UAX FP

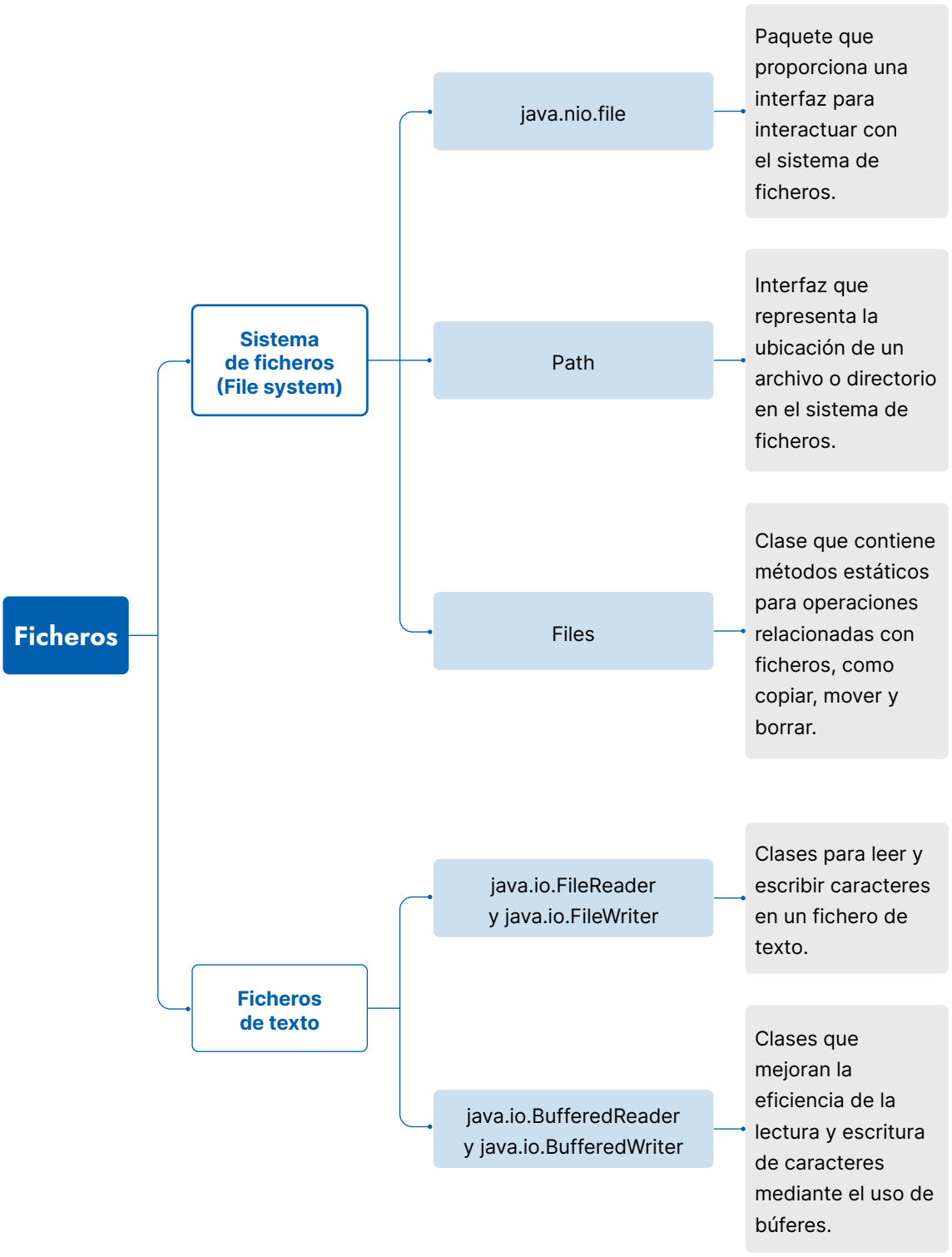
UF1

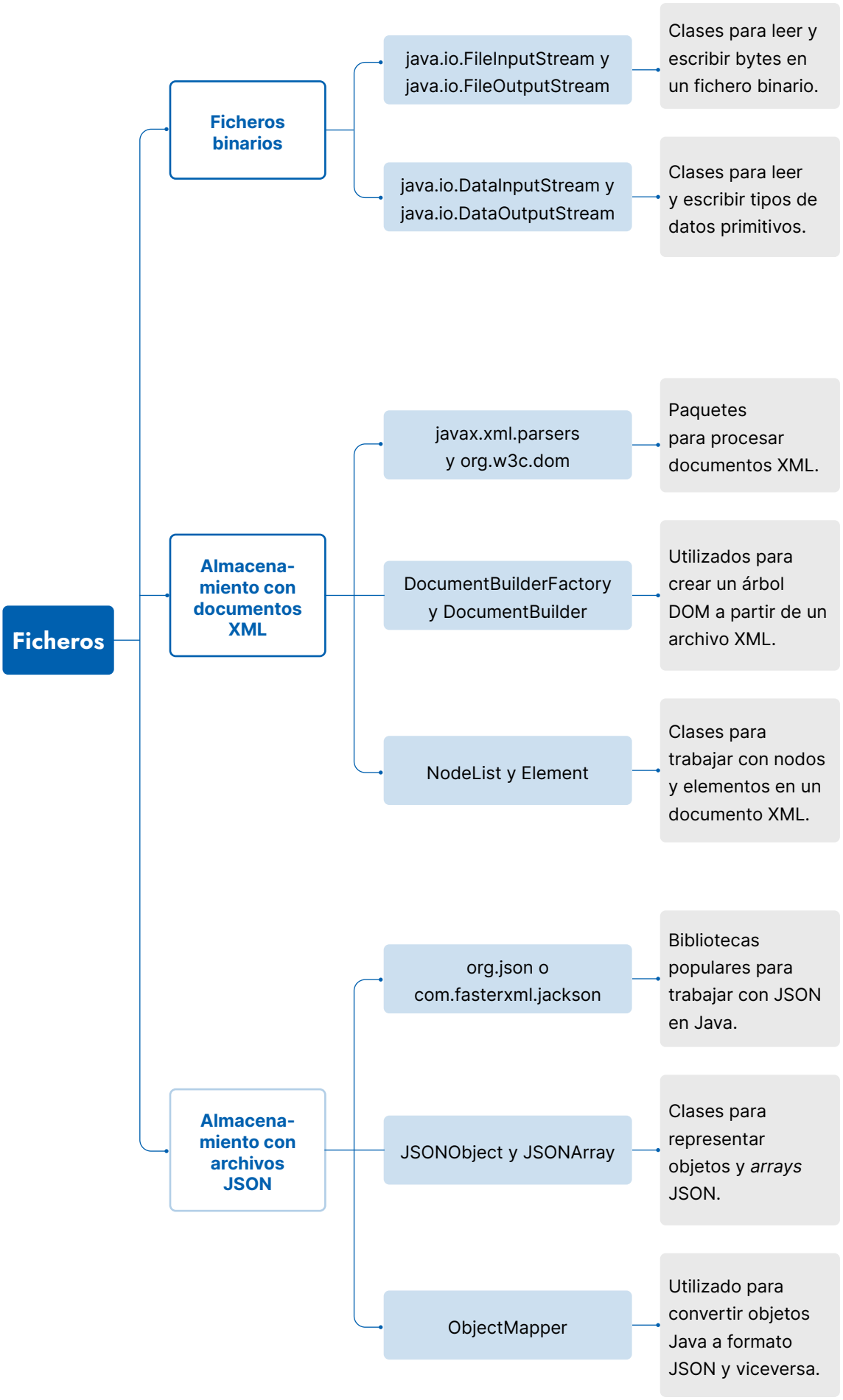
Sistemas de ficheros

Acceso a datos

ÍNDICE

Mapa conceptual




Ficheros
Ficheros binarios
java.io.FileInputStream y java.io.FileOutputStream
Clases para leer y escribir bytes en un fichero binario.
java.io.DataInputStream y java.io.DataOutputStream
Clases para leer y escribir tipos de datos primitivos.
Almacena-miento con documentos XML
javax.xml.parsers y org.w3c.dom
Paquetes para procesar documentos XML.
DocumentBuilderFactory y DocumentBuilder
Utilizados para crear un árbol DOM a partir de un archivo XML.
NodeList y Element
Clases para trabajar con nodos y elementos en un documento XML.
Almacena-miento con archivos JSON
org.json o com.fasterxml.jackson
Bibliotecas populares para trabajar con JSON en Java.
JSONObject y JSONArray
Clases para representar objetos y arrays JSON.
ObjectMapper
Utilizado para convertir objetos Java a formato JSON y viceversa.

01 Sistema de ficheros

Desde los albores de la informática, la información se ha almacenado de manera permanente en ficheros. Aunque hoy en día los mayores volúmenes de datos se almacenan en bases de datos, se siguen almacenando tanto fotos como vídeos o una ingente cantidad de formatos en ficheros, que pueden ser utilizados como mecanismo de intercambio de información. Veremos a continuación cómo podemos procesar estos archivos, almacenados en nuestro sistema.

1.1. Gestión y organización de sistema de ficheros

Los sistemas operativos gestionan los dispositivos de almacenamiento de manera casi transparente para el usuario. Sin entrar en detalles de las tecnologías de almacenamiento subyacentes (tipos de dispositivos: discos duros, SSD, tarjetas externas) o del formato del sistema de ficheros (**NTFS**, **ext3**, **NFS**, **APFS**, etc.), estos nos ofrecen una abstracción donde solamente nos hemos de preocupar de dos conceptos: los ficheros y los directorios.

DEFINICIÓN

Entendemos los **ficheros** como los contenedores de información, mientras que los **directorios** son los organizadores de los ficheros, que pueden contener ficheros y otros directorios.

Debemos tener en cuenta el tipo de información que contendrá el fichero, y podemos clasificarlos en ficheros de texto o ficheros binarios:

- Los **ficheros de texto** son aquellos que contienen información en una codificación interpretable por los editores de texto básicos, como **vi**, **nano** o **emacs**, e incluso interpretable por los **navegadores web**.
- Los **ficheros binarios** son aquellos cuya información se guarda codificada, representando y almacenando información de cualquier naturaleza, como imágenes, vídeo, datos, etc. Aun así, la información es accesible, aunque en texto plano puede guardarse ocultada, mediante algoritmos de encriptación, como, por ejemplo, los ficheros de **password** de Apache o los ficheros de certificados.

1.2. Acceso al sistema de ficheros

Desde Java se facilita más aún la gestión de todo esto, mediante la **clase File**, cuya utilidad es que representa un enlace a un elemento genérico del sistema de archivos.

Los **constructores principales** que posee son:

`File(File parent, String child)`	Crea el acceso a un elemento (`child`) a partir de un objeto File existente (`parent`), que representa a un directorio. El constructor está sobrecargado, y el primer elemento puede ser simplemente un `String`
`File(String pathname)`	Crea el acceso a un elemento mediante una ruta absoluta, o bien un elemento dentro de la ruta de ejecución.

Imaginemos que estamos desarrollando una aplicación de gestión de archivos para una empresa. Esta aplicación permitirá a los usuarios realizar diversas operaciones de gestión de archivos y directorios, como crear, borrar, copiar y mover archivos, así como crear y borrar directorios. Además, la aplicación necesita admitir tanto operaciones secuenciales como aleatorias en archivos de datos.
En este contexto, podríamos implementar las clases de gestión de archivos y directorios en Java para satisfacer las necesidades de la aplicación. A continuación, te mostraré cómo podríamos desarrollar estas clases en un ejemplo completo:

```
import java.io.*;

// Clase para operaciones de gestión de archivos
class FileManager {
    // Método para crear un archivo
    public void crearArchivo(String nombreArchivo) {
        File archivo = new File(nombreArchivo);
        try {
            if (archivo.createNewFile()) {
             System.out.println("Archivo creado: " + archivo.getName());
            } else {
                System.out.println("El archivo ya existe.");
            }
        } catch (IOException e) {
            System.out.println("Error al crear el archivo.");
            e.printStackTrace();
        }
    }
```

```
    // Método para borrar un archivo
    public void borrarArchivo(String nombreArchivo) {
        File archivo = new File(nombreArchivo);
        if (archivo.delete()) {
            System.out.println("Archivo borrado: " + archivo.getName());
        } else {
            System.out.println("No se pudo borrar el archivo.");
        }
    }

    // Método para copiar un archivo
    public void copiarArchivo(String origen, String destino) {
        File archivoOrigen = new File(origen);
        File archivoDestino = new File(destino);
        try (FileInputStream fis = new FileInputStream(archivoOrigen);
            FileOutputStream fos = new FileOutputStream(archivoDestino))
{
            byte[] buffer = new byte[1024];
            int length;
            while ((length = fis.read(buffer)) > 0) {
                fos.write(buffer, 0, length);
            }
            System.out.println("Archivo copiado de " + origen + " a " +
destino);
        } catch (IOException e) {
            System.out.println("Error al copiar el archivo.");
            e.printStackTrace();
        }
    }

    // Método para mover un archivo
    public void moverArchivo(String origen, String destino) {
        File archivoOrigen = new File(origen);
        File archivoDestino = new File(destino);
        if (archivoOrigen.renameTo(archivoDestino)) {
            System.out.println("Archivo movido de " + origen + " a " +
destino);
        } else {
            System.out.println("No se pudo mover el archivo.");
        }
```

```
    }
}

// Clase para operaciones de gestión de directorios
class DirectoryManager {
    // Método para crear un directorio
    public void crearDirectorio(String nombreDirectorio) {
        File directorio = new File(nombreDirectorio);
        if (directorio.mkdir()) {
                System.out.println("Directorio creado: " + directorio.
getName());
        } else {
            System.out.println("El directorio ya existe.");
        }
    }

    // Método para borrar un directorio
    public void borrarDirectorio(String nombreDirectorio) {
        File directorio = new File(nombreDirectorio);
        if (directorio.delete()) {
                System.out.println("Directorio borrado: " + directorio.
getName());
        } else {
            System.out.println("No se pudo borrar el directorio.");
        }
    }
}

// Clase para operaciones de gestión de archivos secuenciales
class SequentialFileManager {
    // Implementación
}

// Clase para operaciones de gestión de archivos aleatorios
class RandomAccessFileManager {
    // Implementación
}

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

```
        FileManager fileManager = new FileManager();
        DirectoryManager directoryManager = new DirectoryManager();

        // Crear un archivo
        fileManager.crearArchivo("documento.txt");

        // Copiar un archivo
      fileManager.copiarArchivo("documento.txt", "copia_documento.txt");

        // Mover un archivo
        fileManager.moverArchivo("documento.txt", "documentos/documento.
txt");

        // Crear un directorio
        directoryManager.crearDirectorio("archivos");

        // Borrar un directorio
        directoryManager.borrarDirectorio("archivos");
    }
}
```

En este ejemplo, hemos creado una aplicación de gestión de archivos y directorios simple. La clase **`Main`** contiene el método **`main`** donde se instancia **`FileManager`** y **`DirectoryManager`** para realizar algunas operaciones de prueba, como crear, copiar, mover archivos, y crear y borrar directorios. Este ejemplo puede ser ampliado y mejorado para satisfacer las necesidades específicas de una aplicación real de gestión de archivos.

1.3. Rutas

CLAVES Y CONSEJOS

Cuando accedemos a los sistemas de ficheros, tenemos que plantearnos en qué sistema está ejecutándose nuestro programa. En todos los sistemas operativos actuales existe la gestión en directorios y ficheros, pero debemos

tener en cuenta el **`separator`**, referido al carácter que separa un elemento contenido dentro de otro cuando especificamos las rutas de carpetas. En sistemas basados en Linux es la barra (**`Slash, /`**), mientras que en sistemas Microsoft es la contrabarra (**`Backslash, \`**).

Para hacer nuestros programas portables deberemos utilizar la constante **`File.separator`**, que nos devuelve dicho elemento.

En sistemas Microsoft, en caso de tener que escribir alguna ruta, deberemos escapar la **contrabarra**, ya que tiene un significado especial dentro de los **`Strings`**, cosa que no ocurre con la barra:

EJEMPLO

```
File f=new File("/home/alumno/texto.md") // sistemas Linux
File f=new File("C:\\Usuarios\\alumno\\Escritorio\\texto.md") //
sistemas Windows
```

Los nombres de las variables indican claramente que representan archivos en sistemas Linux y Windows respectivamente, haciendo el código más legible y autoexplicativo.

```
/ Para sistemas Linux
File archivoLinux = new File("/home/alumno/texto.md");

// Para sistemas Windows
File archivoWindows = new
File("C:\\Usuarios\\alumno\\Escritorio\\texto.md");
```

En los sistemas operativos también aparece el concepto de «separador de rutas» o **path separator**, carácter que se utiliza para separar una lista de **`path`**. Habitualmente, es el carácter «:», pero, por portabilidad, puede obtenerse mediante la constante **`File.pathSeparator`**.

IMPORTANTE

Las rutas **absolutas** están referidas respecto a la raíz del sistema operativo, mientras que las **relativas** lo hacen desde el punto de ejecución del programa, desde donde se ejecuta este.

Comprobaciones

Una vez se ha obtenido la referencia al elemento del sistema de ficheros, debemos comprobar si dicho elemento es un fichero o un directorio, y lo más importante, si existe o no. Para esas comprobaciones disponemos de los siguientes métodos de la clase **`File`**:

`boolean exists()`	Devuelve **`true`** si el objeto al que apuntamos existe.
`boolean isFile()` `boolean isDirectory()`	Devuelven **`true`** en el caso de que la referencia sea un fichero o un directorio, respectivamente. Estas funciones son excluyentes.

Para el caso de que queramos manipular elementos, disponemos de:

`boolean createNewFile()`	Crea un fichero nuevo, solo en caso de que no exista.
`boolean mkdir()` `boolean mkdirs()`	Crea un directorio nuevo, solo en caso de que no exista. Mediante **`mkdirs()`** crearemos también los directorios padre del actual, en caso de que no existan.
`boolean renameTo(File)`	Permite renombrar el elemento actual a otro.
`boolean delete()`	Elimina el elemento.

IMPORTANTE

Cuando vayamos a trabajar con el contenido de los ficheros, no necesitaremos crearlo previamente, ya que los métodos de apertura de fichero, en caso de no existir, lo crean automáticamente.

1.4. Propiedades de los elementos

Para poder consultar ciertas propiedades de los ficheros, presentamos algunos de los métodos de utilidad que posee la clase **File**.

`boolean canRead()` `boolean canWrite()` `boolean canExecute()`	Como su nombre indica, informan de los permisos de usuario para las operaciones de lectura, escritura y ejecución de cada fichero.
`long length()`	Devuelve el tamaño en **bytes** del fichero. Para los directorios, esta información no es representativa.
`String[] list()`	Devuelve un vector de **`String`** con los elementos que contiene el directorio actual.

En el siguiente **programa** se muestra el uso de cada una de las opciones anteriores.

EJEMPLO

```
import java.io.File;

public class FileInformation {

    public static void main(String[] args) {
        // Verificar si se proporcionó una ruta como argumento de línea
de comandos
        if (args.length < 1) {
              System.out.println("Por favor, proporcione una ruta como
argumento.");
            return;
        }

        // Obtener la ruta del primer argumento
        String ruta = args[0];

        // Crear un objeto File con la ruta proporcionada
        File archivo = new File(ruta);

        // Verificar si el archivo o directorio existe
        if (archivo.exists()) {
            // Verificar si es un archivo
            if (archivo.isFile()) {
                // Mostrar información sobre el archivo
                mostrarInformacionArchivo(archivo);
            } else {
                // Mostrar lista de archivos en el directorio
                mostrarArchivosDirectorio(archivo);
            }
        } else {
            System.out.println("El fichero o ruta no existe.");
        }
    }

    // Método para mostrar información sobre un archivo
    private static void mostrarInformacionArchivo(File archivo) {
        System.out.println("Información del archivo:");
        System.out.println("Nombre: " + archivo.getName());
      System.out.println("Ruta absoluta: " + archivo.getAbsolutePath());
```

```
        System.out.println("Tamaño: " + archivo.length() + " bytes");
        System.out.println("Puede ejecutarse: " + archivo.canExecute());
        System.out.println("Puede leerse: " + archivo.canRead());
        System.out.println("Puede escribirse: " + archivo.canWrite());
    }

    // Método para mostrar la lista de archivos en un directorio
    private static void mostrarArchivosDirectorio(File directorio) {
        System.out.println("Archivos en el directorio " + directorio.
getAbsolutePath() + ":");
        String[] archivos = directorio.list();
        for (String nombreArchivo : archivos) {
            System.out.println("\t" + nombreArchivo);
        }
    }
}
```

Tabla que resume la explicación del código:

PUNTO	DESCRIPCIÓN
1	Importaciones: Importamos la clase `File` del paquete `java.io` para trabajar con archivos y directorios.
2	Clase `FileInformation`: Definimos una clase llamada `FileInformation` que contiene el método `main`, punto de entrada de la aplicación.
3	Método `main`: Punto de inicio de la aplicación. Verifica si se proporciona al menos un argumento de línea de comandos, que representa la ruta del archivo o directorio a analizar.
4	Obtención de la ruta: Si se proporciona al menos un argumento, se obtiene la ruta del primer argumento y se crea un objeto `File` con esa ruta.
5	Verificación de existencia: Se verifica si el archivo o directorio especificado existe. Si existe, se determina si es un archivo o un directorio.
6	Manejo de archivos: Si el objeto `File` representa un archivo, se llama al método `mostrarInformacionArchivo` para mostrar información detallada sobre el archivo.
7	Manejo de directorios: Si el objeto `File` representa un directorio, se llama al método `mostrarArchivosDirectorio` para mostrar la lista de archivos en ese directorio.
8	Método `mostrarInformacionArchivo`: Muestra información detallada sobre un archivo, como su nombre, ruta, tamaño y permisos.

9	Método **`mostrarArchivosDirectorio`**: Muestra la lista de archivos contenidos en un directorio.
10	Tratamiento de excepciones: Manejo de posibles excepciones, como la falta de argumentos de línea de comandos o la no existencia del archivo o directorio especificado.

Veamos la lógica del ejemplo anterior, en el que comprobamos la existencia del fichero:

- En caso de no existir, se informa.
- En caso de existir, se verifica de qué tipo es:
  - Si es un fichero, se muestran algunos de sus atributos.
  - Si es un directorio, se muestran los elementos que contiene, mediante un listado simple.

Con todo esto, ya podemos movernos por la estructura de ficheros del sistema en el que estemos y empezar a acceder a los contenidos.

02 Ficheros de texto

Las clases que posee Java para la gestión de ficheros de texto son las siguientes:

CLASE	UTILIDAD
`FileWriter/FileReader`	Nos permite gestionar ficheros de texto, aunque solo mediante lecturas/ escrituras carácter a carácter o en bloques de longitud constante.
`BufferedWriter` `BufferedReader`	Creados a partir de FileWriter/FileReader, permiten el tratamiento de ficheros de texto más cercano al ser humano, ya que el tratamiento de sus datos puede realizarse línea a línea. Sigue incorporando los mecanismos de proceso carácter a carácter de la clase a partir de la cual se crea.

IMPORTANTE

Es muy importante tener en cuenta que crear un **objeto File** con el acceso a un elemento no crea un fichero (de momento), aunque nos permitirá hacerlo a partir de él.

2.1. Procesado carácter a carácter

VÍDEO LAB

Para consultar el siguiente vídeo sobre Acceso a ficheros, escanea el código QR o pulsa aquí.

Los objetos **FileWriter/FileReader** suelen ser adecuados para tratamientos generales de ficheros, pero, debido a la dificultad de saber cuánto ocupa la información ***a priori***, no se usan demasiado. Debes de fijarte en que, por ejemplo, no podemos responder a la pregunta «¿Qué longitud tienen las frases o líneas?».

Estas clases poseen como **métodos generales**:

`int read()`	El primero de ellos lee un solo carácter, aunque devuelto en un entero, por lo que necesitaremos hacer una conversión de tipo a `char` (cásting). Este método devuelve un -1 cuando no puede leerse.
`int read char[],offset,max)`	El segundo de ellos lee un bloque de caracteres de una longitud máxima determinada, a partir de una posición en el array, devolviendo la cantidad de datos leídos. Este método devuelve un `-1` cuando no puede leerse.
`void write(int c)`	Escribe el carácter pasado como argumento.
`void write (char[])`	Escribe el conjunto de caracteres pasados dentro del vector.

El siguiente ejemplo imprime por pantalla un fichero, leyendo una letra y mostrándola a continuación:

EJEMPLO

```
File f = new File(ruta);       //suponemos que existe el fichero
FileReader  fr = new FileReader(f);
int c;
while ((c = fr.read()) != -1) {
      char letra = (char) c;
      System.out.print(letra);
}
```

2.2. Procesado línea a línea

El principal inconveniente para estos casos es la ausencia del concepto de línea. Este concepto se introduce en las clases **`BufferedWriter`** y **`BufferedReader`**. Mediante estas clases, podemos leer un texto hasta encontrar el separador de la línea, que viene representado mediante la constante «**`\n`**».

IMPORTANTE

El carácter **`\n`** puede denotarse como el **LF (*line feed*)**; en ocasiones se combina también con **CR (*carriage return*)** por compatibilidad con protocolos antiguos, representando los dos movimientos que se introducían en la máquina de escribir para avanzar y volver al principio de la línea.

- Para **leer archivos línea a línea**, deberemos, como antes, crear la referencia a un fichero y crear el objeto. Leeremos posteriormente con el método **`String readLine()`**, guardando el contenido de dicha lectura en una variable **`String`**. Este método devolverá **`null`** cuando no se ha podido efectuar la lectura porque se ha llegado al final del fichero.

Explicación sobre cómo procesar archivos línea por línea y cómo manejar los caracteres de nueva línea en Java:

1. **Estructura de los archivos**: Los archivos pueden tener diferentes estructuras, como archivos CSV (Comma-Separated Values) donde los datos están separados por comas, archivos con longitudes fijas donde cada campo tiene una longitud específica, o archivos de configuración donde cada línea puede contener información de diferentes tipos.

2. **Leer archivos línea por línea**: Para leer archivos línea por línea en Java, podemos utilizar la clase **`BufferedReader`**, que proporciona el método **`readLine()`** para leer una línea completa del archivo. Este método devuelve una cadena de caracteres (**`String`**) que contiene el contenido de la línea leída. Cuando se alcanza el final del archivo, el método **`readLine()`** devuelve **`null`**.

3. **Procesamiento de archivos con estructuras específicas**:

   - **Archivos CSV**: Para procesar archivos CSV, podemos leer cada línea con `readLine()` y luego dividir la línea en campos utilizando el método `split(",")` u otras técnicas de análisis de cadenas.
   - **Archivos con longitudes fijas**: En estos archivos, cada campo ocupa una cantidad fija de caracteres. Podemos leer cada línea con `readLine()` y luego extraer cada campo utilizando métodos como `substring()`.
   - **Archivos de configuración**: En estos archivos, cada línea puede contener información de diferentes tipos. Podemos leer cada línea con `readLine()` y luego analizar su contenido según el formato esperado.

4. **Destacando el método `readLine()`**: Como mencionaste, para leer archivos línea por línea en Java, necesitamos crear una referencia al archivo y un objeto `BufferedReader`. Luego, utilizamos el método `readLine()` para leer cada línea del archivo, almacenando su contenido en una variable `String`. Cuando llegamos al final del archivo, el método `readLine()` devuelve `null`, lo que nos permite salir del bucle de lectura.

Como ejemplo:
Imaginemos que estamos desarrollando un programa Java para procesar un archivo de configuración que contiene información heterogénea en cada línea. Este archivo de configuración es utilizado por una aplicación para cargar diversas configuraciones, como opciones de usuario, ajustes de la aplicación, o datos de conexión a bases de datos.
Supongamos que el archivo de configuración tiene el siguiente formato:

```
# Archivo de configuración
usuario=admin
contraseña=secreta
servidor=localhost
puerto=3306
```

Cada línea del archivo contiene un par clave-valor separado por el signo igual (=). Estas líneas pueden representar diferentes tipos de información, como cadenas de texto, números o valores booleanos.
Aquí te muestro cómo podríamos desarrollar un código en Java para leer este archivo línea por línea, procesarlo y cargar la configuración en nuestra aplicación:

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

public class ConfiguracionApp {

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

        // Intentamos abrir el archivo y procesarlo
                try (BufferedReader lector = new BufferedReader(new
FileReader(archivoConfiguracion))) {
            String linea;
            // Iteramos sobre cada línea del archivo
            while ((linea = lector.readLine()) != null) {
                // Procesamos la línea
                procesarLineaConfiguracion(linea);
            }
        } catch (IOException e) {
           System.err.println("Error al leer el archivo de configuración:
" + e.getMessage());
        }
    }

    // Método para procesar cada línea del archivo de configuración
    private static void procesarLineaConfiguracion(String linea) {
        // Dividimos la línea en la clave y el valor, separados por el
signo igual
        String[] partes = linea.split("=");
        if (partes.length == 2) {
            String clave = partes[0].trim();
            String valor = partes[1].trim();
            // Realizamos acciones según la clave y el valor
            switch (clave) {
                case "usuario":
                    System.out.println("Usuario: " + valor);
                    break;
                case "contraseña":
                    System.out.println("Contraseña: " + valor);
                    break;
```

```
                case "servidor":
                    System.out.println("Servidor: " + valor);
                    break;
                case "puerto":
                    System.out.println("Puerto: " + valor);
                    break;
                default:
                    System.out.println("Clave desconocida: " + clave);
            }
        } else {
            // La línea no tiene el formato esperado
              System.out.println("Línea de configuración no válida: " +
linea);
        }
    }
}
```

En este código, leemos el archivo de configuración línea por línea utilizando un objeto **`BufferedReader`**. Para cada línea leída, la procesamos utilizando el método **`procesarLineaConfiguracion`**, donde dividimos la línea en la clave y el valor utilizando el método **`split("=")`**. Luego, realizamos acciones según la clave y el valor obtenidos, como mostrarlos por pantalla o realizar configuraciones en la aplicación.

IMPORTANTE

Hay que tener en cuenta que, en los ficheros de texto, entre línea y línea se almacena el carácter «**`\n`**». Al leer mediante líneas, dicho «**`\n`**» desaparece; es decir, leemos las líneas, y los «**`\n`**», como son los separadores, se pierden. Si queremos mantenerlos en ficheros de salida, deberemos añadirlos de manera explícita.

- Para la **escritura por bloques o líneas**, necesitaremos el objeto de tipo **`BufferedWriter`**. El método que permite la escritura es **`void write(String s)`**, que envía al fichero el texto pasado. Ten en cuenta que este método no añade el **`\n`**, por lo que debemos solucionarlo con una de estas dos opciones:
  - **String texto="Esto es una línea"; fw.write(texto); fw.newLine();**
  - **String texto="Esto es una línea"; fw.write(texto+"\n");**

2.3. Archivos CSV

DEFINICIÓN

Los archivos **CSV (*comma separated values*)** son ficheros de texto plano que contienen información respecto a una entidad. Se estructuran en líneas, donde cada línea representa a una ocurrencia de una entidad. Cada una de estas líneas contiene una serie de valores separados por comas, que especifican las propiedades o características de cada ocurrencia.

EJEMPLO

Sea, por ejemplo, el extracto de un archivo **alumnos.csv**, donde se indican para una serie de alumnos su nombre, su edad, el CFGS que cursa y su nota de acceso.

```
Angela Veron,20,DAW,7.1223
Silvia Climent,24,DAM,5.5
Jordi Costa,22,DAW,9
```

Deberemos, en este punto, fijarnos en los siguientes detalles:

- Las líneas no tienen la misma longitud, pero eso no supondrá ningún problema porque vienen delimitadas por el **`\n`**.
- Deben existir todos los campos; si hay alguno vacío, deben aparecer las comas que lo delimitan.
- Si, por algún motivo, el carácter coma (`,`) puede aparecer dentro del texto (como, por ejemplo, «Pérez Angulo, José Ramón»), entonces el fichero CSV debe utilizar como separador otro carácter, como el «`;`» o el «`#`», o incluso separarlo por el carácter especial tabulación (**`\t`**).

Como resumen, el algoritmo para procesar dicho fichero deberá:

1. Abrir el fichero.
2. Recorrer hasta el final.
3. Leer cada línea y procesarla.
4. Calcular la información de salida.
5. Generar el fichero de salida, si procede.

Estamos trabajando en un proyecto que tiene como objetivo la gestión de datos de alumnos para contribuir al logro del Objetivo de Desarrollo Sostenible (ODS) número 4 de la Agenda 2030 de las Naciones Unidas, que busca garantizar una educación inclusiva, equitativa y de calidad para todos, promoviendo oportunidades de aprendizaje durante toda la vida. En este contexto, nuestro sistema necesita manejar información sobre los alumnos, como sus nombres, edades, programas de estudios y calificaciones.
Desarrollo de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para archivos CSV:

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

public class AlumnoCSVManager {

    private static final String ARCHIVO_ALUMNOS = "alumnos.csv";

    public static void main(String[] args) {
        // Crear algunos alumnos de ejemplo
        List<Alumno> alumnos = new ArrayList<>();
        alumnos.add(new Alumno("Angela Veron", 20, "DAW", 7.1223));
        alumnos.add(new Alumno("Silvia Climent", 24, "DAM", 5.5));
        alumnos.add(new Alumno("Jordi Costa", 22, "DAW", 9));
```

```
        // Guardar los alumnos en el archivo CSV
        guardarAlumnosCSV(alumnos);

        // Leer y mostrar los alumnos del archivo CSV
        List<Alumno> alumnosLeidos = leerAlumnosCSV();
        for (Alumno alumno : alumnosLeidos) {
            System.out.println(alumno);
        }
    }

    // Método para guardar la lista de alumnos en un archivo CSV
    private static void guardarAlumnosCSV(List<Alumno> alumnos) {
      try (PrintWriter escritor = new PrintWriter(new FileWriter(ARCHIVO_
ALUMNOS))) {
            for (Alumno alumno : alumnos) {
                escritor.println(alumno.toCSV());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // Método para leer los alumnos desde un archivo CSV
    private static List<Alumno> leerAlumnosCSV() {
        List<Alumno> alumnos = new ArrayList<>();
   try (BufferedReader lector = new BufferedReader(new FileReader(ARCHIVO_
ALUMNOS))) {
            String linea;
            while ((linea = lector.readLine()) != null) {
                String[] partes = linea.split(",");
                String nombre = partes[0];
                int edad = Integer.parseInt(partes[1]);
                String programa = partes[2];
                double nota = Double.parseDouble(partes[3]);
                alumnos.add(new Alumno(nombre, edad, programa, nota));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return alumnos;
    }
```

```
    // Clase para representar a un alumno
    private static class Alumno {
        private String nombre;
        private int edad;
        private String programa;
        private double nota;

        public Alumno(String nombre, int edad, String programa, double
nota) {
            this.nombre = nombre;
            this.edad = edad;
            this.programa = programa;
            this.nota = nota;
        }

        public String toCSV() {
                return String.join(",", nombre, String.valueOf(edad),
programa, String.valueOf(nota));
        }

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

Este código muestra cómo realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en un archivo CSV que contiene información sobre alumnos. Primero, creamos algunos objetos de tipo **`Alumno`** como ejemplos y los guardamos en un archivo CSV utilizando el método **`guardarAlumnosCSV()`**. Luego, leemos los alumnos desde el archivo CSV utilizando el método **`leerAlumnosCSV()`** y los mostramos por consola.

Explicación código:

PUNTO	DESCRIPCIÓN
1	Importaciones: Importamos las clases necesarias para trabajar con archivos y listas.
2	Definición de la clase **`AlumnoCSVManager`**: Creamos una clase para manejar la gestión de alumnos en archivos CSV.
3	Constante **`ARCHIVO_ALUMNOS`**: Definimos una constante que representa el nombre del archivo CSV.
4	Método **`main()`**: Punto de entrada de la aplicación.
5	Creación de alumnos de ejemplo: Creamos una lista de objetos **`Alumno`** con datos de ejemplo.
6	Guardar alumnos en el archivo CSV: Llamamos al método **`guardarAlumnosCSV()`** para escribir los datos de los alumnos en el archivo CSV.
7	Leer y mostrar alumnos del archivo CSV: Llamamos al método **`leerAlumnosCSV()`** para obtener y mostrar los alumnos del archivo CSV.
8	Método **`guardarAlumnosCSV()`**: Guarda la lista de alumnos en el archivo CSV.
9	Método **`leerAlumnosCSV()`**: Lee los alumnos desde el archivo CSV y los devuelve como una lista.
10	Clase interna **`Alumno`**: Representa un alumno y proporciona métodos para convertir el objeto en una cadena CSV y para imprimir sus datos.

03 Ficheros binarios. Ficheros de objetos

Existen infinidad de datos que necesitan guardarse en formato binario. Esto tiene algunas ventajas:

- Es una medida de protección, ya que los ficheros de texto, son visibles desde cualquier entorno.
- Las operaciones de lectura y escritura son secuenciales, es decir, debemos recorrer todos los caracteres antes de llegar a uno deseado. Esta limitación se elimina con los archivos binarios.

CLAVES Y CONSEJOS

Por tanto, estamos ante la posibilidad de guardar datos de cualquier naturaleza, pero podemos ir mucho más allá. Los objetos tienen una constitución estructurada y están formados por atributos, y en última instancia son estos los que deberán ser guardados.

IMPORTANTE

Imaginemos que poseemos una clase denominada Alumno, que posee un nombre, una edad y un peso, de tipos texto, entero y real. Para guardar un objeto de tipo alumno, podemos ir accediendo mediante los métodos correspondientes (`getters`) a los distintos atributos y guardándolos uno a uno en los ficheros. Como veremos, existen formas de realizar el guardado automático de todos los atributos (el objeto entero) mediante un mecanismo denominado Serialización.

3.1. Clases Java para los ficheros binarios

Anteriormente, exploramos algunas clases de la biblioteca de Java diseñadas para el manejo de archivos de texto. No obstante, es importante señalar que **estas bibliotecas no son adecuadas para el procesamiento de ciertos formatos de archivo.** Para abordar esta necesidad específica, Java proporciona las siguientes clases:

`FileInputStream` **`FileOutputStream`**	Son clases generales de acceso a ficheros binarios. Estas clases se crean, al igual que los ficheros de texto, de un objeto **File**. Permiten leer o escribir bytes ad hoc mediante sus métodos genéricos **`read()`** y **`write()`**, por lo que no son del todo intuitivos. Para leer de un fichero hasta llegar a su fin, necesitaremos evaluar lo que devuelve **`read`**, ya que la constante «-1» indica que hemos llegado al final del fichero.
`DataInputStream` **`DataOutputStream`**	Son clases generadas a partir de las correspondientes anteriores. La gran ventaja es la incorporación de los métodos **`readTIPO()`** y **`writeTIPO()`**, siendo tipo cualquiera de los tipos básicos. El flujo de la información viaja en un solo sentido: o leemos o escribimos. Ejemplos de estas funciones son **`readInt()`**, **`readDouble()`**, **`readUTF()`** y las correspondientes de **`writeInt()`**, **`writeUTF()`**.
`RandomAccessFile`	Esta clase engloba las dos anteriores, ya que permite tanto leer como escribir los tipos adecuados. Además, tiene métodos de posicionamiento en una determinada posición mediante la función **`seek(int pos)`**. También la función **`skipBytes(int cantidad)`** permite avanzar (saltarse) la cantidad determinada de bytes.
`ObjectInputStream` **`ObjectOutputStream`**	Esta clase permite, además de los métodos de **`DataInputStream`** y **`DataOutputStream`**, leer y escribir objetos directamente. Se requiere la serialización de los objectos, implementando la interfaz **`Serializable`**.

Ejemplo con `FileInputStream` y `FileOutputStream`:

```
import java.io.*;

public class EjemploFicherosBinarios {

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

        // Escribir datos en el archivo binario
        try (FileOutputStream fos = new FileOutputStream(nombreArchivo);
             DataOutputStream dos = new DataOutputStream(fos)) {
            dos.writeInt(42);
            dos.writeDouble(3.1416);
            dos.writeUTF("Hola, mundo!");
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        // Leer datos del archivo binario
        try (FileInputStream fis = new FileInputStream(nombreArchivo);
             DataInputStream dis = new DataInputStream(fis)) {
            int entero = dis.readInt();
            double decimal = dis.readDouble();
            String texto = dis.readUTF();
            System.out.println("Entero: " + entero);
            System.out.println("Decimal: " + decimal);
            System.out.println("Texto: " + texto);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo con `RandomAccessFile`:

```
import java.io.*;

public class EjemploRandomAccessFile {

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

        try (RandomAccessFile raf = new RandomAccessFile(nombreArchivo,
"rw")) {
            // Escribir datos en el archivo
            raf.writeInt(42);
            raf.writeDouble(3.1416);
            raf.writeUTF("Hola, mundo!");

            // Posicionar el puntero al inicio del archivo
            raf.seek(0);

            // Leer datos del archivo
            int entero = raf.readInt();
            double decimal = raf.readDouble();
            String texto = raf.readUTF();
            System.out.println("Entero: " + entero);
            System.out.println("Decimal: " + decimal);
            System.out.println("Texto: " + texto);
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo con ObjectInputStream y ObjectOutputStream:

```
import java.io.*;

public class EjemploObjectStream {

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

        // Escribir objetos en el archivo binario
        try (FileOutputStream fos = new FileOutputStream(nombreArchivo);
             ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(new Persona("Juan", 25));
            oos.writeObject(new Persona("María", 30));
        } catch (IOException e) {
            e.printStackTrace();
        }

        // Leer objetos del archivo binario
        try (FileInputStream fis = new FileInputStream(nombreArchivo);
             ObjectInputStream ois = new ObjectInputStream(fis)) {
            Object obj1 = ois.readObject();
            Object obj2 = ois.readObject();
            if (obj1 instanceof Persona && obj2 instanceof Persona) {
                Persona persona1 = (Persona) obj1;
                Persona persona2 = (Persona) obj2;
                System.out.println("Persona 1: " + persona1);
                System.out.println("Persona 2: " + persona2);
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
```

```
    // Clase para representar una persona
    static class Persona implements Serializable {
        private String nombre;
        private int edad;

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

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

ObjectOutputStream.writeObject(Object obj): Este método es utilizado para escribir un objeto en un flujo de salida de objetos. Serializa el objeto proporcionado y escribe los bytes resultantes en el flujo de salida.

- **ObjectInputStream.readObject()**: Este método es utilizado para leer un objeto del flujo de entrada de objetos. Lee los bytes del flujo de entrada, los deserializa y devuelve el objeto resultante.
- **FileOutputStream(String name)**: Constructor de la clase **FileOutputStream**. Crea un flujo de salida de bytes hacia un archivo con el nombre especificado.
- **ObjectOutputStream(OutputStream out)**: Constructor de la clase **ObjectOutputStream**. Crea un flujo de salida de bytes para escribir objetos en el flujo de salida proporcionado.
- **FileInputStream(String name)**: Constructor de la clase **FileInputStream**. Crea un flujo de entrada de bytes desde un archivo con el nombre especificado.
- **ObjectInputStream(InputStream in)**: Constructor de la clase **ObjectInputStream**. Crea un flujo de entrada de bytes para leer objetos del flujo de entrada proporcionado.

- **`Serializable`**: Interfaz de marcador que se utiliza para indicar que una clase puede ser serializable. Las clases que implementan esta interfaz pueden ser convertidas en una secuencia de bytes y viceversa, lo que permite su almacenamiento en archivos o su transmisión a través de la red.

Guardar texto en el formato binario

Respecto a la **escritura de cadenas de texto** (y su guardado en ficheros binarios) debemos tener en cuenta lo siguiente:

- **`writeString(String texto)`** → almacena el **`String`** en el fichero.
- **`writeUTF(String texto)`** → almacena el **`String`** en el fichero, pero, al ser en formato UTF, guarda en dos bytes que preceden al texto la longitud de la cadena.

¿Para qué sirve esto? Para saber la longitud del texto, en previsión de saber la cantidad de texto cuando hagamos la operación de lectura.

EJEMPLO

Escribimos en un fichero el número 10 (4 bytes), el texto HOLA (4 bytes) y el numero 20 (4 bytes).

```
writeInt(10);
writeString("Hola");
writeInt(20);
```

En el fichero encontraremos:

0	1	2	3	4	5	6	7	8	9	10	11
10				H	O	L	A	20			

Al leer, primero leemos en **`int`** sin complicaciones (4 bytes), pero el problema es que no podemos leer el texto, ya que no hay ninguna marca que indique dónde termina el texto. La solución, mediante **`writeUTF()`**, queda como sigue:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
10				4		H	O	L	A	20			

Como podemos observar, ahora el primer **`readInt()`** leerá el 10, y el siguiente **`readUTF()`** procesa dos bytes, para leer el 4, y entonces sabe que tiene que leer el texto HOLA (los cuatro siguientes bytes), pudiendo así leer el siguiente número que queda al final.

3.2. Flujos de datos

DEFINICIÓN

Entendemos por **flujos de datos** los transvases de datos que se realizan entre un origen y un destino. Si la información que viaja no precisa ser interpretada, el mecanismo adecuado para enviar y recibir esos flujos de datos son los `FileInputStream` y `FileOutputStream`. Habitualmente, en estos flujos **(streams)** la información se envía sin interpretar como flujos de bits.

3.3. Decoradores

En caso de querer **aplicar una transformación** a dichos flujos de datos, necesitamos una capa de «maquillaje» que nos transforme dichos **bits a datos.** Esas clases decoradoras son las que se crean a partir de las anteriores y que contiene los métodos específicos.

A. Escritura/lectura de objetos

Mediante las últimas clases decoradas, `ObjectInputStream` y `ObjectOutputStream`, podemos leer y escribir objetos directamente. Los métodos que añaden para la realización de su cometido son:

`writeObject (Object o)`

Método de `ObjectOutputStream`
Permite la escritura de un objeto cualquiera dentro de la jerarquía de objetos de Java. Para posibilitar esto, los objetos que queramos guardar deben implementar la interfaz **Serializable**. La serialización es el hecho de convertir un objeto en un flujo de bits; en nuestro caso, para guardar dichos bits en un fichero. También la necesitaremos para enviarlos por sockets de comunicaciones o por cualquier **stream**.
Java guardará en la cabecera de los ficheros una metainformación extraída de la clase de los objetos que contiene (básicamente, los tipos de sus atributos).

`Object readObject()`

Método de `ObjectInputStream`
Realiza el proceso de «deserialización» para convertir los bits del fichero en los objetos del programa. Java conoce la descripción del objeto que se va a guardar, ya que se ha guardado en la cabecera del fichero.
Como el resultado de la carga del objeto es un `Object`, necesitaremos la realización de una conversión (cásting) al tipo de objeto adecuado. Si el objeto que cargamos no puede transformarse en el que indicamos en el cásting, aparecerá la excepción `ClassNotFoundException`.

EJEMPLO

Supongamos el siguiente ejemplo:

```
public class ClaseA {
   private String texto;
}
public class ClaseB implements Serializable {
  private ClaseA elA;
  private int num;
}
```

Si guardamos un objetos del tipo B, nos encontraremos la sorpresa de que aparece una **java.io.NotSerializableException**; esto es debido a que, aunque B está serializado, al intentar guardar el objeto de tipo B, como contiene a un objeto de tipo A, este no está serializado, lo que provoca el error anterior.

B. Escritura/lectura de colecciones

En la mayoría de los programas, no será habitual disponer de un solo objeto, sino de colecciones de estos. Por ello, podemos proceder de varias maneras para su proceso. Supongamos que tenemos una colección de n objetos, y queremos guardarlos en un fichero. Se nos presentan dos posibilidades:

Repetir n veces (1 por elemento) **Guardar 1 elemento** **For(Alumno a: losAlumnos)** **writeObject(a)**	**Guardar la colección** **writeObject(losAlumnos)**
Generamos un fichero con n objetos, de manera individual.	Generamos un archivo con un solo objeto, que contiene una colección de objetos individuales.

Los dos ejemplos previos son simplemente opciones, ninguna es superior o inferior a la otra. La elección entre ambas dependerá de los requisitos específicos del programa, permitiéndonos adoptar una u otra en función de las necesidades particulares.

Supongamos que estamos desarrollando una aplicación de gestión académica para una institución educativa. En esta aplicación, necesitamos manejar una colección de objetos **Alumno**, donde cada objeto representa a un estudiante con sus respectivos datos, como nombre, edad, programa académico, y calificaciones.

Desarrollo de ejemplo CRUD (Crear, Leer, Actualizar, Eliminar) para escritura y lectura de colecciones de objetos:

EJEMPLO

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

public class GestionAcademica {

    private static final String ARCHIVO_ALUMNOS = "alumnos.dat";

    public static void main(String[] args) {
        // Creamos algunos alumnos de ejemplo
        List<Alumno> listaAlumnos = new ArrayList<>();
        listaAlumnos.add(new Alumno("Juan", 20, "Ingeniería", 8.5));
        listaAlumnos.add(new Alumno("María", 22, "Medicina", 9.2));

        // Guardamos la lista de alumnos en un archivo
        guardarAlumnos(listaAlumnos);

        // Leemos la lista de alumnos desde el archivo y la mostramos
        List<Alumno> listaAlumnosLeidos = leerAlumnos();
        for (Alumno alumno : listaAlumnosLeidos) {
            System.out.println(alumno);
        }
    }

    // Método para guardar una lista de alumnos en un archivo
    private static void guardarAlumnos(List<Alumno> alumnos) {
            try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(ARCHIVO_ALUMNOS))) {
            // Escribimos la lista de alumnos en el archivo
            oos.writeObject(alumnos);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // Método para leer una lista de alumnos desde un archivo
    private static List<Alumno> leerAlumnos() {
        List<Alumno> listaAlumnos = new ArrayList<>();
             try (ObjectInputStream ois = new ObjectInputStream(new
```

```
FileInputStream(ARCHIVO_ALUMNOS))) {
            // Leemos la lista de alumnos desde el archivo
            listaAlumnos = (List<Alumno>) ois.readObject();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        return listaAlumnos;
    }

    // Clase para representar a un alumno
    private static class Alumno implements Serializable {
        private String nombre;
        private int edad;
        private String programaAcademico;
        private double calificacionPromedio;

       public Alumno(String nombre, int edad, String programaAcademico,
double calificacionPromedio) {
            this.nombre = nombre;
            this.edad = edad;
            this.programaAcademico = programaAcademico;
            this.calificacionPromedio = calificacionPromedio;
        }

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

Tabla que explica el código en pasos:

PUNTO	DESCRIPCIÓN
1	Definición de la clase **`GestionAcademica`**: Se crea una clase principal para gestionar la aplicación de gestión académica.
2	Declaración de la constante **`ARCHIVO_ALUMNOS`**: Se define una constante que representa el nombre del archivo donde se guardarán los datos de los alumnos (**`alumnos.dat`**).
3	Método **`main()`**: Punto de entrada de la aplicación.
4	Creación de la lista de alumnos de ejemplo: Se crea una lista de objetos **`Alumno`** con datos de ejemplo.
5	Llamada al método **`guardarAlumnos()`**: Se llama al método para guardar la lista de alumnos en un archivo.
6	Llamada al método **`leerAlumnos()`**: Se llama al método para leer la lista de alumnos desde el archivo.
7	Iteración y visualización de la lista de alumnos leída: Se itera sobre la lista de alumnos leída y se muestra cada alumno por consola.
8	Método **`guardarAlumnos()`**: Método para guardar una lista de alumnos en un archivo.
9	Método **`leerAlumnos()`**: Método para leer una lista de alumnos desde un archivo.
10	Clase **`Alumno`**: Clase para representar a un alumno.
11	Implementación de la interfaz **`Serializable`**: Se implementa la interfaz **`Serializable`** en la clase **`Alumno`** para permitir la serialización de objetos.

04 Almacenamiento con documentos XML

Cuando queremos guardar datos que puedan ser leídos por diferentes aplicaciones y plataformas, lo más adecuado es hacer uso de formatos estándares de almacenamiento, que múltiples aplicaciones puedan entender. Un caso muy concreto son los lenguajes de marcas, y el más conocido es el estándar **XML (*extensible markup language*)**.

Estructuración de información con XML

En documentos XML, organizamos la información mediante marcas o hashtags:

- **Características de las marcas:** Las marcas tienen un principio y un final definidos. Pueden anidarse y contener información textual.
- **Formato textual de la información:** La información en XML es siempre textual. Se evita la problemática de representación diversa de datos.
- **Universalidad de representación:** Cualquier tipo de dato se convierte a texto. Asegura uniformidad en la representación de datos.
- **Gestión de codificación de texto:** XML aborda la variabilidad en sistemas de codificación de texto. Permite especificar la codificación utilizada en la cabecera del documento.

Conclusiones:

XML ofrece una estructura jerárquica para organizar información. La transformación de datos a texto garantiza consistencia. La inclusión de información de codificación en la cabecera aborda desafíos de sistemas diversos.

¿SABÍAS QUE...?

La manera de almacenar la información en XML de forma jerárquica se asemeja mucho a la manera en que lo hacen los objetos en una aplicación, de modo que estos pueden traducirse de una forma relativamente cómoda a un documento XML.

4.1. Analizadores XML

DEFINICIÓN

Un ***parser* o analizador XML** es una clase que permite analizar un fichero XML y extraer la información de él, relacionándola según su posición en la jerarquía.

Los analizadores, según su forma de funcionar, pueden ser:

Analizadores secuenciales o sintácticos

Van extrayendo el contenido según se van descubriendo los ***hashtags*** de apertura y cierre. Son muy rápidos, pero tienen el problema de que hay que leer todo el documento para acceder a una parte concreta. En Java existe el analizador **SAX (*Simple API for XML*)** como analizador secuencial.

Analizadores jerárquicos

Suelen ser los más utilizados, ya que guardan todos los datos del documento XML en memoria, en forma de estructura jerarquizada **(DOM o modelo de objetos del documento)**; son los preferidos para aplicaciones que tengan que leer los datos de forma más continua.

4.2. El modelo de objetos del documento (DOM)

DEFINICIÓN

El **DOM (*document object model*)** es la estructura especificada por el W3C donde se almacena la información de los documentos XML.

- El **DOM** ha sido ligado sobre todo al mundo web, con HTML y **Javascript** como principales impulsores.
- En **Java**, el **DOM** se implementa haciendo uso de interfaces.
- La interfaz principal del DOM en Java es **Document**, y representa todo el documento XML. Como se trata de una interfaz, esta podrá implementarse en varias clases.

Junto con la clase Document, el **W3C** introduce la clase abstracta **DocumentBuilder,** la cual posibilita la creación del DOM a partir de XML. Adicionalmente, se presenta la clase **DocumentBuilderFactory,** encargada de fabricar instancias de **DocumentBuilder,** dado que, al ser abstracta, no permite una instanciación directa.

La manera de **recuperar el DOM de un XML** sería:

```
public Document ObreXML(String nombreArchivoXML) throws
IOException,SAXException, ParserConfigurationException,
FileNotFoundException {
    // Creamos una instancia de DocumentBuilderFactory
    DocumentBuilderFactory dbFactory =
DocumentBuilderFactory.newInstance();
    // Con la instancia de DocumentBuilderFactory creamos un
DocumentBuilder
    DocumentBuilder dBuilder = dbFactory.
newDocumentBuilder();
    //Con el método "parse" de DocumentBuilder obtenemos el
documento
    Document doc = dBuilder.parse(new
File(nombreArchivoXML));
    return doc;
}
```

Descripción de las excepciones en el código proporcionado:

EXCEPCIÓN	DESCRIPCIÓN
IOException	Esta excepción se lanza si ocurre un error de entrada o salida al intentar acceder al archivo XML especificado. Puede ser causada por problemas como permisos insuficientes, problemas de red o problemas de hardware.
SAXException	Esta excepción se lanza si ocurre un error durante el análisis del documento XML utilizando la API SAX (Simple API for XML). Esto puede ocurrir debido a un documento XML mal formado o a problemas durante el análisis.
ParserConfigurationException	Esta excepción se lanza si ocurre un error de configuración del analizador XML. Por ejemplo, puede lanzarse si el analizador no puede ser configurado según lo especificado.
FileNotFoundException	Esta excepción se lanza si el archivo XML especificado no puede ser encontrado en el sistema de archivos. Esto puede suceder si la ruta del archivo Es incorrecta o si el archivo no existe en el lugar especificado.

Estas excepciones están relacionadas con diferentes aspectos de la lectura y análisis del archivo XML. Cada una de ellas indica un posible problema que puede ocurrir durante la ejecución del método `ObreXML()`. Al declarar estas excepciones en la firma del método, se indica que el método puede arrojar alguna de estas excepciones y que quien lo llame debe manejarlas adecuadamente.

Utilidad de DocumentBuilder en creación y almacenamiento de DOM

- **Creación de DOM nuevo:**

  `DocumentBuilder` facilita la creación de un nuevo DOM con `newDocument()`.

- **Preparación para almacenamiento:**

  El DOM recién creado se utilizará para almacenar documentos XML.

- **Proceso futuro:**

  En secciones siguientes, exploraremos la adición de elementos y el almacenamiento de documentos XML. Por ahora, nos concentraremos en la interpretación y lectura del DOM.

4.3. Clases y métodos del DOM

Hasta ahora hemos visto como abrir y **parsear** un documento XML. Veamos cómo acceder a su contenido. El DOM tiene una estructura jerárquica, formada por nodos.
Los diferentes tipos de nodos que nos podemos encontrar son:

- `Document`, que es el nodo principal y representa todo el XML.
- `Element`, que representa los diferentes ***hashtags*** (incluida la raíz).
- `TextElement`, que representa el contenido de un ***hashtag*** de texto.
- `Attribute`, que representa los atributos.

IMPORTANTE

Todas estas interfaces derivan de la interfaz `Node`, por la que heredarán sus atributos y métodos y, además, aportarán atributos y métodos propios.

A. Lectura de documentos XML

Supongamos que disponemos del siguiente XML (recortado):
XML modificado con atributos:<curso>

```
    <modulo id="1">
        <nombre>Acceso a Datos</nombre>
        <horas>6</horas>
        <nota>8.45</nota>
    </modulo>
    <modulo id="2">
        <nombre>Programación de servicios y procesos</nombre>
        <horas>3</horas>
        <nota>9.0</nota>
    </modulo>
</curso>
```

Enfoque orientado a objetos al encapsular la lógica en métodos y utilizando la POO.

Nombres y zona declarativa: Los nombres de las variables están explícitos y la zona declarativa está claramente definida al principio del método **`procesarNodosModulo`**.

- **Inicialización**: El NodeList **`modulos`** se inicializa obteniendo los nodos de módulo de la raíz del documento XML.
- **POO y métodos**: Se utiliza un método **`obtenerValorEtiqueta`** para encapsular la lógica de obtención del valor de una etiqueta de un elemento.
- **Control de excepciones**: Se captura cualquier excepción que pueda ocurrir durante el procesamiento de los nodos y se imprime su traza en caso de producirse un error.
- **Acceso a un atributo**: Aunque no se ha implementado en este ejemplo, se proporciona una descripción de cómo se puede acceder a un atributo de un nodo utilizando el método **`getAttribute("nombreAtributo")`**.

```
import org.w3c.dom.*;

public class ProcesarModulos {
```

```
    // Método para procesar los nodos de módulo
    public void procesarNodosModulo(Node raiz) {
        try {
            // 1. Obtención de una lista de nodos 'modulo'
            NodeList modulos = raiz.getElementsByTagName("modulo");

            // Para cada nodo 'modulo', lo recorremos
            for (int i = 0; i < modulos.getLength(); i++) {
                Element moduloElement = (Element) modulos.item(i);

                // 2. Muestra el nombre del nodo
                System.out.println("Módulo " + (i + 1));

                // 3. Mostramos los valores de las etiquetas
                String nombre = obtenerValorEtiqueta(moduloElement,
                "nombre");
                inthoras=Integer.parseInt(obtenerValorEtiqueta(moduloElement,
                "horas"));
                doublenota=Double.parseDouble(obtenerValorEtiqueta(modulo
                Element, "nota"));

                System.out.println("Nombre: " + nombre);
                System.out.println("Horas: " + horas);
                System.out.println("Nota: " + nota);
                System.out.println();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Método para obtener el valor de una etiqueta de un elemento
    private String obtenerValorEtiqueta(Element elemento, String etiqueta)
    {
        return elemento.getElementsByTagName(etiqueta).item(0).
        getTextContent();
    }
}
```

Explicación código:

1. **Método `procesarNodosModulo`:**
   - Este método es responsable de procesar los nodos de tipo “modulo” en el documento XML.
   - Recibe un nodo raíz como parámetro, suponiendo que este nodo raíz representa el documento XML completo.
   - Dentro del método, se obtiene una lista de nodos de tipo “modulo” utilizando el método **`getElementsByTagName("modulo")`** del nodo raíz.

   Luego, itera sobre estos nodos de “modulo” utilizando un bucle **`for`**.

2. **Bucle de Iteración**:
   - Para cada nodo de “modulo” en la lista, el bucle realiza lo siguiente:
     - Convierte el nodo actual a un objeto **`Element`**.
     - Muestra el nombre del módulo mediante el número de iteración del bucle.
     - Llama al método **`obtenerValorEtiqueta`** para obtener y mostrar los valores de las etiquetas “nombre”, “horas” y “nota”.
     - Estos valores se obtienen utilizando el método **`getTextContent()`** de la etiqueta correspondiente dentro del nodo “modulo”.

3. **Método `obtenerValorEtiqueta`:**
   - Este método toma un elemento (nodo) y el nombre de una etiqueta como parámetros.
   - Utiliza el método **`getElementsByTagName(etiqueta)`** del elemento para obtener una lista de nodos con la etiqueta especificada.
   - Luego, toma el primer elemento de esta lista (asumiendo que solo hay uno) y obtiene su contenido de texto utilizando el método **`getTextContent()`**.
   - Retorna el contenido de texto de la etiqueta.

4. **Manejo de Excepciones**:
   - Se utiliza un bloque **`try-catch`** para capturar y manejar cualquier excepción que pueda ocurrir durante el procesamiento de los nodos.
   - Si se produce una excepción, se imprime la traza de la excepción utilizando el método **`printStackTrace()`**.

B. Escritura de documentos XML

Para referenciar operaciones de actualización de datos en un fichero XML. Por ejemplo, añadimos valores a los atributos o añadimos nuevos elementos a la estructura existente del documento XML.
Como sigue a continuación:
En código que vamos a incorporar se centra en la creación de un nuevo documento XML y su escritura en un archivo. Aquí está el desglose de lo que hace cada parte del código:

1. Se importan las clases necesarias de Java para trabajar con XML.
2. Se crea la clase **`ActualizarXML`**.
3. En el método **`main`**, se realiza lo siguiente: a. Se inicia un bloque **`try-catch`** para manejar cualquier excepción que pueda ocurrir durante el procesamiento. b. Se crea un nuevo documento XML utilizando **`DocumentBuilderFactory`**, **`DocumentBuilder`** y **`Document`**. c. Se crea el elemento raíz del documento (**`curso`**) y se le añaden atributos (**`nivel`** y **`ciclo`**). d. Se crea un nuevo módulo dentro del elemento raíz (**`curso`**) y se le añade un elemento de nombre (**`nombre`**) con su respectivo texto y atributo (**`curso`**). e. Se transforma y escribe el documento XML en un archivo utilizando **`TransformerFactory`**, **`Transformer`**, **`DOMSource`** y **`StreamResult`**. f. Se imprime un mensaje indicando que los datos han sido actualizados en el archivo XML. g. Se manejan las excepciones si ocurren durante el proceso.

```
import java.io.FileOutputStream;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class ActualizarXML {

    public static void main(String[] args) {
        try {
            // Parseamos el archivo XML existente
            String nombreFichero = "curso"; // Nombre del archivo XML
```

```
            DocumentBuilderFactory dbFactory = DocumentBuilderFactory.
newInstance();
            DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
            Document doc = dBuilder.newDocument(); // Creamos un nuevo
documento

            // Creamos la raíz del documento y añadimos atributos
            Element root = doc.createElement("curso");
            root.setAttribute("nivel", "2");
            root.setAttribute("ciclo", "DAM");
            doc.appendChild(root);

            // Creamos un nuevo módulo y lo añadimos al curso
            Element modulo = doc.createElement("modulo");
            root.appendChild(modulo);

            // Añadimos las características del módulo
            Element nombreModulo = doc.createElement("nombre");
           nombreModulo.appendChild(doc.createTextNode("Programación de
servicios y procesos"));
            nombreModulo.setAttribute("curso", "2");
            modulo.appendChild(nombreModulo);

            // Transformamos y escribimos los cambios en el archivo XML
            TransformerFactory transformerFactory = TransformerFactory.
newInstance();
          Transformer transformer = transformerFactory.newTransformer();
            DOMSource source = new DOMSource(doc);
                        StreamResult result = new StreamResult(new
FileOutputStream(nombreFichero + ".xml"));
            transformer.transform(source, result);

           System.out.println("Datos actualizados en el archivo XML.");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

PASO	EXPLICACIÓN
Importación de clases	Se importan las clases necesarias de Java para trabajar con XML.
Declaración de la clase `ActualizarXML`	Se define la clase principal **`ActualizarXML`**.
Método `main`	Se inicia el método principal **`main`** donde se ejecuta el código principal.
Bloque Try-Catch	Se inicia un bloque **`try-catch`** para manejar excepciones.
Creación de un nuevo documento XML	Se utiliza **`DocumentBuilderFactory`**, **`DocumentBuilder`** y **`Document`** para crear un nuevo documento XML.
Creación del elemento raíz (`curso`)	Se crea el elemento raíz del documento (**`curso`**) y se le añaden atributos (**`nivel`** y **`ciclo`**).
Creación de un nuevo módulo	Se crea un nuevo módulo dentro del elemento raíz (**`curso`**).
Adición de atributos al elemento raíz	Se utilizan los métodos **`setAttribute`** para añadir atributos al elemento raíz (**`curso`**).
Adición de elementos dentro del módulo	Se crea un elemento de nombre (**`nombre`**) dentro del módulo y se le añaden atributos y contenido de texto.
Transformación y escritura del XML	Se utiliza **`TransformerFactory`**, **`Transformer`**, **`DOMSource`** y **`StreamResult`** para transformar y escribir el documento XML en un archivo.
Impresión de un mensaje	Se imprime un mensaje indicando que los datos han sido actualizados en el archivo XML.
Manejo de excepciones	Se manejan las excepciones si ocurren durante el proceso

```
//creamos un elemento módulo y lo añadimos
a la raíz Element modulo = document.
createElement("módulo"); root.appendChild(modulo);

//añadimos las características del Elemento.
Tantas como queramos Element nombre = document.
createElement("nombre");  nombre.appendChild(document.
createTextNode("Acceso a datos")); nombre.
setAttribute("curso","2"); modulo.appendChild(nombre);

// añadir el resto de elementos
```

Transformación de información con Java: clase transformer

Función de Transformer	Transformer en Java facilita la conversión entre diferentes formatos jerárquicos, como el objeto Document que contiene el DOM, a un archivo de texto XML.
Adaptadores en Transformer:	Transformer trabaja con dos tipos de adaptadores: Source y Result.
Adaptadores de fuente (Source)	Clases como DOMSource, SAXSource, o StreamSource actúan como adaptadores para la fuente de información, haciendo compatibles diferentes tipos de contenedores.
Adaptadores de destino (Result)	Clases como DOMResult, SAXResult, y StreamResult actúan como adaptadores equivalentes para el contenedor destino.
Resumen	Transformer utiliza adaptadores para compatibilizar jerarquías diferentes, facilitando así la conversión de información entre formatos.

Para nuestro caso, como tenemos un DOM y lo queremos convertir a **`Stream (fichero)`**, necesitaremos un **`DomSource`** y un **`StreamResult`**. Vemos el código necesario para hacer esto:

```
Transformer trans =
TransformerFactory.newInstance().newTransformer();
DOMSource source = new DOMSource(document); // el DOM
creado previamente
StreamResult result = new StreamResult(new
FileOutputStream(nombreFichero+".xml"));
```

Y finalmente, solo nos queda la transformación de un elemento a otro, el que automáticamente generará el fichero XML de salida en el sistema de ficheros:

```
trans.transform(source, result);
```

La manipulación de documentos XML utilizando Java y XPath. XML (Extensible Markup Language) es un lenguaje de marcas utilizado para almacenar y transportar datos de manera estructurada. XPath, por otro lado, es un lenguaje utilizado para navegar y consultar documentos XML.

En este caso, el objetivo es crear un documento XML desde cero, agregar elementos y atributos, escribir el documento en un archivo y luego procesarlo utilizando XPath para extraer información específica del documento.

Este tipo de proceso es común en aplicaciones donde se necesitan manipular datos estructurados en formato XML, como configuraciones, intercambio de datos entre sistemas, generación de informes, entre otros.

El código proporcionado muestra cómo realizar estas tareas utilizando las API de XML proporcionadas por Java, como **`DocumentBuilderFactory`**, **`DocumentBuilder`**, **`Document`**, **`TransformerFactory`** y **`Transformer`** para la creación, escritura y transformación del documento XML, y **`XPath`** para el procesamiento y consulta del documento XML.

Podemos crear, escribir y procesar un documento XML en Java utilizando XPath. Este proceso es útil cuando necesitamos manipular datos estructurados en formato XML, como configuraciones, intercambio de datos, etc.

EJEMPLO

```
import java.io.File;
import java.io.FileOutputStream;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class ActualizarXML {

    public static void main(String[] args) {
        try {
            // Parseamos el archivo XML existente
            String nombreFichero = "curso"; // Nombre del archivo XML
             DocumentBuilderFactory dbFactory = DocumentBuilderFactory.
newInstance();
            DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
             Document doc = dBuilder.newDocument(); // Creamos un nuevo
documento

            // Creamos la raíz del documento y añadimos atributos
            Element root = doc.createElement("curso");
```

```
            root.setAttribute("nivel", "2");
            root.setAttribute("ciclo", "DAM");
            doc.appendChild(root);

            // Creamos un nuevo módulo y lo añadimos al curso
            Element modulo = doc.createElement("modulo");
            root.appendChild(modulo);

            // Añadimos las características del módulo
            Element nombreModulo = doc.createElement("nombre");
           nombreModulo.appendChild(doc.createTextNode("Programación de
servicios y procesos"));
            nombreModulo.setAttribute("curso", "2");
            modulo.appendChild(nombreModulo);

            // Transformamos y escribimos los cambios en el archivo XML
            TransformerFactory transformerFactory = TransformerFactory.
newInstance();
            Transformer transformer = transformerFactory.
newTransformer();
            DOMSource source = new DOMSource(doc);
            StreamResult result = new StreamResult(new
FileOutputStream(nombreFichero + ".xml"));
            transformer.transform(source, result);

            // Procesamiento de XML con XPath
            Document xmlDocument = dBuilder.parse(new
File(nombreFichero + ".xml"));
            XPath xPath = XPathFactory.newInstance().newXPath();
            XPathExpression expr = xPath.compile("/curso/modulo/
nombre");
            String nombre = (String) expr.evaluate(xmlDocument,
XPathConstants.STRING);
            System.out.println("Nombre del módulo: " + nombre);

            System.out.println("Datos actualizados en el archivo
XML.");

        } catch (Exception e) {
```

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

PASO	EXPLICACIÓN
Importación de clases	Se importan las clases necesarias de Java para trabajar con XML y XPath.
Declaración de la clase `ActualizarXML`	Se define la clase principal **`ActualizarXML`**.
Método `main`	Se inicia el método principal **`main`** donde se ejecuta el código principal.
Bloque Try-Catch	Se inicia un bloque **`try-catch`** para manejar excepciones.
Creación de un nuevo documento XML	Se crea un nuevo documento XML con un elemento raíz y atributos.
Creación de un nuevo módulo	Se crea un nuevo módulo dentro del elemento raíz y se le añaden atributos y contenido de texto.
Transformación y escritura del XML	Se transforma y escribe el documento XML en un archivo.
Procesamiento de XML con XPath	Se parsea el archivo XML y se utiliza XPath para obtener el nombre del módulo.
Impresión de un mensaje	Se imprime un mensaje indicando que los datos han sido actualizados en el archivo XML.
Manejo de excepciones	Se manejan las excepciones si ocurren durante el proceso.

05 Almacenamiento con archivos JSON

DEFINICIÓN

JSON es otro formato de texto ligero para el intercambio de datos. Las siglas **JSON** provienen de **JavaScript object notation (notación de objetos de JavaScript)**, y se trata de un subconjunto de la notación literal de objetos de este lenguaje, que se ha adoptado junto con XML como uno de los grandes estándares de intercambio y almacenamiento de datos.

5.1. El formato JSON

La sintaxis básica, junto con los tipos de datos, que podemos representar en JSON es:

Números	tanto enteros como decimales.
Cadenas	expresadas entre comillas y con la posibilidad de incluir secuencias de escape.
Booleanos	para representar los valores ***true*** y ***false***.
Null	para representar el valor nulo.
Array	para representar listas de cero o más valores, de cualquier tipo, entre corchetes y separados por comas.
Objetos	como colecciones de pares <clave>:<valor>, separados por comas y entre llaves, y de cualquier tipo de valor.

EJEMPLO

Ejemplo de fichero JSON como respuesta a una API rest sobre películas de ***Star Wars.***

```
{
   "name": "Luke Skywalker",
   "hair_color": "blond",
   "birth_year": "19BBY",
   "gender": "male",
   "films": [
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/6/",
   ],
    ...
}
```

5.2. JSON y Java. Librería org.JSON

Existe un gran abanico de librerías Java para la manipulación de documentos **JSON (GSON, Jackson, JSON. simple...)**. En esta ocasión, utilizaremos la librería `org.json`, que podemos encontrar y añadir a nuestro proyecto en el repositorio de Maven.

La librería ofrece un conjunto de **clases para procesar documentos JSON para Java**, además de **conversores entre JSON y XML.**

De entre las clases que ofrece, podríamos destacar:

org.json.JSONObject

Almacena un objeto JSON con pares atributo valor, con los tipos vistos anteriormente: **Boolean**, **JSONArray**, **Number**, **String** y **JSONObject.NULL**. Podemos añadir y eliminar atributos de este objeto mediante los métodos:

- **void put(String key, Object Valor)** → añade un objeto identificado por una clave. Podemos utilizar métodos más concretos para los tipos, ya que estos métodos están sobrecargados.
- **JSONObject get(String key)** → retorna el objeto identificado por dicha clave.
- **TIPO getTIPO(String key)** → retorna el tipo básico identificado por esta clave, donde TIPO es cualquiera de los tipos básicos (String, int, boolean, etc.)
- En caso de acceder a una clave inexistente, saltará la excepción **JSONException**.

org.json.JSONTokener

Procesa una cadena JSON; utilizado internamente por **JSONObject**.

org.json.JSONArray

Almacena una serie de valores representados por un array JSON. Dispone de mecanismos para:

- **put (JSONObject)** → almacena en un **JSONArray** el objeto indicado.
- Recorrido del array:
- **int length()** → devuelve la dimensión.
- **JSONObject get(int index)** → devuelve el elemento de la posición indicada.

org.json.JSONWriter

Ofrece una forma de producir texto JSON. Dispone de un método **append(String)** que añade más texto a un objeto JSON de tipo texto, además de los métodos **key(String)** y **value(String)** para añadir claves y valores a una cadena JSON.

5.3. Creación de un objeto JSON a partir de un objeto Java

1. Crearemos un JSONObject vacío para la representación JSON.

2. Utilizando el método get, añadiremos cada atributo del objeto al JSONObject, según su tipo básico.

3. Tomando como referencia la clase Modulo de secciones anteriores, añadiremos un método que proporcionará una representación JSON del módulo.

4. Crearemos un método en la clase Modulo para obtener la representación JSON del mismo.

Se establece un proceso simple para crear un JSONObject en Java y añadir atributos a partir de un objeto de referencia:

```
public JSONObject getAsJSON(){
    JSONObject modulo = new JSONObject();

    modulo.put("nombre", this.nombre);
    modulo.put("horas", this.horas);
    modulo.put("nota", this.nota);
```

Comentamos el código del Primer bloque.

1. // en caso de existir la posibilidad de un valor nulo, deberíamos gestionarlo:
Este comentario señala que si alguno de los atributos del objeto **Modulo** puede ser nulo, debemos gestionar este caso para evitar errores al convertirlo a JSON. En el ejemplo, se utiliza **JSONObject.NULL** para representar un valor nulo en JSON.
2.-// si no es nulo, su valor, en caso contrario, JSONObject.NULL
Este comentario explica cómo se maneja el caso de un atributo nulo utilizando una expresión ternaria. Si el atributo no es nulo, se utiliza su valor; de lo contrario, se utiliza **JSONObject.NULL**.

Ahora, vamos a desarrollar el segundo bloque de código:

1. **// Raíz del documento "curso"..**
   Este comentario indica que se está creando el objeto JSON que será la raíz del documento JSON final.
2. **// Que contendrá una lista de módulos:**
   Este comentario explica que se creará un arreglo JSON que contendrá los objetos JSON de cada módulo.
3. **// Rellenamos el array con los JSONObject, uno por cada Modulo**
   Aquí se describe el proceso de iteración sobre cada módulo en la lista **Curso**, convirtiendo cada uno en un objeto JSON utilizando el método **getAsJSON()** y agregándolo al arreglo JSON **jsarray**.
4. **// El array será una entrada en el JSONObject Raíz, por lo que debemos añadirlo**
   Este comentario indica que el arreglo JSON **jsarray** será un valor dentro del objeto JSON **curso**.

Finalmente, el código para escribir el objeto JSON en un archivo:

// como parámetro, la cantidad de espacios de indentación.
Este comentario explica que el método **toString(4)** del objeto **curso** se utiliza para convertir el objeto JSON en una cadena de texto con una indentación de 4 espacios, lo que hace que el archivo JSON resultante sea más legible.

Ahora, suponiendo que tenemos un array de Modulo, podemos generar un objeto «curso» que contendrá a todos los Modulo:

```
// Raíz del documento "curso"..
JSONObject curso=new JSONObject();

// Que contendrá una lista de módulos:
JSONArray jsarray = new JSONArray();

// Rellenamos el array con los JSONObject, uno por cada Modulo
// llamando a la anterior función anterior Curso es una
colección de módulos
for(Modulo m:this.Curso)  {
    JSONObject modulo_json=m.getAsJSON();
     // añadimos el objeto al vector
     jsarray.put(modulo_json);
}
```

```
// El array será una entrada en el JSONObject Raíz, por lo
que debemos añadirlo
curso.put("curso", jsarray);
```

Ahora solo nos queda el objeto raíz, guardarlo en un fichero de texto JSON. Para ello:

```
try {
        FileWriter file = new FileWriter("curso.json");
        file.write(curso.toString(4)); // 4 espacios de
indentación en el fichero
        file.close();
    } catch (IOException e) {
         System.out.println("Error en la creación del fichero
json");
     }
```

Como podemos observar, el método interno toString de la clase JSONObject devuelve esa representación en formato texto. Como parámetro, la cantidad de espacios de indentación.

5.4. Lectura de ficheros JSON

La lectura de ficheros JSON es muy sencilla, ya que **el propio constructor de `JSONObject` permite crearlos a partir de un `String`.** Si conseguimos que dicho `String` contenga el fichero de texto, ya lo tenemos conseguido, por tanto:

```
try {
    // Con FileReader leemos carácter a carácter y lo
guardamos en un String
    FileReader file = new FileReader("fichero.JSON");
    String myJson="";

    int i;
    while ((i=file.read()) != -1)
          myJson=myJson+((char) i);
     file.close();

    // Con el constructor de JSONObject, el JSON lo generamos
a partir del String:
    JSONObject modulos=new JSONObject(myJson);
```

```
} catch (Exception e)
{
    System.out.println("Error cargando el fichero");
}
```

El siguiente ejemplo se actualiza un archivo JSON que contiene información sobre módulos de un curso. Se lee el archivo JSON, se procesa su contenido para actualizar los datos de los módulos y luego se guardan los cambios en el mismo archivo. A continuación, explicaré el código en forma de tabla con opciones:

PASO	EXPLICACIÓN	OPCIONES
1. Lectura del archivo JSON	Se lee el archivo JSON y se guarda su contenido en un **`StringBuilder`**.	
2. Creación del objeto JSON	Se crea un objeto **`JSONObject`** a partir del contenido del **`StringBuilder`**.	
3. Actualización de datos	Se accede al array de módulos dentro del objeto JSON y se itera sobre cada módulo. Se obtienen y actualizan los valores de los módulos según la lógica deseada. » Se incrementan en 1 las horas de cada módulo.	
4. Guardado de los cambios	Se sobrescribe el archivo JSON original con los cambios realizados. Se utiliza un **`FileWriter`** para escribir el objeto JSON modificado en el archivo.	
5. Manejo de excepciones	Se manejan las excepciones que pueden ocurrir durante el proceso de actualización y escritura del archivo JSON.	» Si ocurre un error, se muestra un mensaje indicando el problema.

Este enfoque permite comprender de manera estructurada cada paso del proceso de actualización del archivo JSON, desde la lectura inicial hasta el manejo de posibles errores.

El código actualizado:

```
import org.json.*;

public class ActualizacionJSON {

    public void actualizarJSON(String nombreArchivo) {
        try {
```

```
            // Lectura del archivo JSON
            FileReader fileReader = new FileReader(nombreArchivo);
            StringBuilder jsonBuilder = new StringBuilder();
            int character;
            while ((character = fileReader.read()) != -1) {
                jsonBuilder.append((char) character);
            }
            fileReader.close();

            // Creación del objeto JSONObject a partir del JSON
            JSONObject modulosJSON = new JSONObject(jsonBuilder.
toString());

            // Actualización de datos
            JSONArray modulosArray = modulosJSON.getJSONArray("curs");
            for (int i = 0; i < modulosArray.length(); i++) {
                JSONObject modulo = modulosArray.getJSONObject(i);
                // Actualización de los valores del módulo
                String nombre = modulo.getString("nombre");
                int horas = modulo.getInt("horas");
                double nota = modulo.getDouble("nota");

                // Supongamos que queremos incrementar las horas de
                cada módulo en 1 horas++;
                // También podríamos actualizar otros atributos o
aplicar lógica de actualización

                // Actualizamos los valores en el objeto JSON
                modulo.put("horas", horas);
            }

            // Guardar los cambios en el archivo JSON
            FileWriter fileWriter = new FileWriter(nombreArchivo);
            fileWriter.write(modulosJSON.toString(4)); // 4 espacios de
indentación
            fileWriter.close();

            System.out.println("Datos actualizados y guardados en el
archivo JSON.");
```

```
        } catch (IOException | JSONException e) {
            System.out.println("Error al actualizar el archivo JSON: "
+ e.getMessage());
        }
    }

    public static void main(String[] args) {
        ActualizacionJSON actualizador = new ActualizacionJSON();
        actualizador.actualizarJSON("fichero.JSON");
    }
}
```

UAX FP