1. Introducción

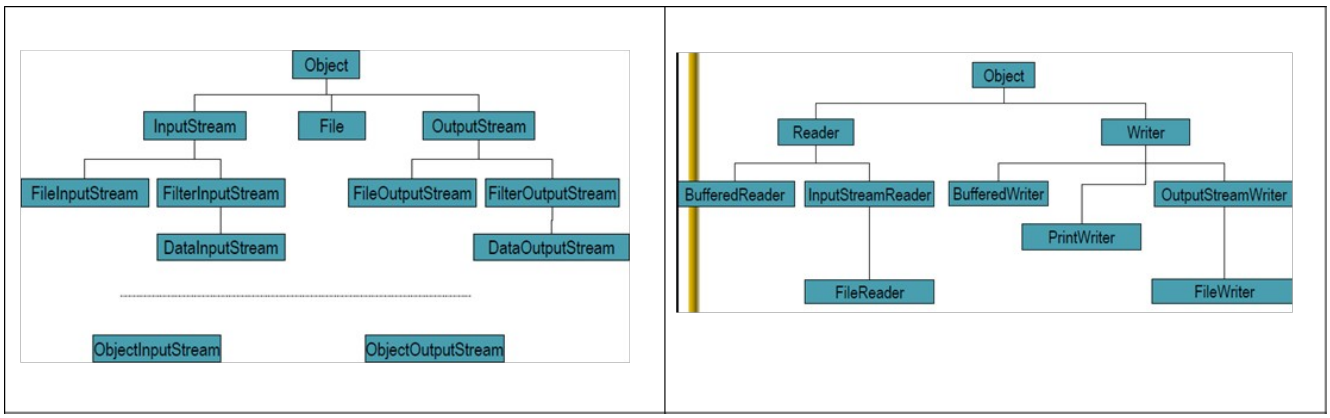
A las operaciones, que constituyen un flujo de información del programa con el exterior, se les conoce como Entrada/Salida (E/S).

Las operaciones de E/S en Java las proporciona el paquete **java.io** que incorpora i*nterfaces, clases y excepciones* para acceder a todo tipo de ficheros.

Las clases de E/S de java.io las podemos agrupar fundamentalmente en:

- Clases para operar con ficheros en el sistema de ficheros local (File, FileFilter, ..)
- Clases para leer entradas desde un flujo de datos (InputStream, Reader,...)
- Clases para escribir entradas a un flujo de datos(OutputStream, Writer,...)
- Clases para gestionar la serialización de objetos

Algunas de estas clases las puedes ver en la siguiente figura:



La documentación oficial del paquete java.io la puedes consultar en la siguiente dirección:

http://docs.oracle.com/javase/6/docs/api/
http://docs.oracle.com/javase/7/docs/api/

2. Clases Asociadas a la gestión de archivos y directorios

Veremos las siguientes:

- La **clase File** permite realizar la mayoría de las operaciones con ficheros y directorios(creación, eliminación, ver propiedades, listar directorios, etc).

- La **interface FilenameFilter** permite filtrar ficheros, o sea, obtener aquellos con una característica determinada, como puede ser que tengan la extensión .odt, o la que nos interese.

1.1. La clase File

La clase ***File*** del paquete ***java.io*** **:**

- proporciona una representación abstracta de ficheros y directorios.

- permite examinar y manipular archivos y directorios, independientemente de la plataforma en la que se esté trabajando: Linux, Windows, etc.

Las instancias de la clase *File* representan nombres de archivo, no los archivos en sí mismos.

El archivo correspondiente a un nombre puede ser que no exista, por esta razón habrá que controlar las posibles excepciones.

Un objeto de clase *File* permite examinar el nombre del archivo, descomponerlo en su rama de directorios o crear el archivo si no existe.

Constructores para crear objetos de tipo File (observa la forma de indicar la ruta en cada sistema):

- *File(String directorioyfichero):*
  - en Linux:
    File fichero1 = new File("/home/ad/ut1/ejemplo1.txt");
  - en Windows:
    File fichero1 = new File("c:\\ad\\ut1\\ejemplo1.txt");
- *File(String directorio, String nombrefichero)*
    String dir ="c:\\ad\\ut1";
    File fichero2 = new File(dir, "ejemplo2.txt");
- *File (File file, String fichero)*
    File file = new File(dir); //siendo dir la ruta o path del ejemplo anterior
    File fichero3 = new File (file, "ejemplo3.txt");

Para archivos que existen, a través del objeto *File*, un programa puede examinar los atributos del archivo, cambiar su nombre, borrarlo o cambiar sus permisos.

Algunos métodos importantes de la clase File son los siguientes:

Método	Descripción
list()	Devuelve un vector de String con los nombres de los archivos.
listFiles()	Devuelve un vector de objetos File.
exists()	Devuelve true si el fichero/directorio existe
getName()	Devuelve el nombre del fichero o directorio
getParent()	Devuelve el nombre del directorio padre o null si no existe.
getAbsolutePath()	Devuelve la ruta absoluta
canRead()	Devuelve true si el fichero se puede leer
canWrite()	Devuelve true si el fichero se puede escribir
createNewFile()	Crea un nuevo fichero, vacío, asociado a File, si y solo si no existe un fichero con dicho nombre.
delete()	Borra el fichero o directorio asociado al File
isDirectory()	Devuelve true si el objeto File corresponde a un directorio.
isFile()	Devuelve true si el objeto File corresponde a un fichero
mkdir(); mkdirs()	Crea un directorio
renameTo()	Renombra el objeto File
length()	El tamaño del fichero en bytes
lastModified()	Devuelve la fecha, medida en milisegundos que han transcurrido desde el 01/01/1970 hasta la última modificación.

Para indicar el directorio actual se debe utilizar “.”

EJEMPLO 1. Listado de ficheros del directorio actual. [VerDir.java]

```
import java.io.*;
public class VerDir{
  public static void main(String[] args)
    { System.out.println("Ficheros en direcotrio actual: ");
    File f = new File (".");  //crea el objero file apuntando al directorio actual
    String[] archivos = f.list(); //guarda las entradas del directorio en el array archivos
    for (int i=0; i < archivos.length; i++) {
        System.out.println(archivos[i]);
     }
   }
}
```

EJEMPLO 2. Muestra información asociada al fichero VerInf.java [VerInf.java]

```
import java.io.*;
public class VerInf{
 public static void main(String[] args)
  { System.out.println("Información sobre el fichero");
  File f = new File ("c:\\ad\\ut1\\VerInf.java");
  if(f.exists()){
        System.out.println("Nombre del fichero: " + f.getName());
        System.out.println("Ruta: " + f.getPath());
        System.out.println("Ruta absoluta: " + f.getAbsolutePath());
        System.out.println("Se puede leer: " + f.canRead());
        System.out.println("Se puede escribir: " + f.canWrite());
        System.out.println("Tamaño: " + f.length());
        System.out.println("Es un directorio: " + f.isDirectory());
        System.out.println("Es un fichero: " + f.isFile());
   }
  }
}
```

Cuando queramos **crear un fichero**, podemos proceder del siguiente modo:

```
try {
        // Creamos el objeto que encapsula el fichero
        File fichero = new File("c:\\ad\\ut1\\miFichero.txt");
        // A partir del objeto File creamos el fichero físicamente
        if (fichero.createNewFile())
            System.out.println("El fichero se ha creado correctamente");
        else
            System.out.println("No ha podido ser creado el fichero");
} catch (Exception ioe) {
              ioe.getMessage();
}
```

Para **crear directorios:**

```
try {
      //Declaración de variables
      String directorio = "C:\\ad";
      File d = new File(directorio);
      //Crear un directorio

      if (d.mkdir())
         System.out.println("Directorio: " + directorio + " creado");

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

EJEMPLO 3. Crea un nuevo directorio en el actual y en él, crea dos ficheros vacíos. Uno de ellos se renombra después como 'NUEVODIR'. [CrearDir.java]

```
import java.io.*;
public class CrearDir {
        public static void main(String[] args){
                File d = new File ("NUEVODIR"); //directorio que creo a partir del actual
                File f1 = new File(d, "FICHERO1.TXT");
                File f2 = new File(d, "FICHERO2.TXT");
                d.mkdir(); //crea físicamente el directorio
                try{
                        if(f1.createNewFile())
                                System.out.println("FICHERO1 creado correctamente....");
                        else System.out.println("No se ha podido crear FICHERO1");
                        if(f2.createNewFile())
                                System.out.println("FICHERO2 creado correctamente");
                        else
                                System.out.println("No se ha podido crear FICHERO2....");
                }catch (IOException ioe) {ioe.printStackTrace();}

                f1.renameTo(new File(d,"FICHERO1NUEVO")); //renombre FICHERO1
                try{
                         File f3 = new File("NUEVODIR\\FICHERO3.TXT");
                        f3.createNewFile();//crea FICHERO3 en NUEVODIR
                }catch (IOException ioe) {ioe.printStackTrace();}
         }
}
```

Para **borrar un fichero** o **directorio** se utiliza el método **delete()**. Por ejemplo:

```
if(f2.delete())
        System.out.println("Fichero borrado...");
else
        System.out.println("No se ha podido borrar el fichero");
```

Para **borrar un directorio** con Java, tendremos que borrar cada uno de los ficheros y directorios que éste contenga. Al poder almacenar otros directorios, se podría recorrer recursivamente el directorio para ir borrando todos los ficheros. Se puede listar el contenido del directorio e ir borrando con: ***File[] ficheros = directorio.listFiles();***

Recuerda que sabemos si el elemento es un directorio mediante ***isDirectory()***

EJEMPLO 4. Se elimina el directorio NUEVODIR del directorio actual, teniendo en cuenta que puede tener ficheros (solo se contempla la posibilidad de que haya ficheros) [BorrarDir.java]

```
import java.io.File;
public class BorrarDir {
  private static final String rutaDirectorio = "NUEVODIR";  //ruta del directorio a borrarse
  public static void main(String[] args) {
    File f;
    String[] archivos;
    File d = new File(rutaDirectorio); //objeto File hacia la ruta del directorio padre
    //si la ruta existe
    if (d.exists()) {
      archivos = d.list();  //obtengo la lista de archivos
      for (int i = 0; i < archivos.length; i++) {
          f = new File(d, archivos[i]); //objeto File hacia la ruta completa del archivo i-ésimo
          if (!f.delete()) { //borra el archivo (se recibe true si la operación tiene éxito)
             System.out.printf("\nNo se pudo borrar el fichero %s\n", f.getName());
        }
      }
      //borra la ruta padre (no tendrá efecto si quedó algún fichero sin borrar)
      d.delete();

    } else {
      //si la ruta no existe
      System.out.printf("\nNo se encuentra el directorio %s\n", rutaDirectorio);
    }
  }
}
```

1.2. Interface FilenameFilter

Hemos visto como obtener la lista de ficheros de una carpeta o directorio. A veces, nos interesa ver no la lista completa, sino los archivos que encajan con un determinado criterio.

Por ejemplo, nos puede interesar un filtro para ver los ficheros modificados después de una fecha, o los que tienen un tamaño mayor del que el que indiquemos, etc.

El interface **FilenameFilter** se puede usar para crear filtros que establezcan criterios de filtrado relativos al nombre de los ficheros. Una clase que lo implemente debe definir e implementar el método:

boolean accept(File dir, String nombre)

Este método devolverá *true* en el caso de que el fichero cuyo nombre se indica en el parámetro *nombre* aparezca en la lista de los ficheros del directorio indicado por el parámetro *dir*.

EJEMPLO 5. En el siguiente ejemplo vemos cómo se listan los ficheros de la carpeta c:\ad\ud2 que tengan la extensión .java. Usamos ***try y catch*** para capturar las posibles excepciones, como que no exista dicha carpeta. [Filtrar.java]

```
import java.io.File;
import java.io.FilenameFilter;

public class Filtrar implements FilenameFilter
    { String extension;
    // Constructor
    Filtrar(String extension){
        this.extension = extension;
    }

    public boolean accept(File dir, String name){
      return name.endsWith(extension); //endsWith(sr) devuelve true si la cadena finaliza en el valor de sr
    }
    public static void main(String[] args) {
        try {
                // Obtendremos el listado de los archivos de ese directorio
            File fichero=new File("c:\\ad\\ut1\\");
            String[] listadeArchivos = fichero.list();

                // Filtraremos por los de extension .java
            listadeArchivos = fichero.list(new Filtrar(".java"));

                // Comprobamos el número de archivos en el listado
            int numarchivos = listadeArchivos.length ;

                // Si no hay ninguno lo avisamos por consola
            if (numarchivos < 1)
                System.out.println("No hay archivos que listar");
                // Y si hay, escribimos su nombre por consola.
            else
            {
                for(int conta = 0; conta < listadeArchivos.length; conta++)
                    System.out.println(listadeArchivos[conta]);
            }
        }
        catch (Exception ex) {
            System.out.println("Error al buscar en la ruta indicada");
        }
    }
}
```

De igual forma, el interface **FileFilter** se puede usar para crear filtros que establezcan criterios de filtrado relativos a objetos File. Se puede usar una instancia de FileFilter con el método ***listFiles(FileFilter filtro)*** de la clase File para filtras objetos File.

El método **boolean accept(File pathname)** es el que establece el test de filtrado, y es el que debe sobrescribir la clase que implemente a FileFilter.

2. Flujos o Streams. Tipos

El siguiente esquema ilustra el principio básico de un programa que **lee datos de un origen y los escribe en un destino.**

Como ejemplos típicos en Java, de **origen y destino** podemos citar: ficheros o archivos, conexiones de red, buffers de memoria y tuberías o pipes.

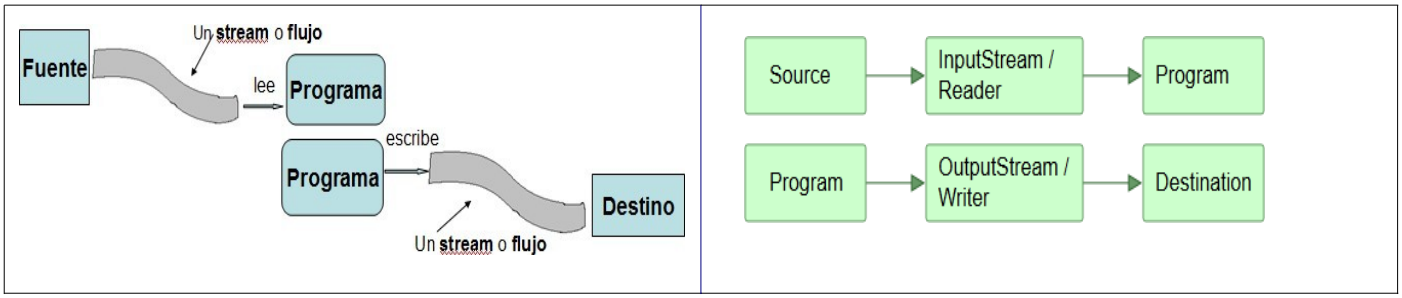
Flujo o stream es una abstracción para tratar la comunicación de información entre un origen y un destino. La vinculación de este flujo al dispositivo físico la hace el sistema de entrada y salida de Java.

En Java, un programa que necesite leer datos de un origen o escribir datos en un destino, lo hace a través de un flujo o stream.

Los **tipos de flujos** definidos en **java.io** son:

- **Flujos de bytes(8 bits).** Reciben y envían datos en formato binario.
  - Lectura/Escritura de datos binarios.
  - Para el tratamiento de los flujos de bytes Java tiene dos clases abstractas: **InputStream** y **OutputStream**
  - Estas dos clases definen los métodos que sus subclases tendrán implementados y, de entre todos, destacan **read()** y **write()** que leen y escriben bytes de datos respectivamente.

- **Flujos de caracteres (16 bits).** Reciben y envían datos como secuencia de caracteres.
  - Lectura/Escritura de caracteres.
  - Para el tratamiento de los flujos de caracteres Java tiene dos clases abstractas: **Reader** y **Writer.**
  - Dichas clases manejan flujos de caracteres Unicode. Y también de ellas derivan subclases concretas que implementan los métodos definidos en ellas siendo los más destacados los métodos **read()** y **write()** que leen y escriben caracteres de datos respectivamente.



En java, existen dos tipos de archivos**:**

- **Archivos Binarios**, basados en flujos de bytes. Se crean y manejan usando flujos de bytes.
- **Archivos de Caracteres**, basados en flujos de caracteres. Se crean y manejan usando flujos de caracteres.

Las clases fundamentales para el **manejo de archivos Binarios** son**:**

- **FileInputStream** para lectura
- **FileOutputStream**para escritura

Junto a ambas se pueden utilizar, entre otras, las siguientes clases:

- **ObjectInputStream / ObjectOutputStream** para serialización de objetos que implementen la interfaz Serializable, y poder así persistir esos objetos.
- **DataInputStream/DataOutputStream** para realizar lectura/escritura de datos primitivos como : int, long, float, double, etc

Las clases fundamentales para el **manejo de archivos de Caracteres** son**:**

- **FileReader** para lectura
- **FileWriter** para escritura

Junto a ambas se pueden utilizar, entre otras, las siguientes clases:

- **CharArrayReader y CharArrayWriter.** Para acceso a caracteres: leen y escriben un flujo de caracteres en un array de caracteres.

Además van a ser interesantes las **clases que proporcionan buferización** como:

- **BufferedInputStream, BufferedOutputStream, BufferedReader y BufferedWriter** que añaden un **buffer** intermedio, minimizando los accesos a disco, y consiguiendo que las operaciones de lectura/escritura se realicen de manera más eficiente.

Si se usan sólo FileInputStream, FileOuputStream, FileReader o FileWriter, cada vez que se efectúa una lectura o escritura, se hace físicamente en el disco duro. Si se leen o escriben pocos caracteres cada vez, el proceso se hace costoso y lento por los muchos accesos a disco duro.

Las clases **BufferedReader, BufferedInputStream, BufferedWriter y BufferedOutputStream** añaden un **buffer** intermedio. Cuando se lee o escribe, esta clase controla los accesos a disco. Así, si vamos escribiendo, se guardarán los datos hasta que haya bastantes datos como para hacer una escritura eficiente.

Al leer, la clase leerá más datos de los que se hayan pedido. En las siguientes lecturas nos dará lo que tiene almacenado, hasta que necesite leer otra vez físicamente. Esta forma de trabajar hace los accesos a disco más eficientes y el programa se ejecuta más rápido.

3. Formas de acceso a un fichero

La forma de acceso a un fichero determina la manera en la que se puede acceder a la información contenida en él , tanto para lectura como para escritura; si se puede por ejemplo localizar de forma directa cierto registro o dato, o por el contrario es necesario leer desde el principio en el fichero hasta llegar al registro o dato buscado.

Los tipos de acceso que soporta Java son:

- **Acceso secuencial:** En este caso los datos se leen de manera secuencial, desde el comienzo del archivo hasta el final. Para acceder a un dato concreto hay que leer los anteriores. Si se añade, será al final del fichero, a partir del último dato escrito. No es posible hacer inserciones entre datos escritos.

  Este tipo de acceso es soportado mediante las clases:

  - FileInputStream/FileOutputStream
  - FileReader/FileWriter.

- **Acceso aleatorio:** permite acceder a los datos directamente, sin necesidad de leer los anteriores y además en cualquier orden. Esto implica que el archivo debe estar disponible en su totalidad al momento de ser accedido. Se manipulan posiciones relativas (número de byte) en vez de direcciones absolutas (pista, sector), lo que hace al programa independiente de la dirección absoluta del fichero en el disco. Este tipo de acceso es soportado mediante la clase **RandomAccessFile.**

4. Clases para la gestión de flujos de datos

En este apartado veremos ejemplos de las diferentes operaciones que se pueden realizar sobre archivos binarios y de texto. Para ello, utilizaremos las clases apropiadas de Java, tanto para acceso secuencial como para acceso aleatorio.

4.1. Ficheros de texto

Las clases ***FileReader*** y ***FileWriter*** están optimizadas para trabajar con caracteres y con texto en general, debido a que tienen en cuenta que cada carácter Unicode está representado por dos bytes.

- Cuando trabajamos con ficheros, cada vez que leemos o escribimos en uno, debemos hacerlo dentro de un manejador de excepciones **try_catch** o mediante **throws.**
- Al usar la clase **FileReader,** se puede generar la excepción ***FileNotFoundExcepcion*** (fichero inexistente o no válido)
- Al usar la clase **File Writer,** se puede generar la excepción ***IOException*** (disco lleno o protegido contra escritura).

Los siguientes **métodos de FileReader**, para lectura devuelven el número de caracteres leídos o -1 si es fin de fichero.

- *int read():* lee un carácter y lo devuelve.
- *int read(char[] buf)*: lee hasta *buf.length* caracteres. Los caracteres leídos del fichero, se van almacenando en *buf.*
- *int read(char[] caracteres, int desplazamiento, int cantidad)*: lee hasta la cantidad de caracteres indicados por el parámetro cantidad y los almacena en el array caracteres comenzando desde la posición indicada por desplazamiento.

Para leer, por ejemplo, solo los primeros 20 caracteres escribimos:

```
char b[]=new char[20];
fr.read(b);
System.out.println(b);
```

Algunos de los **métodos de FileWriter** para escritura de caracteres son:

- *void write(int c)*: escribe un carácter.
- *void write(char[] buf)*: escribe un array de caracteres
- *void write (char[] buf, int desplazamiento, int n)*: escribe *n* caracteres de datos del array *buf*, comenzando en *buf[desplazamiento].*
- *void write(String str)*: escribe una cadena de caracteres.
- *append(char c)*: añade un carácter a un fichero.

Estos métodos pueden lanzar la excepción ***IOException***.

```
String prov[]={"Almería", "Granada","Cadiz", "Málaga", "Córdoba", "Sevilla", "Huelva"};
for (int i=0;i<prov.length;i++)
      fw.write(prov[i]);
```

En todos los casos anteriores, si el fichero existe se sobreescribirá, perdiendo su contenido anterior. Si queremos evitar la sobreescritura y añadir caracteres al final, utilizaremos **FileWriter** de la forma:

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

4.1.1. BufferedReader

FileReader no contiene métodos para leer líneas completas, pero ***BufferedReader*** sí, con el método ***readLine()*** que devuelve la línea leída o *null* si no hay nada o se llega al final. Soporta el método **read()** lee un carácter.

Para construir un ***BufferedReader*** necesitamos la clase ***FileReader***.

4.1.2. BufferedWritter

Deriva de **Writer** y añade un buffer para realizar una escritura eficiente de caracteres. Para construir un ***BufferedWriter*** necesitamos la clase ***FileWriter***.

4.1.3. PrintWritter

Posee los métodos print(String cadena) y println(String cadena) (idénticos a System.out) para escribir en un fichero. Para poder usar esta clase, primero debe crearse el flujo hacia el fichero sobre el que se desea escribir:

```
PrintWriter pw=new PrintWriter(new FileWriter(new File("c:\\ad\\ut1\\FichTexto.txt");
for(int i=1; i<11;i++)
    pw.println("Fila número:" + i);

pw.close();
```

4.2. Ficheros binarios

Los **Fichero binarios** almacenan secuencias de dígitos binarios que no son legibles directamente desde un editor de texto, como ocurría con los ficheros de texto.

- Dos clases que permiten trabajar con ellos son ***FileInputStream*** y ***FileOutputStream***, para lectura y escritura respectivamente.

- Al usar la clase ***FileInputStream*** se puede generar la excepción ***FileNotFoundException*** (fichero inexistente o no válido), y al usar ***FileOutputStream*****,** se puede generar la excepción ***IOException*** (disco lleno o protegido contra escritura).

Los **métodos para lectura** que proporciona **FileInputStream** devuelven el número de bytes leídos o -1 si se ha llegado al final del fichero. Entre ellos están:

- *int read(): lee un byte y lo devuelve*
- *int read(byte[] b): lee hasta b.length bytes de datos de una matriz de bytes*
- *int read(byte[] datos, int desplazamiento, int cantidad)*: lee la cantidad de bytes indicados por el parámetro cantidad y los almacena en el array datos comenzando desde la posición indicada por desplazamiento.

Entre los **métodos para escritura** que proporciona **FileOutputStream** destacamos los siguientes:

- *void write(int b): escribe un byte*
- *void write(byte [] b) escribe un array de b bytes.*
- *void write(byte[] datos, int desplazamiento, int cantidad)*: Escribe hasta la cantidad de bytes indicados por el parámetro cantidad comenzando en la posición desplazamiento del array de datos.

Por defecto, la escritura sobreescribe el contenido del fichero. Para evitarlo se debe crear el flujo de la siguiente forma:

FileOutputStream fileout = new FileOutputStream(fichero, true)

Por último, al igual que sobre los flujos de caracteres, sobre los flujos de bytes también se pueden crear **Buffers**, usando las clases ***BufferedInputStream*** y ***BufferedOutputStream***.

BufferedInputStream entrada = new BufferedInputStream(new FileInputStream(FICHERO))

4.2.1. Datos de tipo primitivo en ficheros binarios

Las clases ***DataInputStream*** **y** ***DataOutputStream*** se usan para leer y escribir datos de tipos primitivos (int, float, long, etc). Estas clase definen diversos métodos **readXXX() y writeXXX()** para la lectura y la escritura de datos de tipo primitivo sobre un fichero.

Algunos **métodos de DataInputStream** son: (leen y devuelven un dato del tipo indicado)

- *int readInt(), char readChar(), long readLong(), float readFloat(), double readDouble(), String readUTF()*

Algunos **métodos de DataOutputStream** son: (escriben datos del tipo indicado)

- *void writeInt(int v)*
- *void writeChar(int v)*
- *void writeFloat(float v)*
- *void writeLong(long v)*
- *void writeUTF(String str)*
- *void writeChars(String str)*

4.2.2. Serialización de objetos

¿Cómo guardar un objeto completo, por ejemplo un objeto empleado, en un fichero?

Java permite guardar objetos en ficheros binarios mediante la **serialización**, que consiste en convertir el objeto en una secuencia de bytes que posteriormente puede ser restaurada al objeto original.

- El objeto debe implementar la **interface Serializable** que dispone de métodos para guardar y leer objetos en un fichero binario.

- Se deben usar las clases **ObjectInputStream** y **ObjectOutputStream** permiten leer y escribir objetos serializables mediante los métodos:

  - ***Object readObject()***:Usado para leer un objeto del fichero. Puede lanzar las excepciones: ***IOException, ClassNotFoundException***

  - *void* ***writeObject(Object obj)***: Usado para escribir el objeto especificado en el fichero. Puede lanzar la excepción ***IOException***.

ObjectOutputStream puede darnos algunos problemas, por ejemplo:

- si una vez que hemos escrito datos en el fichero lo cerramos, y después volvemos a abrirlo para añadirle datos (*FileOutputStream(fichero, true)*), entonces se añade una cabecera entre los objetos introducidos anteriormente y los nuevos. Esto origina el problema de que al leer el fichero se produzca una excepción.

Para resolver este problema es necesario eliminar esa cabecera intermedia. Esto se puede conseguir sobreescribiendo el método *writeStreamHeader()* de la clase FileOutputStream, que es el encargado de escribir las cabeceras.

4.3. Ficheros de acceso aleatorio

Todo lo que se ha visto hasta el momento son operaciones realizadas sobre ficheros a los que se accede de forma secuencial. Se empieza la lectura en el primer byte, en el primer carácter o en el primer objeto, y seguidamente se leían los siguientes uno a continuación de otro hasta llegar al final del fichero. Igualmente, cuando escribimos los datos en el fichero estos se van escribiendo a continuación de la última información escrita.

Java dispone de la clase **RandomAccessFile** dispone de métodos para acceder de forma aleatoria a un fichero binario y para posicionarnos en una posición concreta del mismo.

Este tipo de ficheros maneja un puntero que indica la posición actual en el fichero. Al crear el fichero se coloca en el byte 0, apuntado al principio del mismo. Las sucesivas lecturas y escrituras, ajustan el puntero según la cantidad de bytes leídos o escritos.

Esta clase **no forma parte de la jerarquía InputStream/OutputStream,** ya que su comportamiento es totalmente distinto, pudiendo avanzar y retroceder dentro de un fichero.

La **creación de un fichero de acceso aleatorio** o directo se puede hacer de las siguientes formas:

- Escribiendo el nombre del fichero:

  fichero= new RandomAccessFile(String nombre, String modoAcceso)

- Con un objeto File:

  fichero=new RandomAccessFile(File fich, String modoAcceso)

En ambos casos *modoAcceso* puede ser:

- **r**: Abre el fichero en modo de solo lectura. El fichero debe existir. Una operación de escritura sobre este fichero lanzará la excepción ***IOException***.

- **rw**: Abre el fichero en modo lectura y escritura. Si el fichero no existe lo crea.

Para **lectura y escritura en el fichero de acceso directo** pueden usarse los métodos ***read()*** y ***write()*** de las clases *DataInputStream* y *DataOutputStream.*

Para **manejar el puntero o apuntador asociado a un objeto RandomAccessFile** existen diferentes métodos, entre ellos destacamos los siguientes:

- ***long getFilePointer()***: devuelve la posición actual del puntero del fichero.

- ***void seek(long position)***: coloca el puntero del fichero en una *posición* determinada, desde el comienzo del mismo.

- ***long length()***: devuelve el tamaño del fichero en bytes. La posición length() marca el final del fichero.

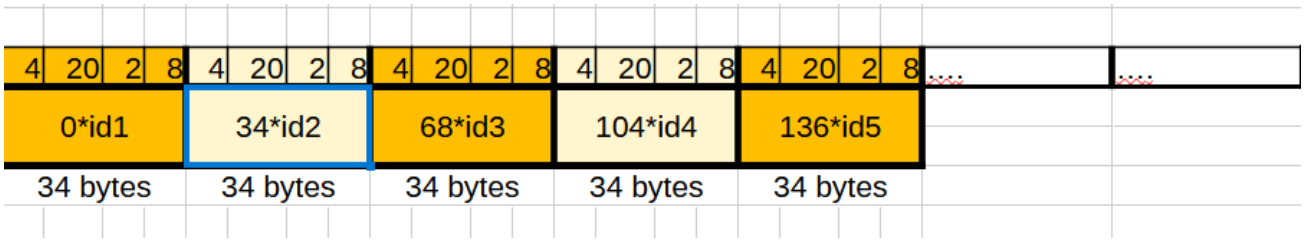
- ***int skipBytes(int desplazamiento)***: desplaza el puntero desde la posición actual el número de bytes indicados en desplazamiento.

4.3.1 Operaciones sobre ficheros de acceso aleatorio.

Escritura

En el ejemplo **EscribirFicheroAleatorio.java**, se insertan datos de empleados en un fichero aleatorio. Los datos a insertar son: identificador (valor mayor que 0), apellido, numero de departamento y salario. La longitud del registro de cada empleado es la misma (34 bytes) y los tipos que se insertan y su tamaño en bytes son:

- Identificador, entero que ocupa 4 bytes.
- Apellido, cadena de 10 caracteres. Como Java utiliza caracteres UNICODE, cada carácter de una cadena ocupa 16 bits (2 bytes). Por tanto, apellido ocupa 20 bytes.
- Departamento, tipo entero corto (short) que ocupa 2 bytes.
- Salario, tipo double, ocupa 8 bytes.



Lectura

Para realizar la lectura del fichero completo, se deben ir leyendo los registros en el mismo orden en el que fueron escritos. En este caso no es necesario posicionarse porque se hace un recorrido secuencial byte a byte. La lectura finalizará cuando la posición actual del fichero sea igual al tamaño del fichero:

file.getFilePointer()= =file.length()

Además, solo se visualizarán los registros cuyo identificador sea mayor que 0, dado que si es 0, quiere decir que se trata de un hueco.

Ver: LeerTodosRegistros.java

Búsqueda/Visualización

También sobre este tipo de ficheros se pueden hacer operaciones de búsqueda/visualización de un registro concreto. En ese caso es necesario pasar como parámetro el identificador del registro a buscar/visualizar. A partir del identificador se calcula la posición de inicio del registro, y si esta es mayor o igual a la longitud del fichero, querrá decir que el registro no está. En caso contrario, el registro existe y por tanto, se deberá posicionar el puntero para proceder a la lectura del registro.

Ver: VisualizarUnRegistro

Añadir

Para añadir registros, simplemente se debe posicionar el puntero del fichero al final del mismo, y se añade el nuevo registro:

```
long posicion=file.length();
file.seek(posicion);
```

Actualizar

Para modificar un registro determinado, se accede a su posición y se llevan a cabo las modificaciones correspondientes del o de los campos deseados.

Por ejemplo, si se quiere modificar el departamento a 40 e incrementar en 300 el salario del registro cuyo identificador es 4, se deben seguir los siguientes pasos:

1. Se localiza del registro con el identificador determinado.
2. Se localiza dentro de dicho registro la posición del departamento, posicionándose para asignar en esa posición el nuevo valor.
3. Una vez actualizado el departamento, de nuevo debe localizarse dentro del registro la posición del salario, posicionándose para leer el valor actual del salario, se retrocede para volver a la posición inicial del salario, y finalmente, se actualiza el valor del salario.

Ver: ModificarEliminarRegistro.

Borrar

Para el borrado de registros, estos no se pueden borrar físicamente, se debe hacer un borrado lógico. Para ello se marcará el registro a eliminar poniendo en el identificador un valor no válido, por ejemplo, 0. También se puede hacer añadiendo a los registros un campo lógico que tendrá valor 1 si el registro está activo o 0 si no está activo.

Ver: ModificarEliminarRegistro.

5. Trabajo con ficheros XML

El trabajo con ficheros XML se puede llevar a cabo usando dos enfoques muy diferentes:

1. DOM (Modelo de Objetos de Documento).

   Almacena toda la estructura del documento en memoria en forma de árbol con nodos padre, nodos hijo y nodos hoja. Una vez creado el árbol, se van recorriendo los distintos nodos y se analiza a qué tipo particular pertenecen. Tiene su origen en el W3C.

2. SAX (Simple API para XML).

   Lee un fichero XML de forma secuencial y produce una secuencia de eventos (comienzo y fin de documento, comienzo y fin de etiqueta….) en función de lo que va leyendo. Cada evento llama a un método establecido por el desarrollador.

Sin embargo, estas metodologías no fueron diseñadas teniendo presente a Java. Con la idea de tener una metodología más acorde con Java surgió JDOM (Java Document Object Model).

Esta última permite crear documentos, navegar por su estructura, añadir, modificar o borrar elementos.

En definitiva, JDOM es una librería para Java diseñada específicamente para trabajar con documentos XML de una manera sencilla, legible y natural, siguiendo la filosofía de que un documento XML debe manipularse como un árbol de objetos Java, no con APIs complejas y de bajo nivel como SAX o DOM.

Destacar que esta librería ya ha evolucionado a JDOM2. Para incluirla en un proyecto Gradle es necesario incluir su dependencia:

```
implementation 'org.jdom:jdom2:2.0.6'
```

Algunas de sus librerías más importantes son:

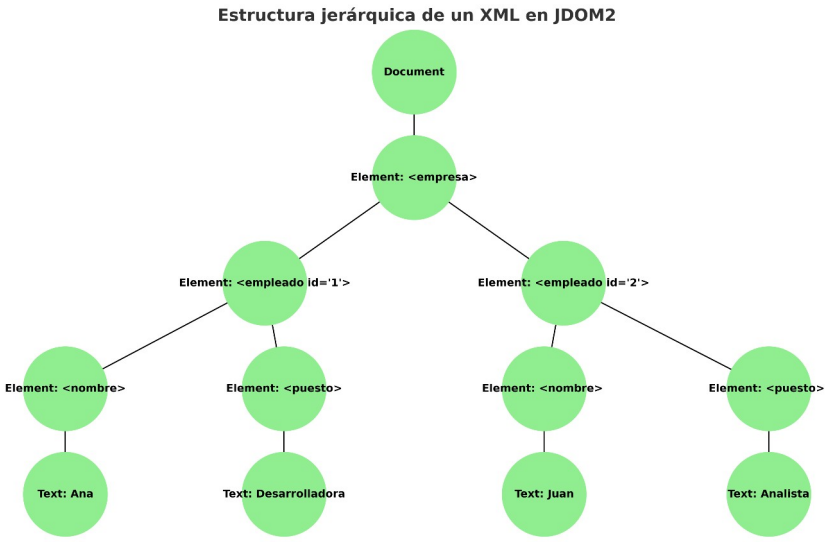
- **org.jom2**: Representa un documento XML y sus componentes:
  - ***Document***: Representa el documento XML completo.
  - ***Element***: Representa un elemento XML (etiquetas).
  - ***Attribute***: Representa un atributo de un elemento.
  - ***Text***: Contenido textual dentro de un elemento.
  - ***Comment***: Comentarios dentro del XML.
  - ***Namespace***: Manejo de espacios de nombres.
- **org.jdom2.input**: Proporciona clases que permiten parsear un fichero XML y transformarlo en un árbol de objetos JDOM. Su clase principal es ***SAXBuilder*** que construye un objeto ***Document*** a partir de un fichero XML.

```
SAXBuilder builder = new SAXBuilder();

Document doc = builder.build(new File("personas.xml"));

Element root = doc.getRootElement();
```

- **org.jdom2.output**: Proporciona clases para convertir un documento JDOM en una representación de salida, como un texto XML. Su clase principal es ***XMLOutputter*** que serializa un ***Document*** o ***Element*** a texto XML.



5.1. Creación de un documento XML

Lo primero que se debe hacer es crear el elemento raíz y a continuación cada uno de sus hijos. Para ello se debe usar la clase ***Element***. Esta clase proporciona los métodos para obtener y manipular los elementos del XML, acceder directamente al contenido textual del elemento, manipular sus atributos,…

Métodos útiles de ***Element*** son:

- ***Element("nombre")*** → constructor de la clase. Permite crear un elemento.
- ***getName()*** → devuelve el nombre del elemento.
- ***getChild("nombre")*** → devuelve solo el primer elemento nombre dentro del elemento actual.
- ***getChildText("nombre")*** → devuelve directamente el texto.
- ***getChildren("nombre")*** → devuelve una lista de elementos hijos.
- ***getAttributeValue("nombre")***→ devuelve el atributo indicado del elemento.
- ***setAttribute("nombre", valor)***→ crea un atributo en el elemento y le asigna un valor.
- ***addContent(Element e); addContent(e.setText(String))***→ Añade el elemento al final de la lista. También permite crear el elemento y asignarle texto.

Finalmente, se crea el documento asignándole el elemento raíz. Para ello se usará la clase ***Document*** y la clase ***XMLOutputter*** que mediante los métodos indicados a continuación, permite guardar el documento en un fichero XML:

- ***XMLOutputter***: Constructor que recibe como parámetro el formato de salida del XML. Se usa el método ***Format.getPrettyFormat()*** para generar el fichero XML con indentación y saltos de línea.

- ***output(Document doc, FileWriter file)***: Método que realiza la escritura del árbol de documento en el fichero indicado como parámetro.

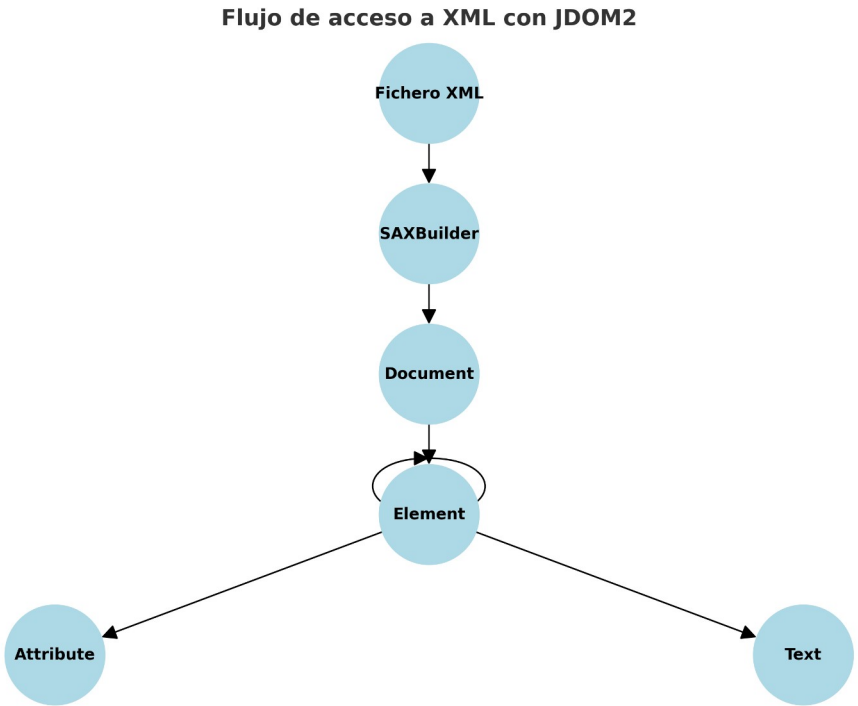
- ***outputString(Document)***: Permite obtener el árbol de documentos XML en formato ***String***.

Ver: CrearXMLPersonas.

5.2. Lectura de un documento XML

Para la lectura de un documento XML se usa la clase ***SAXBuilder***. El método ***build(File fichero)*** lee el fichero XML, analiza su contenido y crea un objeto ***Document***. Este objeto es la representación en memoria del fichero XML. Con el método ***getRootElement()*** se obtiene el elemento raíz que permitirá recorrer los elementos hijos, usando los métodos de la clase ***Element*** previamente indicados.

Ver: LeerXMLPersonas.



5.3. Añadir y eliminar datos de un documento XML

Para esta operación hay que proceder de forma muy similar a la lectura. Se usa la clase ***SAXBuilder***, se lee el fichero XML, se construye el objeto ***Document*** y se obtiene el elemento raíz. Después añadimos o eliminamos el elemento deseado.

Para añadir un nuevo nodo se usa el método anteriormente comentado ***addContent***.

En cambio, para eliminar un elemento, se usa el método ***removeContent*** del objeto ***Element*** indicando el elemento a eliminar. En este caso es **muy importante comprobar que realmente es el elemento que se quiere eliminar**.

Finalmente, una vez añadido y/o eliminado el elemento, se debe volver a escribir el árbol DOM en el fichero XML usando los correspondientes métodos de la clase ***XMLOutputter***.

Ver: InsertaYEliminaPersonas

En el caso de que se quiera eliminar varios elementos (por ejemplo, todas las personas que tengan 25 años), se debe usar un **iterador** ***(Iterator)*** para el recorrido de los nodos. Además, en lugar de utilizar el método ***removeContent*** hay que usar el método ***remove*** para eliminar los elementos que cumplan la condición establecida.

Ver: EliminarXMLPersonas

5.4. Modificar datos de un documento XML

Para modificar un elemento se usa el método ***getChild("nombreNodo")***. Este método devolverá el nodo a actualizar, procediendo a modificar:

- su contenido usando el método ***setText("cadena")***
- su atributo (o añadir un nuevo atributo) usando el método ***setAttribute("nombre", valor)***.

En el caso de querer modificar todos los nodos que tengan como texto un determinado valor, se debe usar el método ***getChildren("nombreNodo")***, procediendo a continuación de forma similar al caso anterior.

Ver: ModificarXMLPersonas

6. Trabajo con ficheros JSON

JSON (**JavaScript Object Notation – Notación de Objetos de JavaScript**) es un formato de texto ligero para intercambio de datos. Es ampliamente utilizado debido a su simplicidad y legibilidad, tanto para humanos como para máquinas. Además, es independiente del lenguaje de programación. Al igual que XML se usa para transmitir datos entre un cliente y un servidor y para almacenar configuraciones.

Sus características principales son:

- Formato basado en texto plano.
- Ligero.
- Fácil de leer y escribir.
- Estandarizado.
- Estructura jerárquica: permite representar datos estructurados y complejos (con objetos anidados).

Un texto JSON consta de:

- Claves: Cadenas escritas entre comillas dobles (“)

- Valores: Pueden ser:
  - Cadenas.
  - Números.
  - Booleanos: true o false.
  - Objetos: otra estructura JSON.
  - Arrays: lista de valores entre []
  - null: para indicar un valor vacío.

Ejemplo:

```
{
  "usuario": {
    "id": 101,
    "nombre": "Ana García",
    "email": "ana.garcia@example.com",
    "edad": 28,
    "activo": true,
    "roles": ["usuario", "editor"]
  },
  "configuracion": {
    "tema": "oscuro",
    "notificaciones": true,
    "idioma": "es"
  },
  "historial_login": [
    "2025-09-28T08:30:00",
    "2025-09-29T09:15:00",
    "2025-09-30T10:00:00"
  ]
}
```

Descripción del contenido del ejemplo mostrado:

1.- usuario

- Objeto que contiene información personal del usuario.
- Campos:
  - id: número identificador del usuario.
  - nombre: nombre completo.
  - email: dirección de correo electrónico.
  - edad: edad en años.
  - activo: indica si el usuario está activo (booleano).
  - roles: lista de roles que tiene el usuario.

2.- configuracion:

- Objeto con preferencias del usuario.
- Campos:
  - tema: el tema de la interfaz (por ejemplo, "oscuro" o "claro").
  - notificaciones: si están activadas las notificaciones (booleano).
  - idioma: idioma preferido (código de idioma, en este caso español "es").

3.- historial_login:

- Array de cadenas con las fechas y horas en las que el usuario inició sesión.

6.1. Acceso a ficheros JSON

Se puede realizar usando diversas librerías entre las que destacan: **Jackson** y **Gson** (proporcionada por Google).

En este caso se va a usar **Jackson** al ser una librería que facilita la conversión de objetos Java a JSON (serialización) y de JSON a objetos Java (deserialización). Para incluir estas librerías lo más correcto es usar ***gradle*** para que nos incluya las dependencias en el proyecto:

```
implementation group: 'com.fasterxml.jackson.core', name: 'jackson-databind', version: '2.18.2'
```

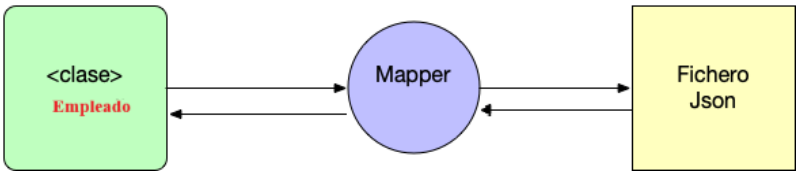
Una vez añadidas las dependencias, se debe crear la clase que se corresponda con cada objeto del fichero JSON. Por ejemplo, si el fichero JSON va a recoger objetos de tipo empleado cuya información es la mostrada a continuación:

```
{
	"id": 3,
	"apellido": "SEVILLA",
	"dep": 30,
	"salario": 2200.0
}
```

Lectura

Para proceder a su lectura, se deberá implementar una clase asociada a cada objeto JSON. Por ejemplo, se puede tener una clase ***Empleado*** cuyos atributos sean:

```
public class Empleado
{
        private int id;
        private String apellido;
        private int dep;
        private double salario;
….
}
```



Esta clase se usará para mapear cada entrada del fichero JSON a objetos de tipo **Empleado**.

Para realizar este mapeo se utiliza la clase ***ObjectMapper***, clase principal de la librería ***Jackson*** que permite mapear JSON a objetos Java y viceversa.

El contenido del fichero se puede mapear a:

1. Un objeto de tipo ***Map<String, Object>***
2. Un objeto (en nuestro ejemplo de tipo ***Empleado)***.
3. Una lista de objetos (en nuestro ejemplo una lista de Empleados).

En ambos casos se usa el método ***readValue(File, TypeReference<T> valueTypeRef)*** que va a permitir realizar la deserialización del contenido JSON de un fichero determinado a un tipo Java determinado.

El método ***readValue()*** puede lanzar las excepciones ***IOException***, ***StreamReadException*** y ***DatabindException***.

TypeReference<T>: especifica el tipo de dato Java que se espera obtener (por ejemplo **List<Empleado>, un objeto Empleado, un Mapa**))

Además, es necesario que la clase a deserializar incluya el **constructor sin parámetros estando el cuerpo del mismo vacío**. En caso de no incluirlo se generará un error similar al siguiente:

Cannot construct instance of `ficherosjson.Empleado` (no Creators, like default constructor, exist): cannot deserialize from Object value

Una vez leídos los valores, se pueden usar los respectivos métodos ***getter*** del objeto leído para acceder individualmente a cada uno de sus atributos.

Ver: LeerUnEmpleadoJSON.

Escritura

Para escribir desde cero en un fichero JSON, se ha de crear un ***ObjectMapper*** y usar el método ***writeValue(File, tipo)*** para escribir en el fichero el objeto Java deseado.

El objeto Java a escribir puede:

1. Estar almacenado en un Mapa
2. Ser un objeto individual
3. Estar almacenado en una lista.

El método ***writeValue*** puede generar las excepciones ***StreamWriteException***, ***DatabindException*** e ***IOException***.

Ver: EscribirObjetoEmpleado, EscribirUnJSON

En el caso de que se quiera generar un JSON con un formato indentado, añadiendo sangrías, saltos de línea y espacios en blanco, en lugar de un JSON con todo en una sola línea, se puede usar el método ***writerWithDefaultPrettyPrinter()*** del objeto ***ObjecMapper***.

mapper.writerWithDefaultPrettyPrinter().writeValue(jsonFile, empleado);

Modificar, añadir y eliminar datos

Para modificar, añadir o eliminar datos en un fichero JSON, se debe leer el fichero, se hacen los cambios en el objeto representado en Java, y finalmente, se vuelve a guardar en el fichero.

Ver: ModificarEmpleados

6.2. Clase JsonNode

Otra opción para el trabajo con ficheros JSON es usar la clase ***JsonNode*** que permite modificar el fichero JSON sin tener que mapear el JSON a clases POJO.

Por tanto, esta clase será muy útil cuando no se quiere (o no se puede) mapear directamente a clases Java, y se necesita recorrer o inspeccionar datos JSON dinámicamente.

Con ella puedes:

- Leer y navegar por estructuras JSON sin necesidad de crear clases Java específicas.
- Acceder a valores con métodos como ***.get()***, etc.
- Diferenciar tipos de nodos (objetos, arrays, números, booleanos…).

Se usará el método ***readTree(File)*** de la clase ***ObjectMapper*** para obtener el ***JsonNode***. De este modo se convierte el fichero JSON en un árbol de nodos JSON.

Ver: LeerEmpleadosJsonNode

JsonNode dispone de los métodos ***isObject()*** e ***isArray()*** para verificar si el JSON es un solo objeto o es una array de objetos, respectivamente.

Luego se usarán las clases:

- ***ObjectNode***: Si el JSON es un objeto (va {}) se usa el método ***put()*** para añadir o modificar un campo clave. Si se intenta modificar un campo que no existe, se creará automáticamente.

  Esta clase también consta del método ***remove("clave")*** para eliminar un campo del fichero. Por ejemplo si se quiere eliminar el campo edad, se escribiría:

  objectNode.remove("edad");

  Por otro lado, si se quiere leer el valor de un campo, simplemente se usa el método ***get()*** pasando como parámetro el campo clave seguido del tipo de datos al que se desea convertir: ***asText, asInt, asBoolean***,...

  objectNode.get("ciudad").asText;

- ***ArrayNode***: si el JSON es un array ([]) y el método ***add()*** para agregar un nuevo objeto al array.

Añadir campos a un objeto

Para esto se lee el fichero JSON usando ***ObjectMapper*** y mediante el método ***readTree*** se obtiene el árbol de nodos JSON. A continuación se debe consultar si se trata de un objeto mediante el método ***isObject***. En caso afirmativo, el ***JsonNode*** se convierte a un ***ObjectNode*** sobre el que se ejecuta el método ***put()*** para añadir todos los campos que se deseeen.

Por ejemplo, si se quiere añadir dos campos ("edad":35 y "ciudad":Almería) al fichero persona.json,

```
{
 "id" : 1,
 "nombre" : "Ana"
}
```

se usaría el método ***put*** de un objeto ***ObjectNode***. El resultado después de añadir ambos campos sería:

```
{
 "id" : 1,
 "nombre" : "Ana",
 "edad": 35,
 "ciudad": "Almería"
}
```

Por último, se escribe el JSON modificado al fichero usando el método ***writeValue()***.

Ver: AgregarCampoJsonNode

Añadir campos a un array de objeto

Para esto se lee el fichero JSON usando ***ObjectMapper*** y mediante el método ***readTree*** se obtiene el árbol de nodos JSON. A continuación se debe consultar si se trata de un array mediante el método ***isArray***. En caso afirmativo, el ***JsonNode*** se convierte a un ***ArrayNode*** que habrá que recorrer hasta encontrar el elemento al que se desea añadir el campo.

Una vez encontrado se convierte a ***ObjectNode*** para poder modificarlo, usando el método ***put()*** para añadir el nuevo campo.

Por último, se escribe el JSON modificado al fichero usando el método ***writeValue()***.

Ver: AgregarCampoArrayJsonNode

Si se quiere añadir el mismo campo a todos los elementos del array, simplemente se eliminaría la condición dentro del bucle que recorre el array.

Eliminar campos de un objeto

Para esto se lee el fichero JSON usando ***ObjectMapper*** y mediante el método ***readTree*** se obtiene el árbol de nodos JSON. A continuación se debe consultar si se trata de un objeto mediante el método ***isObject***. En caso afirmativo, el ***JsonNode*** se convierte a un ***ObjectNode*** sobre el que se ejecuta el método ***remove()*** para eliminar todos los campos que se deseeen.

Por último, se escribe el JSON modificado al fichero usando el método ***writeValue()***.

Ver: EliminarCampoJsonNode

Otros ejemplos

Para agregar un nuevo elemento se usará el método ***createObjectNode()*** del objeto ***ObjectMapper*** para crear un nuevo nodo de objeto JSON. Con el método ***put()*** asignamos los pares clave-valor y por último se añade el objeto al ***ArrayNode*** y el método ***add()*** para agregar un nuevo objeto.

Ver: AgregarElementoArrayJsonNode; ModificarSalarioEmpleadosJsonNode; ModificarEliminarEmpleadosJsonNode;