
            System.out.println("Error al leer el archivo XML: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Error durante la conversión de XML a JSON: " + e.getMessage());
        }
    }
}
```



El resultado *JSON* para el archivo "personas.xml" será:

```
{
  "personas": {
    "persona": [
      {
        "nombre": "Juan",
        "edad": 30,
        "ciudad": "Nueva York"
      },
      {
        "nombre": "Alicia",
        "edad": 28,
        "ciudad": "Los Ángeles"
      }
    ]
  }
}
```



## TAREA

22. (Opcional) Crear una clase llamada Persona con los campos nombre, edad y dirección. Serializar una instancia de Persona a un archivo *JSON* y luego deserializarla para recuperar la información.
23. Definir una clase Libreria que contenga una lista de libros. Cada libro debe tener las propiedades título, autor y año de publicación. Crear una lista de libros, serializarla a *JSON* y luego deserializarla para recuperar la lista.

## 13 Control de excepciones.

### 13.1 Excepciones en Java.

Una excepción es un evento que ocurre durante la ejecución de un programa y detiene el flujo normal del código. El control de excepciones permite detectar y corregir estos errores, evitando que la aplicación se cierre inesperadamente.

Existen **dos tipos principales de excepciones**:

- ✓ **Excepciones comprobadas** (checked exceptions): **deben ser declaradas en la firma del método con throws o capturadas con try-catch.**
- ✓ **Excepciones no comprobadas** (unchecked exceptions): no requieren ser declaradas o capturadas explícitamente.

### 13.2 Manejo de excepciones comprobadas.

Hay **dos formas principales de manejar** excepciones comprobadas:

- **Declarar la excepción en la firma del método**: si un método puede lanzar una excepción comprobada, debe declararse **usando throws**.

Se debe utilizar el siguiente **formato**:

```
public static tipoDeDato identificadorDelMétodo(parámetros_formales) throws ClaseExcepción {
    // Cuerpo del método. Puede contener instrucciones y/o return
}
```

Si ocurre una excepción, esta se propagará al método que llamó, y debe ser manejada ahí con **try-catch** o seguir propagándola.

Ejemplo:

```
public class Main {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
            System.out.println("Resultado: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error aritmético: " + e.getMessage());
        }
    }
}
```

```

    }
}

public static int divide(int a, int b) throws ArithmeticException {
    if (b == 0)
        throw new ArithmeticException("División entre cero");
    return a / b;
}
}

```

En este ejemplo, una división por cero lanza una `ArithmeticException`, que se captura en el bloque `catch` del método `main`.

- **Capturar la excepción con try-catch-finally:** envuelve el código susceptible de errores en un bloque `try`. El bloque `catch` captura y maneja las excepciones, y el bloque `finally` (opcional) se ejecuta siempre, haya ocurrido o no una excepción.

Se debe utilizar el siguiente **formato**:

```

try {
    // Código que puede generar una excepción
} catch (ClaseExcepción variableRecogeExcepción) {
    // Manejo de la excepción
} finally {
    // Código que se ejecuta siempre (opcional)
}

```

Ejemplo:

```

public class Main {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
            System.out.println("Resultado: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error aritmético: " + e.getMessage());
        }
    }

    public static int divide(int a, int b) {
        try {
            if (b == 0)
                throw new ArithmeticException("División entre cero");
            return a / b;
        } catch (ArithmeticException e) {
            System.out.println("Excepción capturada en divide: " + e.getMessage());
            return 0; // Valor por defecto o manejo alternativo
        }
    }
}

```

En este ejemplo, el manejo de la excepción se realiza dentro del método `divide`, capturando la excepción y retornando un valor alternativo.

La elección entre manejar excepciones declarando `throws` o usando `try-catch` depende del nivel de control y del contexto del programa. Usar `throws` permite propagar la excepción hacia los métodos superiores, mientras que `try-catch` maneja la excepción localmente.

Además, **Java ofrece try-with-resources para cerrar automáticamente recursos** como archivos, lo que facilita el manejo seguro de recursos y evita fugas.

De forma general, la clase `Exception` recoge todos los tipos de excepciones. Si se desea un control más exhaustivo del tipo de error que se produce, se debe concretar más la clase de excepción correspondiente.

Por ejemplo, cuando se intenta convertir al tipo de dato numérico entero un dato introducido por el usuario en un campo de texto se utiliza una sentencia como la que se muestra en el ejemplo. Si el valor introducido no es numérico, sino una cadena de caracteres, la llamada a `Integer.parseInt` produce una excepción, como se puede apreciar en la salida estándar:

```

EjemploException.java
Source History
1 package com.jlmr.ejemploexception;
2
3 import java.util.Scanner;
4
5 public class EjemploException {
6     public static void main(String[] args) {
7         Scanner sc = new Scanner(System.in);
8
9         System.out.print("Introduce un número: ");
10        int num = Integer.parseInt(sc.nextLine());
11
12        sc.close();
13    }
14
15 }

```

```

Output - Run (EjemploException)
Introduce un número: 12a
Exception in thread "main" java.lang.NumberFormatException: For input string: "12a"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:781)
    at com.jlmr.ejemploexception.EjemploException.main(EjemploException.java:10)
Command execution failed.
org.apache.commons.exec.DefaultExecutor: Process exited with an error: 1 (Exit value: 1)
at org.apache.commons.exec.DefaultExecutor.executeInternal(DefaultExecutor.java:404)
at org.apache.commons.exec.DefaultExecutor.execute(DefaultExecutor.java:166)
at org.codehaus.mojo.exec.ExecMojo.executeCommandLine(ExecMojo.java:982)

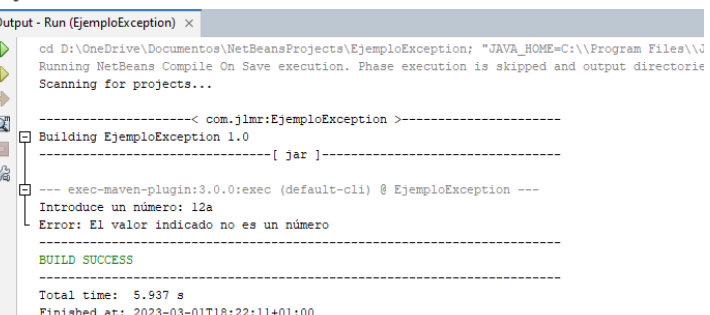
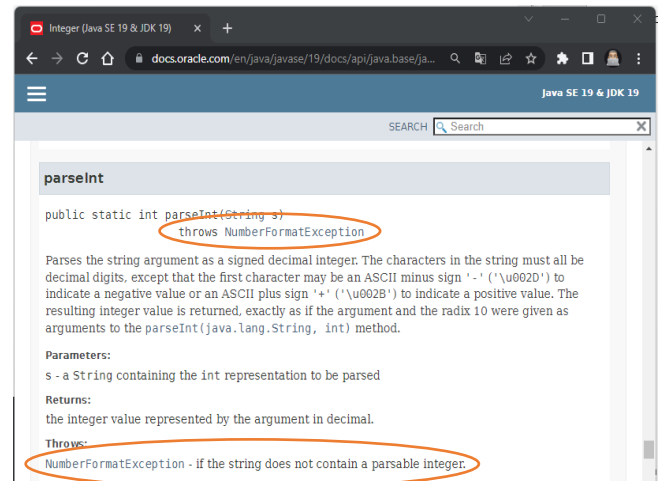
```

Se puede apreciar que se produce una excepción del tipo `NumberFormatException`, por tanto, se debería capturar esa excepción para controlar el error.

Si se vuelve a ejecutar el programa y se introduce un valor que no sea un número entero el programa no terminará de forma anormal (ver imagen).

En la **documentación de la API** se puede encontrar el tipo de excepción que lanza (throws) un determinado método. Para ello, observar si al final de la declaración del método aparece la sentencia `throws`. Por ejemplo, en el caso del método `parseInt` de la clase `Integer` se muestra:

```
try {
    int num = Integer.parseInt(s:sc.nextLine());
} catch (NumberFormatException e) {
    System.out.println(x: "Error: El valor indicado no es un número");
}
```

Si no se tiene claro el tipo de excepción o se desea manejar cualquier error, se puede usar la clase genérica `Exception` en el bloque `catch`.

```
try {
    int num = Integer.parseInt(s:sc.nextLine());
} catch (Exception e) {
    System.out.println(x: "Se ha detectado un error");
}
```

Ejemplo con finally:

```
public static void main(String[] args) {
    try {
        int d = 0;
        int n = 20;
        int fracción = n / d;
    } catch (ArithmeticException e) {
        System.out.println("En el reloj catch debido a Exception =" + e);
    } finally {
        System.out.println(x: "Dentro del bloque finally");
    }
}
```

Ejemplo completo:

```
public static void main(String[] args) {
    /* Se declaran dos variables para almacenar dos números
    y una tercera para almacenar el resultado de una división */
    double primerNumero, segundoNumero, resultado;

    // Se declara un objeto Scanner para leer los datos
    Scanner lectura = new Scanner( source: System.in);

    try { // Se intenta pedir los números al usuario
        System.out.println(x: "Introduce el primer número");
        primerNumero = lectura.nextDouble();

        System.out.println(x: "Introduce el segundo número");
        segundoNumero = lectura.nextDouble();

        // y dividir los dos números
        resultado = primerNumero / segundoNumero;

        // Se muestra el resultado por pantalla.
        System.out.println("El resultado es " + resultado);
    } catch (ArithmeticException e) { /* En caso de que surja algún error
    de tipo aritmético. Ej.: que no se puede dividir entre cero */
        // Se muestra este mensaje
        System.out.println(x: "No se puede dividir entre cero");
    } catch (Exception ex) { /* En caso de que la excepción no sea de
    tipo aritmético, se ejecutará este bloque */
        /* Se muestra este mensaje en caso de que no
        sea una excepción de tipo aritmético */
        System.out.println(x: "Se ha producido un error");
    } finally { /* Tanto si se produce la excepción, como si no se produce,
    se ejecutará el bloque finally */
        lectura.close();
    }
}
```

## 13.3 Obtener mensaje de la excepción.

Cuando se produce una excepción en un bloque try, la variable especificada en catch captura información sobre esa excepción, comúnmente nombrada como e o ex. Para mostrar el mensaje de error, se pueden usar los métodos getMessage() o toString() de la variable capturada:

```
System.out.println(e.getMessage());
System.out.println(e.toString());
```

Al usar try, se pierde el mensaje de error estándar que incluye información sobre la línea donde ocurrió la excepción, útil durante el desarrollo. Para obtener el mensaje completo y la pila de llamadas, se usa el método printStackTrace.

```
e.printStackTrace();
```

```

EjemploException.java
Source History
Scanner sc = new Scanner( source: System.in);
System.out.print( s: "Introduce un número: ");
try {
    int num = Integer.parseInt( s: sc.nextLine());
} catch(Exception e) {
    System.out.println( s: "Se ha detectado un error");
    System.out.println( s: e.getMessage());
    e.printStackTrace();
}

Output - Run (EjemploException)
cd D:\OneDrive\Documentos\NetBeansProjects\EjemploException; "JAVA_HOME=C:\Program Files\Java\jdk-
Running NetBeans Compile On Save execution. Phase execution is skipped and output directories of depe
Scanning for projects...

-----< com.jlmr:EjemploException >-----
Building EjemploException 1.0
-----[ jar ]-----

--- exec-maven-plugin:3.0.0:exec (default-cli) @ EjemploException ---
Introduce un número: 12a
Se ha detectado un error
For input string: "12a"
java.lang.NumberFormatException: For input string: "12a"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:665)
    at java.base/java.lang.Integer.parseInt(Integer.java:781)
  
```

## 13.4 Uso obligatorio de try-catch.

En los ejemplos previos, se controlaron excepciones de métodos como Integer.parseInt(), aunque no es obligatorio hacerlo. Se utilizó try-catch para gestionar entradas incorrectas.

Sin embargo, **algunos métodos exigen obligatoriamente try-catch**. Este requisito se detecta por errores de compilación como:

```
error: unreported exception xxxxxxxxxxException; must be caught or declared to be thrown
```

*En muchos casos, las excepciones son obligatorias, por ejemplo, al manejar archivos, acceso a BD, etc.*

La clase FileReader permite obtener el contenido de archivos alojados en el almacenamiento local del equipo. Uno de los métodos constructores tiene la siguiente declaración, como puede verse en la API:

```

FileReader
public FileReader(File file)
    throws FileNotFoundException
Creates a new FileReader, given the File to read, using the default charset.
Parameters:
file - the File to read
Throws:
FileNotFoundException - if the file does not exist, is a directory rather than a regular file, or
for some other reason cannot be opened for reading.
See Also:
Charset.defaultCharset()
  
```

Si en una aplicación se incluye el siguiente trozo de código sin encerrar en una estructura try-catch se producirá el error anterior.

```

lic class EjemploException {
    public static void main(String[] args) {
        unreported exception FileNotFoundException; must be caught or declared to be thrown
        (Alt-Enter shows hints)
    }
}
  
```

En NetBeans, si se hace clic en la bombilla del margen izquierdo, se podrá encerrar automáticamente esta sentencia dentro de un bloque try-catch usando la opción Surround Statement with try-catch:

```

public static void main(String[] args) {
    FileReader fr = new FileReader( fileName: "archivo.txt");
    Add throws clause for java.io.FileNotFoundException
    Surround Statement with try-catch
  
```

Se obtendrá el siguiente resultado:

```

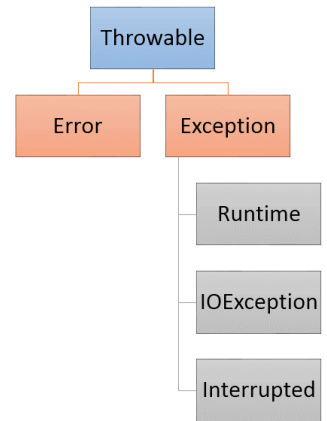
try {
    FileReader fr = new FileReader( fileName: "archivo.txt");
} catch (FileNotFoundException ex) {
    Logger.getLogger( name: EjemploException.class.getName()).log( level: Level.SEVERE, msg: null, thrown: ex);
}
  
```

Por defecto se utiliza Logger para mostrar mensajes de excepciones, aunque puede usarse cualquier método de salida.

## 13.5 Jerarquía de clases de excepciones.

Después de ejecutar un bloque `catch`, la ejecución continúa después del `try-catch`.

Las excepciones en *Java* siguen una jerarquía, con todas las excepciones extendiendo de la clase `Throwable`, que tiene dos subclases:



- ✓ **Error**: indica problemas graves como errores de memoria o del JVM.
- ✓ **Exception**: maneja errores que se pueden recuperar con `try-catch`.
  - **RuntimeException**: se lanza en tiempo de ejecución y no se detecta en compilación, como la división por cero o `NullPointerException`.
  - **IOException**: se lanza durante operaciones de entrada/salida.
  - **InterruptedException**: ocurre en el manejo de hilos.

## 13.6 Excepciones más frecuentes.

Algunas de las excepciones más comunes en Java son:

- **ArithmeticException**: se lanza cuando ocurre un error aritmético, como la división entre cero.

```

try {
    int resultado = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error aritmético: " + e.getMessage());
}
  
```

- **NullPointerException**: se lanza cuando se intenta acceder a un objeto que tiene un valor nulo (`null`).

```

try {
    String text = null;
    int length = text.length();
} catch (NullPointerException e) {
    System.out.println("Referencia nula: " + e.getMessage());
}
  
```

- **ArrayIndexOutOfBoundsException**: se lanza cuando se intenta acceder a un índice no válido de un array.

```

try {
    int[] numbers = { 1, 2, 3 };
    int value = numbers[5];
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Índice de arreglo inválido: " + e.getMessage());
}
  
```

- **FileNotFoundException**: se lanza cuando no se puede encontrar un archivo en una operación de lectura o escritura.

```

try {
    File file = new File("archivo.txt");
    FileReader reader = new FileReader(file);
} catch (FileNotFoundException e) {
    System.out.println("Archivo no encontrado: " + e.getMessage());
}
  
```

- **IOException**: se lanza cuando se produce un error de entrada/salida durante la lectura o escritura de datos.

```

try {
    BufferedReader reader = new BufferedReader(new FileReader("archivo.txt"));
    String line = reader.readLine();
} catch (IOException e) {
    System.out.println("Error de E/S: " + e.getMessage());
}
  
```

Ejemplo: media de dos números.

```

import java.util.Scanner;

public class MediaDosNumeros01 {
    public static void main(String[] args) {
        double numero1, numero2;
        Scanner sc = new Scanner(System.in);

        System.out.println("***** Media de dos números *****");
        try {
  
```

```

***** Media de dos números *****
Introduce el primer número: 23
Introduce el segundo número: sdfs
Datos introducidos incorrectos.
Finaliza el programa.
  
```

```

        System.out.print("Introduce el primer número: ");
        numero1 = sc.nextDouble();
        System.out.print("Introduce el segundo número: ");
        numero2 = sc.nextDouble();
        System.out.println("La media es " + (numero1 + numero2) / 2);
    } catch (Exception e) {
        System.out.println("Datos introducidos incorrectos.");
    } finally {
        sc.close();
        System.out.println("Finaliza el programa.");
    }
}
}

```

**Ejemplo:** media de dos números, repitiendo la solicitud de datos si no se introducen correctamente.

```

import java.util.Scanner;

public class MediaDosNumeros02 {
    public static void main(String[] args) {
        double numero1 = 0, numero2 = 0;
        boolean valido = false;
        Scanner sc = new Scanner(System.in);

        System.out.println("***** Media de dos números *****");
        do {
            try {
                System.out.print("Introduce el primer número: ");
                numero1 = Double.parseDouble(sc.nextLine());
                valido = true;
            } catch (Exception e) {
                System.out.println("Se debe introducir un número. Prueba de nuevo.");
            }
        } while (!valido);
        valido = false;
        do {
            try {
                System.out.print("Introduce el segundo número: ");
                numero2 = Double.parseDouble(sc.nextLine());
                valido = true;
            } catch (Exception e) {
                System.out.println("Se debe introducir un número. Prueba de nuevo.");
            }
        } while (!valido);
        sc.close();
        System.out.println("La media es " + (numero1 + numero2) / 2);
    }
}

```

**Ejemplo:** división de dos números controlando la división por cero.

```

import java.util.InputMismatchException;
import java.util.Scanner;

public class DivisionDosNumeros {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int numero1, numero2;

        try {
            System.out.print("Introduce el primer número: ");
            numero1 = scanner.nextInt();
            System.out.print("Introduce el segundo número: ");
            numero2 = scanner.nextInt();

            int resultado = dividir(numero1, numero2);
            System.out.println("El resultado de la división es: " + resultado);
        } catch (InputMismatchException e) {
            System.out.println("Error: Ingrese solo números enteros.");
        } catch (ArithmeticException e) {
            System.out.println("Error: No se puede dividir entre cero.");
        } finally {
            scanner.close();
            System.out.println("Fin del programa.");
        }
    }
}

```



```

        public static int dividir(int a, int b) {
            if (b == 0)
                throw new ArithmeticException("División entre cero");
            return a / b;
        }
    }
}

```

En este ejemplo, se utiliza la clase `Scanner` para leer dos números enteros ingresados por el usuario. Se manejan dos posibles excepciones: `InputMismatchException`, que se lanza si el usuario ingresa un valor no numérico, y `ArithmeticException`, que se lanza si el segundo número ingresado es cero.

El bloque `try` envuelve el código que puede lanzar las excepciones, y las captura en los bloques `catch` correspondientes, mostrando un mensaje de error adecuado en cada caso. El bloque `finally` se ejecuta siempre, independientemente de si ocurre una excepción o no, y se utiliza para cerrar el objeto `Scanner` y mostrar un mensaje de finalización del programa.

Es importante manejar las excepciones correctamente para asegurar un comportamiento controlado y robusto de la aplicación.

**Ejemplo:** división de dos números controlando la división por cero utilizando una excepción personalizada.

```

import java.util.InputMismatchException;
import java.util.Scanner;

class DivisionPorCeroException extends Exception {
    public DivisionPorCeroException(String mensaje) {
        super(mensaje);
    }
}

public class ExcepcionPropia {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int numero1, numero2;

        try {
            System.out.print("Introduce el primer número: ");
            numero1 = scanner.nextInt();
            System.out.print("Introduce el segundo número: ");
            numero2 = scanner.nextInt();

            int resultado = dividir(numero1, numero2);
            System.out.println("El resultado de la división es: " + resultado);
        } catch (InputMismatchException e) {
            System.out.println("Error: Ingrese solo números enteros.");
        } catch (DivisionPorCeroException e) {
            System.out.println(e.getMessage());
        } finally {
            scanner.close();
        }
    }

    public static int dividir(int a, int b) throws DivisionPorCeroException {
        if (b == 0)
            throw new DivisionPorCeroException("ERROR: División por cero. Segundo número igual a 0.");
        return a / b;
    }
}

```

En este ejemplo, se define una clase llamada `DivisionPorCeroException`, que extiende de la clase `Exception` y representa una excepción personalizada para manejar la división por cero. La clase incluye un constructor que recibe un mensaje como argumento y lo pasa al constructor de la clase base `Exception`.

En el método `dividir`, se verifica si el denominador es cero. Si lo es, se lanza una instancia de `DivisionPorCeroException` con un mensaje descriptivo.

En el método `main`, se llama a la función `dividir`, proporcionando un numerador y denominador. Si el denominador es cero, se lanza la excepción `DivisionPorCeroException`, la cual se captura mediante un bloque `try-catch` y se muestra un mensaje de error adecuado.

## 14 Depuración y documentación.

## 14.1 Depuración de programas con *NetBeans*.

*La depuración es esencial para encontrar y corregir errores (bugs) en el código. Existen diversas herramientas y técnicas para facilitar este proceso, principalmente a través de los IDEs como NetBeans, Eclipse e IntelliJ IDEA, que ofrecen funciones integradas para depurar programas Java.*

### Guía para depurar programas *Java*:

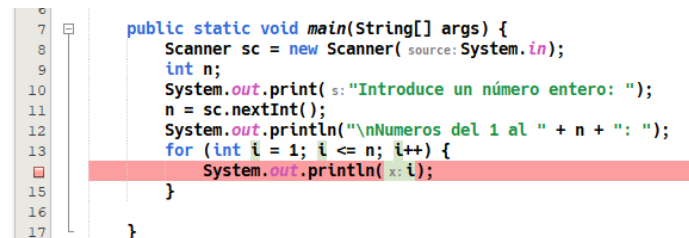
- ✓ **Utilizar un IDE:** los IDEs populares tienen herramientas de depuración que permiten establecer puntos de interrupción (*breakpoints*), inspeccionar variables y seguir la ejecución del código paso a paso.
- ✓ **Establecer puntos de interrupción:** un punto de interrupción indica al depurador que detenga la ejecución en un punto específico, permitiendo examinar el estado del programa y sus variables. Se pueden establecer desde el margen izquierdo del código o con atajos de teclado.
- ✓ **Ejecución paso a paso:** avanzar a través del código línea por línea para observar su ejecución y detectar problemas.
- ✓ **Inspeccionar variables:** examinar los valores de las variables en tiempo de ejecución para verificar si son los esperados y detectar errores de cálculo.
- ✓ **Mostrar trazas de pila (*stack traces*):** ante una excepción, el depurador muestra en qué parte del código ocurrió el error, facilitando su rastreo.
- ✓ **Utiliza registros (*logs*):** agregar declaraciones de registro para imprimir mensajes de depuración y controlar cuánta información se muestra.
- ✓ **Prueba con datos de entrada específicos:** los errores pueden ocurrir solo con ciertos datos; probar con entradas problemáticas para identificar y resolver errores.
- ✓ **Utilizar herramientas externas:** herramientas como *jdb* (*Java Debugger*) en la línea de comandos ofrecen capacidades de depuración avanzadas.
- ✓ **Leer y comprender los mensajes de error:** los mensajes de error suelen ser informativos y proporcionan pistas sobre la ubicación del problema.
- ✓ **Consultar la documentación:** si se usan bibliotecas o *frameworks*, revisar su documentación para obtener ayuda con problemas específicos.

*La depuración es una habilidad importante en el desarrollo de software y puede llevar tiempo. Sin embargo, con práctica y paciencia, te volverás más eficiente en la identificación y solución de errores en tu código Java.*

La **depuración** (*debug*) permite examinar los programas para **observar la ejecución línea por línea y los valores de las variables**.

Para depurar un programa, primero se debe **establecer un punto de interrupción** donde la ejecución de la aplicación se detendrá. Esto se puede hacer de las siguientes **maneras sobre la línea de código** deseada:

- ✓ **Clic en el margen izquierdo.**
- ✓ Menú contextual → *Toggle Line Breakpoint*.
- ✓ Pulsando la combinación de teclas: **[Ctrl] + [F8]**.
- ✓ Menú *Debug* → *Toggle Line Breakpoint*.



```

7
8
9
10
11
12
13
14
15
16
17


public static void main(String[] args) {
    Scanner sc = new Scanner( source: System.in);
    int n;
    System.out.print( s: "Introduce un número entero: ");
    n = sc.nextInt();
    System.out.println("\nNumeros del 1 al " + n + ": ");
    for (int i = 1; i <= n; i++) {
        System.out.println( x: i);
    }
}

```

Al establecer un punto de interrupción, **la línea se resalta en rosado y aparece un pequeño cuadrado en el margen izquierdo** (■).


Una vez establecido al menos un punto de interrupción, se debe **ejecutar la aplicación en modo depuración**, ya sea sobre el proyecto completo o solo sobre el archivo actual:

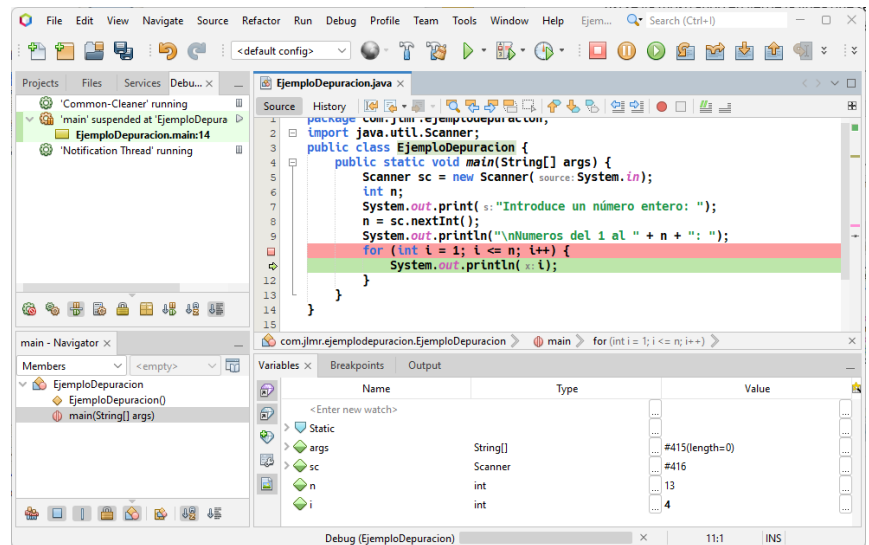
- **Depurar archivo actual:**
  - Menú contextual → *Debug file*.
  - Menú *Debug* → *Debug file*.

- Combinación de teclas: **[Ctrl] + [Mayús] + [F5]**.
- **Depurar proyecto:**
  - Menú **Debug** → **Debug Project (NombreProyecto)**.
  - Icono **Debug Project** ().

Cuando la ejecución alcanza un punto de interrupción, la línea de código se destaca en color verde. En la parte inferior del IDE se muestra la ventana de variables locales, donde se pueden observar los valores de las variables en tiempo real.

Para continuar con la ejecución línea por línea, utilizar la opción **Step over**:

- ✓ Pulsar la tecla **[F8]**.
- ✓ Menú **Debug** → **Step over**.
- ✓ Icono **Step over** ().



La línea actual se resaltarán en verde y los valores de las variables se actualizarán en la ventana inferior conforme avanza la ejecución.

Si se desea que el programa continúe sin más interrupciones, usar la opción **Continue**:

- ✓ Pulsando la tecla **[F5]**.
- ✓ Menú **Debug** → **Continue**.
- ✓ Icono **Continue** ().

## 14.2 Documentación.

Tradicionalmente, se ha usado **JavaDoc** para la documentación en **Java**. Sin embargo, las versiones más recientes del lenguaje también incluyen anotaciones.

Los comentarios **JavaDoc** están diseñados para describir clases y métodos. Comienzan con **/\*\***, pueden extenderse a varias líneas (con el carácter **\*** al principio de cada línea) y terminan con **\*/**.

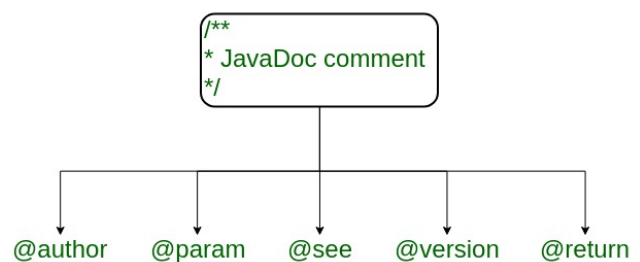
*Es importante añadir explicaciones a todo lo que no sea evidente. Pero, no hay que repetir lo que se hace, sino explicar por qué se hace.*

Dado que estos comentarios están destinados a ser leídos por otros programadores, se ha establecido un formato común para asegurar su legibilidad. Los comentarios **JavaDoc** deben incluir etiquetas especiales que comienzan con **@**, como **@author**, **@param**, **@return**, **@deprecated**, **@see**, **@version**, ... Estas etiquetas suelen colocarse al principio de una línea y ayudan a proporcionar información específica sobre el código documentado.

Ejemplo:

```
/**
 * Una clase para representar círculos situados sobre el plano. Cada círculo
 * queda determinado por su radio junto con las coordenadas de su centro.
 * @version 2.2, 10/09/2024
 * @author Nombre Apellidos
 */
public class CirculoDocumentada {
    protected double x, y; // Coordenadas del centro.
    protected double r; // Radio del círculo.

    /**
     * Crea un círculo a partir de su origen y su radio.
```



```

* @param x La coordenada x del centro del círculo.
* @param y La coordenada y del centro del círculo.
* @param r El radio del círculo. Debe ser mayor o igual a 0.
*/
public CirculoDocumentada(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
}

/**
 * Cálculo del área de este círculo.
 * @return El área (mayor o igual que 0) del círculo.
 */
public double area() {
    return Math.PI * r * r;
}

/**
 * Indica si un punto está dentro del círculo.
 * @param px componente x del punto.
 * @param py componente y del punto.
 * @return true si el punto está dentro del círculo o false en otro caso.
 */
public boolean contiene(double px, double py) {
    /* Calculamos la distancia de (px,py) al centro del círculo (x,y),
       que se obtiene como raíz cuadrada de (px-x)^2+(py-y)^2. */
    double d = Math.sqrt((px - x) * (px - x) + (py - y) * (py - y));
    return d <= r; // El círculo contiene el punto si d es menor o igual al radio.
}
}

```

Una **anotación** es un **texto que puede utilizar otras herramientas** (no solo el *JavaDoc*) **para comprender mejor qué hace ese código o como documentarlo**.

Cualquiera puede crear sus propias anotaciones simplemente definiéndolas como una interfaz *Java*. Sin embargo, se tendrán que programar clases propias para extraer la información que proporcionan dichas anotaciones.

Una anotación en *Java* puede ser procesada por un generador de código fuente, por el compilador o por una herramienta de despliegue. En ocasiones, las **anotaciones en Java también son llamadas *metadata***, pero el término anotación es el más descriptivo y más utilizado.

#### Características generales de las anotaciones en Java:

- ✓ Las anotaciones *Java* **no cambian la actividad de un programa ordenado**.
- ✓ **Ayudan a relacionar metadatos** (datos) **con los componentes del programa**, es decir, constructores, estrategias, clases, etc.
- ✓ Las anotaciones en *Java* no son comentarios sin adulteraciones, ya que **pueden cambiar la forma en que el compilador trata el programa**.

#### Usos principales de las anotaciones en Java:

- ✓ **Información para el compilador**.
- ✓ **Procesamiento en tiempo de compilación y tiempo de implementación**.
- ✓ **Procesamiento de tiempo de ejecución**.

```

@Anotacion(elemento="informacion del elemento", orden=1)
@Override
@SuppressWarnings("unchecked")
public void metodoAnotado() { ... }

```

Más información: <https://www.oracle.com/es/technical-resources/articles/java/javadoc-tool.html>



## TAREA

**24. Documentar los ejercicios de la unidad si no se ha ido haciendo conforme se realizaban. Realizar la depuración del ejercicio 22 y adjuntar capturas de pantalla.**