UNIDAD DIDÁCTICA 1

MANEJO DE FICHEROS

MÓDULO PROFESIONAL:
ACCESO A DATOS

CESUR
Tu Centro Oficial de FP

Índice

RESUMEN INTRODUCTORIO

En esta unidad introduciremos los conceptos básicos sobre la lectura de ficheros y los diferentes tipos y métodos que tenemos para poder trabajar con ellos dentro de una aplicación informática.

En concreto, estudiaremos las diferentes clases que nos encontramos dentro del lenguaje Java para poder trabajar con ficheros, la relación entre el tipo de fichero y, por lo tanto, de la clase a emplear.

Al final de la unidad también aprenderemos la potencialidad de los ficheros XML y cuáles son los estándares actuales y ejemplos de trabajo desde Java.

INTRODUCCIÓN

En informática, a partir del objetivo de tratar de forma automatizada la información, surge una necesidad: hacer que los datos permanezcan más allá de la ejecución del proceso o la aplicación que los ha creado. Las aplicaciones tienen que poder guardar y recuperar datos y, por tanto, los datos tienen que ser persistentes.

El primer tipo y formato que nos permite conseguir este objetivo es el de los ficheros, objetos sencillos de manejar y sobre los cuales se pueden realizar las operaciones básicas de lectura y escritura de almacenaje de información.

Hasta hace bien poco, este mecanismo era utilizado por más de una aplicación para el almacenamiento incluso de información estructurada. Aunque con la evolución de los sistemas de gestión de bases de datos, los ficheros han sido relegados a almacenamientos de información secuencial en su gran mayoría.

En este sentido, dentro de Java, nos encontramos con diversos paquetes y librerías que nos permiten justamente realizar estas operaciones básicas sobre ficheros.

CASO INTRODUCTORIO

En la empresa que hemos sido contratados, tenemos que desarrollar una aplicación de escritorio que edite textos adaptado a las necesidades de la empresa, ya que el formato y las particularidades son especiales.

Ya has realizado toda la programación de la interfaz y funciones de escritura. Ahora, deberás implementar una serie de funciones que le sirvan para guardar el documento, así como para recuperar la información que haya guardado previamente.

Deberás tener en cuenta las clases existentes en Java para manejar ficheros atendiendo a las funciones habituales o incorporar librerías externas.

Al finalizar la unidad, serás capaz de trabajar con datos provenientes de distintos tipos de archivos, conocer las clases Java que implementan los flujos de entrada y salida y trabajar con ficheros de acceso secuencial y aleatorio.

1. CLASES ASOCIADAS A LAS OPERACIONES DE GESTIÓN DE FICHEROS Y DIRECTORIOS

Vas a comenzar tu labor, por lo que debes plantearte cuál será la clase base de inicio dentro de Java, si es independiente del sistema operativo, si es necesario incluir librerías externas y cómo funcionan las clases necesarias para las operaciones básicas sobre ficheros y directorios.

En los sistemas informáticos actuales cualquier dispositivo puede llegar a tener millones de ficheros si tenemos en cuenta al sistema operativo y a las aplicaciones instaladas. La gestión de estos ficheros, su localización y utilización por lo tanto resulta imprescindible.

Los sistemas operativos cuentan con un sistema de gestión de ficheros que, independientemente de que cada uno de ellos sea diferente, se encuentra jerarquizado para que se puedan alcanzar, en primer lugar, los directorios y, en segundo lugar, los ficheros dentro de esos directorios.

Además, si tenemos en cuenta que los dispositivos de almacenamiento han ido evolucionando y nos encontramos con dispositivos externos, dispositivos en red y almacenamiento en la nube, los sistemas operativos han implementado mecanismos de unificación de gestión de ficheros independientemente de su posición, con un acceso unificado.

Un ejemplo extraído del sistema operativo Linux de unificación de estrategias es el siguiente:

```
/direccion/donde/esta/el/fichero.txt
```

En el caso del sistema operativo Windows, la estrategia es bien diferente, manteniendo diferenciados los diferentes sistemas y dispositivos para el acceso, teniendo denominaciones específicas:

```
F:\direccion\donde\se\encuentra\el\fichero.txt
\servidor\donde\esta\el\fichero.txt
```

1.1 La clase File

En el lenguaje de programación Java se utiliza la clase File para gestionar archivos y directorios. Esta clase permite crear y eliminar archivos o directorios, al igual que obtener información de ellos como, por ejemplo, el tamaño de un archivo, los archivos que contiene un directorio, etc. La clase File se encuentra en el paquete java.io.

ENLACE DE INTERÉS

Profundiza sobre esta clase en la documentación oficial de Java:

PARA SABER MÁS

Hay que tener en cuenta que, si se trabaja con OpenJDK, se tiene compatibilidad con la clase File, además de tener otros paquetes y librerías interesantes:

ENLACE DE INTERÉS

Aquí tienes otro ejemplo de dicha compatibilidad:

Para crear un objeto de dicha clase se utiliza el siguiente constructor:

```
public File(String path);
```

Donde *path* indica la dirección absoluta o relativa al directorio actual. Son muchos los métodos que proporciona esta clase. Entre los principales se podrían destacar los siguientes:

MÉTODO	DESCRIPCIÓN
boolean canRead()	Devuelve *true* si se puede leer el fichero; en caso negativo, devuelve *false*.
boolean canWrite()	Devuelve *true* si se puede escribir en el fichero; en caso negativo, devuelve *false*.
boolean createNewFile()	Crea un fichero. Devuelve *true* si se ha podido crear; en caso negativo, devuelve *false*.
boolean delete()	Elimina el fichero o directorio. En el caso de ser un directorio lo que queremos eliminar, este debe estar vacío. Devuelve *true* si se ha podido eliminar; en caso negativo, devuelve *false*.
boolean exists()	Devuelve *true* si el fichero o directorio existe; en caso negativo, devuelve *false*.
String getName()	Devuelve el nombre del fichero o directorio.
String getAbsolutePath()	Devuelve la ruta absoluta del fichero o directorio.

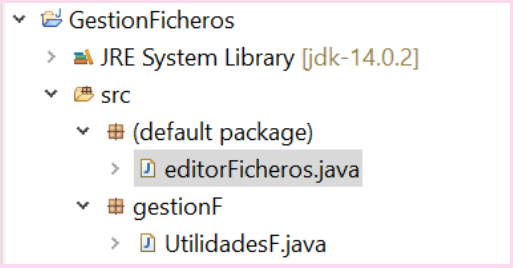
String getCanonicalPath()	Devuelve la ruta única absoluta del fichero o directorio.
String getPath()	Devuelve la ruta con la que se creó el fichero o directorio.
String getParent()	Devuelve el directorio padre del fichero o directorio. En caso de no tener directorio padre, devuelve *null*.
boolean isAbsolute()	Devuelve *true* si es una ruta absoluta; en caso negativo, devuelve *false*.
boolean isDirectory()	Devuelve *true* si es un directorio válido; en caso negativo, devuelve *false*.
boolean isFile()	Devuelve *true* si es un fichero válido; en caso negativo, devuelve *false*.
long lastModified()	Devuelve en milisegundos la última vez que se modificó el fichero. Si el fichero no existe, devuelve 0.
long length()	Devuelve el tamaño en bytes del fichero. Si el fichero no existe, devuelve 0.
String[] list()	Devuelve una lista con todos los ficheros y directorios del directorio indicado.
String[] list(FilenameFilter filtro)	Devuelve una lista con todos los ficheros y directorios del directorio indicado que cumplen con el filtro indicado.
boolean mkdir()	Crea un directorio. Devuelve *true* si se ha podido crear el directorio; en caso negativo, devuelve *false*.
Boolean renameTo(File file)	Renombra el fichero indicado en el parámetro *file*. Devuelve *true* si se ha podido realizar el cambio; en caso negativo, devuelve *false*.

EJEMPLO PRÁCTICO

En el editor de texto que estamos realizando pretendemos incorporar diferentes funcionalidades que proporcionen una interfaz para el manejo de ficheros y directorios.

Para comenzar nuestro proyecto queremos implementar una clase y una utilidad que permita listar el contenido sobre el directorio actual.

1. El primer paso será crear un nuevo proyecto con la siguiente estructura y ficheros.



2. UtilidadesF tendrá las propiedades y métodos para la gestión de ficheros. En concreto, implementamos un primer método para el listado del directorio actual.

```
package gestionF;

import java.io.File;

public class UtilidadesF {

	public void directorioActual() {
		File directorio = new File(".");  //directorio actual
		String[] lista = directorio.list();
		for (int i = 0; i<lista.length; i++)
		{
			System.out.println(lista[i]);
		}
	}

}
```

3. La clase editorFicheros será la clase principal.

```
import gestionF.UtilidadesF;

public class editorFicheros {
	static UtilidadesF utilF=new UtilidadesF();

	public static void main(String[] args) {
		// listamos el directorio actual
```

```
                utilF.directorioActual();
        }

}
```

1.2 Flujos. Flujos basados en bytes y flujos basados en caracteres

Con bastante frecuencia se da el caso de que un programa necesita obtener información de un origen o enviar información a un destino, que no necesariamente tienen por qué tratarse de archivos. Por ejemplo, capturar información desde el teclado o mostrar información en pantalla. Esta comunicación entre el origen de la información y el destino se realiza mediante lo que se conoce como flujo (stream) de información.

Un flujo es un objeto que hace de **intermediario** entre un programa y la fuente o el destino de la información. Esto permite que el programa pueda leer o escribir información en el flujo sin importarle el origen o destino de la información ni el tipo de datos. Esto es lo que se conoce como abstracción, y permite que la programación sea un proceso más sencillo, ya que no es necesario conocer nada del tipo de información (puesto que los datos pueden ser bytes, tipos primitivos, etc.) ni del dispositivo (ya que un flujo puede venir o dirigirse, por ejemplo, a un archivo en disco, otro programa, etc.).

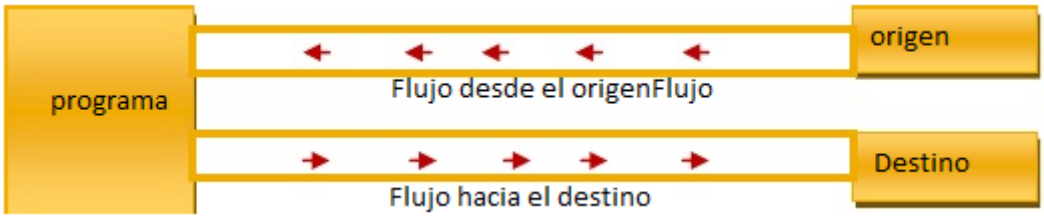
Los algoritmos para leer y escribir datos siguen generalmente el mismo esquema.

Para **leer**, sería:

- Abrir un flujo desde un origen.
- Mientras haya información, leer información.
- Cerrar el flujo.

Y para **escribir**:

- Abrir un flujo hacia un destino.
- Mientras haya información, escribir información.
- Cerrar el flujo.



Comunicaciones con flujos de datos

Se distinguen **dos tipos básicos** de flujos de entrada y salida:

- Flujos de **bytes**.
- Flujos de **caracteres**.

Se recomienda el uso de los flujos de bytes solo para las operaciones más elementales de entrada y salida. En cualquier otro caso es recomendable utilizar los flujos más adecuados según los tipos de datos que hay que manejar.

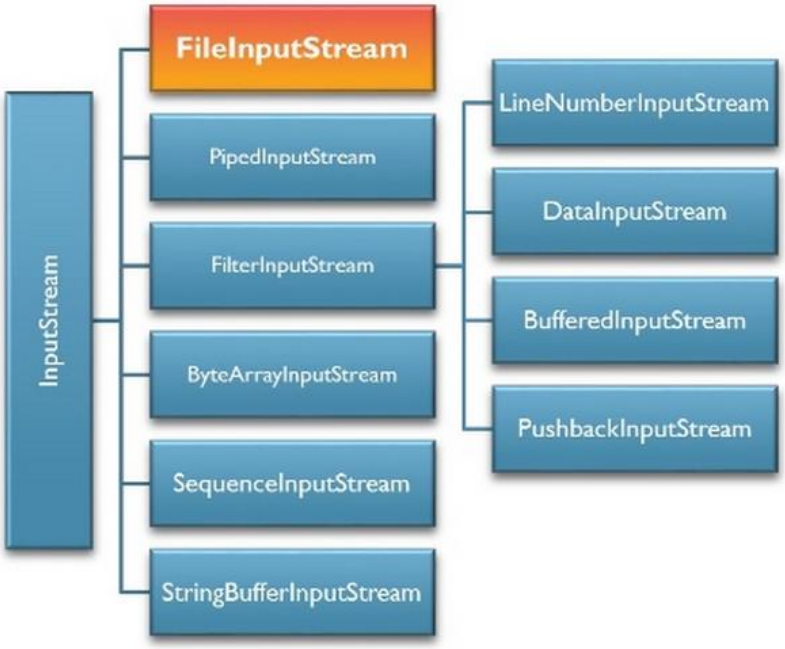
PARA SABER MÁS

Visita esta página web para ampliar la información sobre los flujos y algunas clases utilizadas:

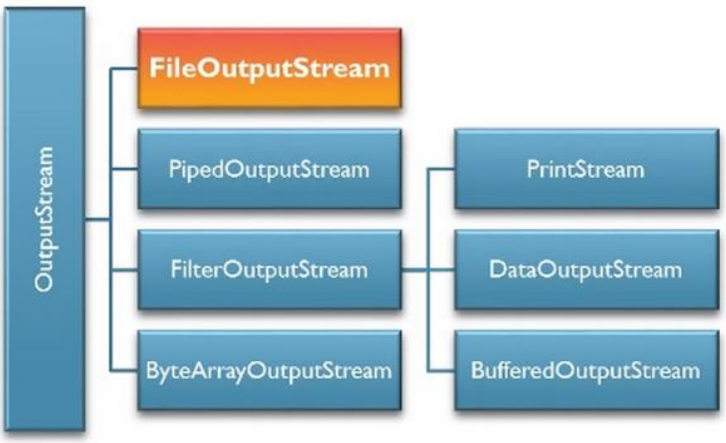
1.3 Flujos de bytes

Los flujos de bytes son un tipo especial de flujo que se encargan de realizar operaciones de entrada y salida en base a bytes de 8 bits. Por tanto, su uso se destina a la lectura y escritura de datos binarios.

Las clases InputStream y OutputStream son las clases padre del resto de clases de flujos de bytes. En concreto, dentro de estas clases hijas, se pueden destacar las clases FileInputStream y FileOutputStream, que se encargan de manipular flujos de bytes que proceden de archivos en disco o se dirigen hacia ellos, respectivamente.



Jerarquía de clases dentro de InputStream



Jerarquía de clases dentro de OutputStream

ENLACE DE INTERÉS

Aquí podrás obtener información más detallada sobre InputStream:

Para usar uno de estos flujos, lo primero será crear un objeto. Para ello, se usa su constructor. Por ejemplo, para leer un fichero que se encuentra en la ruta *path*:

```
in = new FileInputStream(String path);
```

La forma de trabajar con estos flujos es habitualmente mediante un bucle que recorre todo su contenido. La función usada en este caso para leer el contenido es read(), que devuelve un entero, cuyo valor será el del siguiente byte leído del archivo o, en caso de haber alcanzado el final, el valor -1. Es decir, la primera vez que se ejecuta, devolverá el primer byte del archivo, la siguiente el segundo, etc., así hasta que devuelva -1, en cuyo caso, se puede deducir que se ha alcanzado el final del archivo. Su funcionamiento, por tanto, es secuencial.

De esta manera, el bucle básico para leer un archivo sería:

```
while (int b = in.read() != -1) {
    System.out.print((char) b); //Imprimimos el byte como carácter
}
```

Donde en cada iteración, b contendrá el valor de cada uno de los bytes leídos del archivo, empezando por el primero y hasta alcanzar el último.

Mantener flujos abiertos implica un gasto de recursos, por lo que deben cerrarse estos flujos en cuanto dejen de usarse, para evitar malgastar recursos. El programa anterior cierra los flujos en el bloque *finally*, verificando previamente que los flujos archivos fueron efectivamente creados (sus referencias no son *null*).

EJEMPLO PRÁCTICO

Insertar texto aquí.

En el editor de texto que estamos realizando y dentro de las utilidades a incorporar necesitamos tener un método que nos permita importar datos de ficheros específicos con extensión .rpc que provienen de una exportación de una máquina de producción a nuestro fichero actualmente activo.
¿Cómo realizaríamos este proceso?

1. Necesitaremos crear un nuevo método dentro de nuestra clase de utilidades que tenga la siguiente interfaz:

```
public void importarRPC(String ficheroActual,String ficheroRPC)
```

2. En este método incluiremos el siguiente código:

```
    public void importarRPC(String ficheroActual,String ficheroRPC)
throws IOException {
            FileInputStream in=null;
            FileOutputStream out=null;

            try {
                  in= new FileInputStream(ficheroRPC);
                  out= new FileOutputStream(ficheroActual);
                  int b;
                  while((b=in.read())!=-1) {
                        out.write(b); //Escribe el byte
                  }

            }finally {
                  if(in!=null) in.close();
                  if(out!=null) out.close();
            }

    }
```

VÍDEO DE INTERÉS

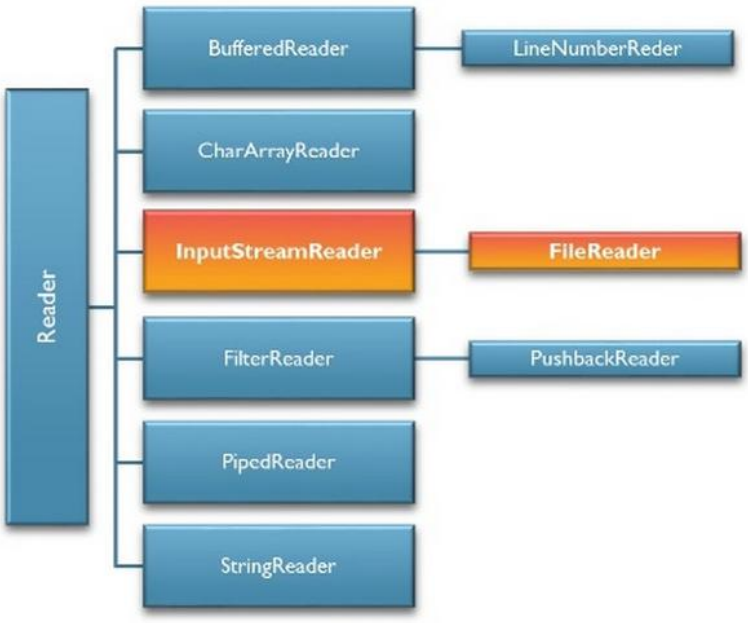
Visualiza un ejemplo de uso de FileInputStream y FileOutputStream:

1.4 Flujos de caracteres

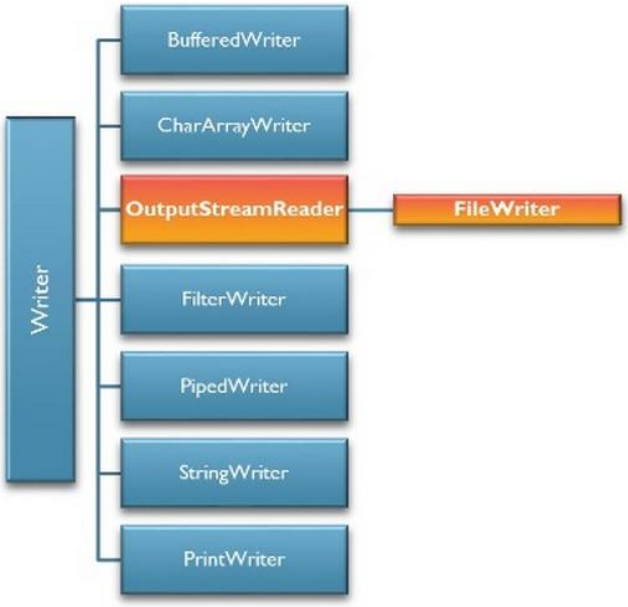
Como se indica en la introducción de este apartado, es recomendable utilizar los flujos adecuados a cada tipo de datos.

La codificación usada por los flujos de caracteres es Unicode, por lo que se pueden utilizar internacionalmente.

Estos flujos se encuentran definidos en las clases **FileReader** y **FileWriter**, las cuales descienden de las clases **Reader** y **Writer**, y se utilizan para la lectura y escritura de caracteres en archivos.



Jerarquía de clases dentro de InputStreamReader



Jerarquía de clases dentro de OutStreamReader

EJEMPLO PRÁCTICO

En el editor de texto que estamos realizando tenemos implementada una utilidad que importa ficheros en modo bytes, de forma rápida y sencilla para un tipo de ficheros específicos muy pequeños. Pero para ficheros únicamente de texto y más grandes, ficheros con extensión .txt, necesitamos implementar una funcionalidad que nos permita realizar incluso correcciones futuras.

¿Cómo realizaríamos este proceso?

1. Necesitaremos crear un nuevo método dentro de nuestra clase de utilidades que tenga la siguiente interfaz:

```
public void importarTXT(String ficheroActual,String ficheroRPC)
```

2. En este método incluiremos el siguiente código:

```
public void importarTXT(String ficheroActual,String ficheroRPC)
throws IOException
{
            FileReader in=null;
      FileWriter out=null;
      try //crea flujos de entrada y salida
      {
            in=new FileReader(ficheroActual);
            out=new FileWriter(ficheroRPC);
            int c; //guarda cada byte en una variable tipo int
            while((c=in.read())!=-1) //lee un byte en c
            {
                  out.write(c);     //escribe el carácter
            }
      }
      finally     //Cierra los flujos
      {
            if(in!=null) in.close();
                        if(out!=null) out.close();
      }
      }
```

VÍDEO DE INTERÉS

Visualiza del minuto 13 al 25 para conocer el acceso a dicheros en la programación en Java:

1.5 Formas de acceso y operaciones básicas

A la hora de almacenar información en un fichero, es importante seguir una estructura determinada que permita recuperar posteriormente esos datos de forma adecuada. El caso más común es que un fichero esté constituido por un conjunto de registros (partes), que normalmente son del mismo tamaño.

Hasta el momento, se han presentado funciones que acceden a dichos contenidos secuencialmente. Sin embargo, aunque ello resulta en un acceso rápido a todo el contenido del archivo, lo cual es muy útil para, por ejemplo, copiar un archivo; hay veces en las que no es la opción más adecuada. En numerosas ocasiones, interesa leer solamente una parte del archivo y no el archivo al completo, por lo que leer todo secuencialmente hasta alcanzar la parte que interesa, resulta muy lento. Para ello, se utiliza el llamado acceso aleatorio.

En función del tipo de acceso que se haga al archivo, existirán una serie de operaciones posibles para su manejo.

1.5.1 Formas de acceso a un fichero

Las principales formas de acceso a un fichero son secuencial y aleatoria:

- **Acceso secuencial.** En los ficheros de acceso secuencial, los datos o registros que lo forman se leen y se escriben en un orden estricto. Por ejemplo, si se desea acceder a un registro, es necesario leer todos y cada uno de los registros anteriores. Otra característica importante es que no es posible realizar inserciones de datos o registros en medio del fichero, sino que se realizan al final de este.

  Para trabajar con ficheros de acceso secuencial se utilizan las clases FileReader y FileWriter, si es para datos de texto, y FileInputStream y FileOutputStream, si es para datos binarios.

- **Acceso aleatorio.** Un fichero de acceso aleatorio es aquel que permite leer o escribir datos en forma no secuencial, es decir, en cualquier orden.

  Para trabajar con ficheros de acceso aleatorio se utiliza la clase java.io.RandomAccessFile, que implementa las interfaces DataInput y DataOutput, las cuales permiten leer y escribir en el fichero.

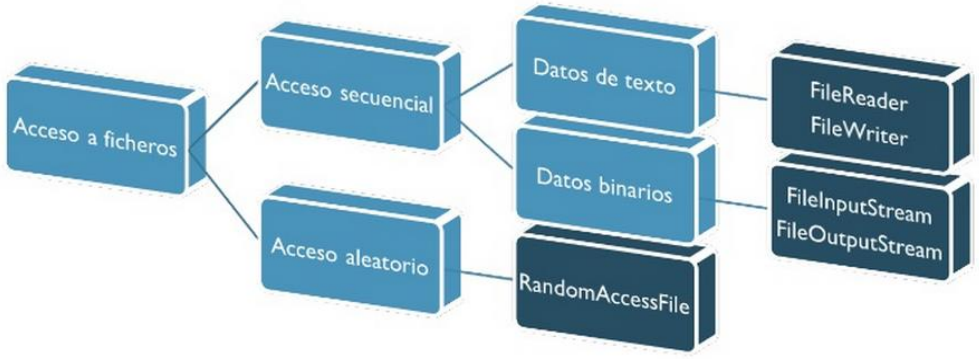
  Para trabajar con la clase RandomAccessFile se debe indicar el nombre del archivo que se quiere abrir o crear, así como si se pretende abrir para lectura o también para escritura.

  Por ejemplo, para abrir un archivo para lectura, se pondría:

```
RandomAccessFile f1 = new RandomAccessFile("archivo.txt","r");
```

  Mientras que, para lectura y escritura, sería:

```
RandomAccessFile f2 = new RandomAccessFile("archivo.txt","rw");
```



Tipos de accesos y clases en Java

EJEMPLO PRÁCTICO

En la empresa en la que estamos desarrollando el editor de texto, tienen la necesidad de almacenar el número de productos que entran por almacén a través de un dispositivo que permite ejecutar Java.

El usuario introducirá un número entero por teclado y lo añadirá al final de un fichero binario recepcion.dat que contiene las cantidades. Necesitamos generar los métodos y programas necesarios para realizar la gestión.

En este caso la aplicación será muy sencilla e incluida totalmente en un único paquete que se encargará de realizar todo el proceso:

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.Scanner;

public class EntradaMercancia {

    static Scanner scanner = new Scanner(System.in);
    static RandomAccessFile fichero = null;

    public static void main(String[] args) {
        int numero;
        try {
            //fichero para el almacenaje de las mercancías
            fichero = new RandomAccessFile("mercancia.dat", "rw");
            mostrarFichero(); //muestra el contenido original del
fichero
            System.out.print("Introduce las unidades recibidas: ");
            numero = scanner.nextInt();
            fichero.seek(fichero.length()); //nos colocamos al final
del fichero
```

```
            fichero.writeInt(numero);       //se escribe el entero
            mostrarFichero();//muestra el contenido del fichero
después de añadir el número

        } catch (FileNotFoundException ex) {
            System.out.println(ex.getMessage());
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        } finally {
            try {
                if (fichero != null) {
                    fichero.close();
                }
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }

    public static void mostrarFichero() {
        int n;
        try {
            fichero.seek(0); //principio del fichero
            while (true) {
                n = fichero.readInt();  //leemos un entero
                System.out.println(n);  //lo mostramos
            }
        } catch (EOFException e) {
            System.out.println("Fin de fichero");
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

VÍDEO DE INTERÉS

Visualiza este recurso para conocer la entrada/salida en Java:

1.5.2 Operaciones básicas sobre ficheros de acceso secuencial y aleatorio

Las operaciones básicas que se pueden realizar sobre un fichero, sea cual sea su forma de acceso son:

- Creación del fichero.
- Apertura del fichero.
- Cierre del fichero.
- Lectura de datos del fichero.
- Escritura de datos en el fichero.

Además, en el caso de los ficheros de **acceso secuencial** se podrían llevar a cabo las siguientes operaciones:

- **Consulta de datos del fichero.** Para realizar consultas en un fichero de acceso secuencial es necesario empezar desde el primer registro e ir recorriendo cada uno de los registros del fichero hasta encontrar el deseado.

- **Añadir datos al fichero.** En el caso de los ficheros de acceso secuencial, solo se permite la inserción de datos o registros al final de estos.

- **Eliminar datos del fichero.** Para eliminar datos o registros de un fichero de acceso secuencial es necesario utilizar un fichero auxiliar para ir volcando todos los registros excepto el que queremos eliminar.

Estas mismas operaciones se pueden realizar sobre un fichero de **acceso aleatorio**, pero el mecanismo varía:

- **Consulta de datos del fichero.** Realizar consultas en un fichero de acceso aleatorio es muy fácil, basta con conocer la clave del registro que se esté buscando y aplicar una función que devuelva la dirección del registro asociado a esa clave, es decir, el punto del archivo en el que se encuentran los datos buscados.

- **Añadir datos al fichero.** En este caso, tan solo es necesario obtener la clave del registro que hay que insertar. A esta clave se le aplica la función de cálculo de dirección y se obtiene el punto del archivo en el que insertar el nuevo registro.

- **Eliminar datos del fichero.** Para realizar eliminaciones se suele utilizar un campo del registro que si se pone a cero indica que el registro debe eliminarse, aunque

físicamente no desaparece. No obstante, también se puede eliminar físicamente si interesa.

PARA SABER MÁS

Consulta el enlace para conocer cómo trabajar con ficheros secuenciales y aleatorios en C++.

1.6 Clases para gestión de flujos de datos desde/hacia ficheros

A continuación, se estudiarán las distintas clases para la gestión de flujos de datos desde o hacia ficheros, en función del tipo de ficheros con los que se esté trabajando.

Tipos de ficheros
Fuente: elaboración propia

1.6.1 Ficheros de texto

Para la gestión de ficheros de texto se utilizan las clases:

- **FileReader:** permite leer caracteres de un fichero de modo secuencial.
- **FileWriter:** permite escribir caracteres en un fichero de modo secuencial.
- **PrintWriter:** permite escribir caracteres en un fichero dándoles un determinado formato.

1.6.2 Ficheros binarios

Para la gestión de ficheros binarios se utilizan las clases:

- **FileInputStream:** permite leer los bytes de un fichero de modo secuencial.
- **FileOutputStream:** permite escribir bytes en un fichero de modo secuencial.
- **DataInputStream y DataOutputStream:** permiten leer y escribir datos primitivos en un fichero. Por ejemplo, enteros, flotantes, etc.
  Además de los métodos para leer y escribir, read() y write() respectivamente, tienen muchos otros métodos:

MÉTODOS PARA LECTURA	MÉTODOS PARA ESCRITURA
boolean readBoolean();	void writeBoolean(boolean v);
byte readByte();	void writeByte(int v);
int readUnsignedByte();	void writeUnsignedByte(String s);
int readUnsingedShort();	void writeShort(int v);
short readShort();	void writeChars(String s);
char readChar();	void writeChar(int v);
int readInt();	void writeInt(int v);
long readLong();	void writeLong(long v);
float readFloat();	void writeFloat(float v);
double readDouble():	void writeDouble(doublé v)
String readUTF();	void writeUTF(String str);

Métodos de lectura y escritura

1.6.3 Ficheros de objetos. Serialización

Para la gestión de ficheros de objetos se utilizan las clases **ObjectInputStream** y **ObjectOutputStream**, las cuales implementan las interfaces ObjectInput y ObjectOutput, subinterfaces de DataInput y DataOutput.

Muchas de las clases estándar soportan lo que se conoce como **serialización de objetos**, que permite guardar un objeto en un fichero escribiendo sus datos en un flujo de bytes. De esta forma, se puede escribir el objeto de una vez, sin necesidad de ir guardando sus atributos uno a uno, e igualmente para recuperarlo.

También es posible conseguir esto con clases definidas por uno mismo. Para realizarlo, es necesario implementar la interfaz Serializable. Para lo cual, se debe incluir, junto a la definición de la clase, "implements Serializable". Todos los atributos que se incluyan en dicha clase deben ser también serializables. De esta forma, se podrá hacer uso de las clases ObjectInputStream y ObjectOutputStream para leer y escribir objetos, respectivamente.

El uso de estas clases puede ser a través de una clase implementada exprofeso, implementando Serializable, o simplemente usando objetos serializables de Java como en el siguiente código ejemplo donde un programa genera una factura usando objetos BigDecimal para los precios y un objeto Calendar para la fecha. Si el método readObject() no devuelve el tipo correcto, el casting puede lanzar la excepción ClassNotFoundException, lo cual es notificado en el método main() mediante la cláusula throws.

```
public class FlujoDeObjetos
{
      static final String archDatos=”Factura.txt”;
static final BigDecimal[] precios={ new BigDecimal(“18.00”),
new BigDecimal(“160.00”), new BigDecimal(“25.00”),
new BigDecimal(“14.00”), new BigDecimal(“2.50”)};
static final int[] cants = {4,2,1,5,50};
static final String[] items = {"Marcadorazul", "Papel A4 500
hojas","Borrador", "CDROM RW", "Sobres A4 transparentes"};

Public static void main (String[] args) throws IOException,
ClassNotFoundException
{
      ObjectOutputStream out=null;
      try
{
out=new ObjectOutputStream(new BufferedOutputStream(new File
OutputStream(archDatos)));
out.writeObject(Calendar.getInstance());
for(int i=0; i<precios.length; i++)
```

```
{
      out.writeObject(precios[i]);
      out.writeInt(cants[i]);
out.writeUTF(items[i]);
}
      }
      finally
      {
            out.close();
      }
ObjectInputStream in=null;
      try
{
in=new ObjectInputStream(new BufferedInputStream(new
FileInputStream(archDatos)));
      Calendar fecha=null;
      BigDecimal precio;
      int cant;
      String item;
      BigDecimal total=new BigDecimal("0");
fecha=(Calendar) in.readObject();
System.out.format("El %Ta, %<tB, %<te, %<Ty, se compro: %n", fecha);
try
{
      while(true)
      {
            precio=(BigDecimal) in.readObject();
            cant=in.readInt();
            item=in.readUTF();
System.out.format("%4d %25s a $%6.2f c/u $%8.2f%n", cant,
item, precio, precio.multiply(new BigDecimal(cant)));
total=total.add(precio.multiply(new BigDecimal(cant)));
            }
      }
      catch(EOFException e)
      {
            System.out.format("\t\t\t\t\t TOTAL $%8.2f%n", total);
      }}
      finally
      {
            in.close();
      }
}
}
}
```

PARA SABER MÁS

Realiza la siguiente lectura para ampliar información sobre la serialización de objetos en Java:

EJEMPLO PRÁCTICO

En la empresa en la que estamos desarrollando diversas aplicaciones que mejoren la digitalización de las diferentes tareas se pretende mejorar el almacenamiento del número de productos que entran por el almacén a través de un dispositivo añadiéndole el nombre del artículo.

El dispositivo tiene capacidad de conexión Wifi y, posteriormente, enviará serializada la lista al servidor central.

El proceso sería el siguiente:

1. Se usa una clase Lista definida como serializable, esto permitirá guardar y recuperar la lista completa de un archivo, mediante una simple llamada a una función.

```
public class ListaProductos implements java.io.Serializable {
private int[] cantidadProductos;
private String[] listaProductos;
private int numMax,contadorProducto;


public ListaProductos (int productosAlmacenar) {
this.listaProductos = new String[productosAlmacenar];
this.cantidadProductos = new int[productosAlmacenar];
numMax= productosAlmacenar;
contadorProducto=1;
}

public int  nuevoProducto(String producto, int cantidad){
if(contadorProducto< numMax){
  this.cantidadProductos[contadorProducto-1] = cantidad;
  this.listaProductos [contadorProducto-1] = producto;
  contadorProducto++;
```

```
  return cantidad;
}else{
  return -1;
}

}

}
```

2. Cualquier programa podría hacer uso de esta clase:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;
import java.util.Scanner;

public class Main
{
public static void main(String[] args)
throws IOException, ClassNotFoundException
{
ListaProductos lista=new ListaProductos (5);
Scanner scanner=new Scanner(System.in);
System.out.println("Introduce las unidades recibidas(-1 para
finalizar): ");
int numero = scanner.nextInt();
while (numero>=0){
 scanner.nextLine();
 System.out.println ("Introduce el nombre del producto: ");
  String producto = scanner.nextLine();
  lista.nuevoProducto(producto, numero);
  System.out.println("Introduce las unidades recibidas(-1 para
finalizar): ");
  numero = scanner.nextInt();
}
ObjectOutputStream salida=new ObjectOutputStream(new
FileOutputStream("media.obj"));
salida.writeObject("guardar este string y un objeto\n");
salida.writeObject(lista);
salida.close();

ObjectInputStream entrada=new ObjectInputStream(new
FileInputStream("media.obj"));
String str =(String)entrada.readObject();
ListaProductos obj1=( ListaProductos)entrada.readObject();
entrada.close();
scanner.close();
}
}
```

1.7 Excepciones, detección y tratamiento

Cuando se está ejecutando un programa y este falla, se genera un error en forma de excepción, es decir, una excepción es un error que se ha detectado durante la ejecución del programa. Por ejemplo, un fallo habitual que se puede producir es un intento de división por cero, en cuyo caso se notifica de que se ha producido dicho error a través de la excepción correspondiente. Una vez que el programa reciba esa excepción, podrá realizar diversas acciones como, por ejemplo, notificar al usuario.

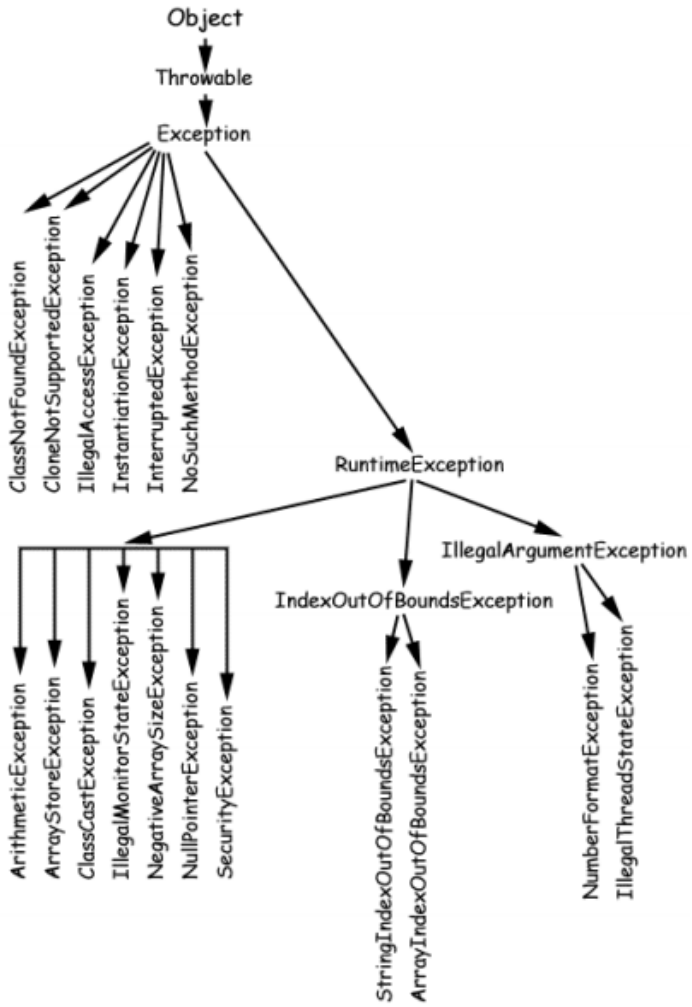
De lo dicho anteriormente, se entiende que un programa siempre debe manejar excepciones porque si no es así, cuando se produzca un error, el programa dejará de ejecutarse o lo hará mal.

En Java se utiliza la construcción **try-catch** para el manejo de excepciones. Su sintaxis es la siguiente:

```
try{
      Código que puede producir la excepción.
} catch(tipo_Excepcionnombre_variable)
{
      Código que se ejecuta si se produce la excepción anterior.
}
```

En cuanto al funcionamiento, el código que puede producir una excepción se sitúa en el bloque try. En el caso de que se produzca la excepción, esta será analizada por el bloque catch. El bloque try se abandona en el mismo momento que se produce la excepción por lo que el código que haya después del que ha producido la excepción no se ejecuta.

Para declarar el tipo de excepción que puede manejar el bloque catch se declara un objeto cuya clase es la clase de la excepción que se va a manejar o una de sus superclases.



Clases relacionadas con excepciones

Además, un bloque try puede ir seguido de varios bloques catch. En este caso cada bloque catch captura un tipo de excepción distinto:

```
try{
      Código que puede producir la excepción.
} catch(nombre_Excepcion e){
      Código que se ejecuta si se produce la excepción anterior.
} catch(nombre_Excepcion e1){
      Código que se ejecuta si se produce la excepción anterior.
}
```

Por último, es posible indicar un **bloque finally**, cuya utilidad es contener el código que queremos que se ejecute sea cual sea la excepción producida. Este bloque de código se ejecuta, aunque no se produzca excepción, y su principal utilidad es no repetir código en los bloques try o catch:

```
try{
      Código que puede producir la excepción.
} catch(nombre_Excepcion e){
```

```
      Código que se ejecuta si se produce la excepción anterior.
} finally{
      Código que se ejecuta siempre.
}
```

1.8 Excepciones en ficheros

Las principales excepciones que se pueden producir al trabajar con ficheros son las producidas al abrir archivos y las producidas al escribir o leer en ellos. Todas estas excepciones derivan de la clase padre **IOException.**

Al tratar de abrir un archivo mediante una llamada al constructor de la clase correspondiente, se puede producir la excepción **FileNotFoundException**.

Por tanto, esta excepción deberá ser contemplada siempre que se realice tal llamada. Existen dos formas de hacerlo:

- Manejar la excepción localmente, es decir, en la función en la que se está creando el objeto:

```
public leerArchivo() {
try {
FileReader fr = new FileReader(archivo);
            } catch (FileNotFoundException e) {
}
      }
```

- Propagar la excepción hacia arriba, en cuyo caso, deberá ser tratada externamente a la función en la que se está creando el objeto:

```
public leerArchivo() throws FileNotFoundException{
FileReader fr = new FileReader(archivo);
}
```

Y, externamente, en el lugar de la llamada:

```
try{
            leerArchivo();
      } catch (FileNotFoundException e) {
      }
```

Otra excepción destacable en el manejo de ficheros es **EOFException**, la cual se produce cuando, al realizar una operación de lectura, se alcanza el final del archivo. Es el caso de readInt().

2. TRABAJO CON FICHEROS DE INTERCAMBIO DE DATOS

También vas a añadir la funcionalidad de manejo de ficheros de datos en la aplicación. Para ello es fundamental conocer las clases básicas para la manipulación de formatos JSON y XML, por lo que repasas las operaciones básicas sobre dichos ficheros.

En el desarrollo de aplicaciones se genera mucha información. Un manejo eficiente de los datos es una necesidad de las organizaciones debido al creciente volumen de información que genera. Para satisfacer esta demanda, se utilizan diversos formatos y ficheros de datos que permiten almacenar y recuperar información de manera persistente y estructurada.

Los ficheros de intercambio facilitan la compatibilidad de información entre diferentes sistemas y aplicaciones, permitiendo una mayor interoperabilidad y flexibilidad en el tratamiento de datos. Uno de los más utilizados en la actualidad es JSON (JavaScript Object Notation) y XML (eXtensible Markup Language).

En este apartado, nos centraremos en el trabajo con ficheros XML, un formato de marcado extensible que permite almacenar datos de manera estructurada y legible. También abordaremos el manejo de ficheros JSON, que se ha convertido en el estándar para el intercambio de datos en aplicaciones web debido a su ligereza y facilidad de uso.

2.1 Trabajo con ficheros XML

Un lenguaje de marcado o lenguaje de marcas es una forma de codificar un documento utilizando etiquetas o marcas que contienen información adicional acerca de la estructura del texto o su presentación.

XML, eXtensible Markup Language ('lenguaje de marcas extensible'), es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) cuyo objetivo es almacenar datos en forma legible. Es un lenguaje derivado del lenguaje SGML que permite definir la gramática de lenguajes específicos para estructurar documentos grandes.

La aplicación de XML no se limita exclusivamente a su uso en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Como ejemplo se podría citar su uso en bases de datos, editores de texto, hojas de cálculo, etc.

Además, es una tecnología bastante simple que tiene a su alrededor otras más completas que la complementan y la hacen mucho más grande y con unas posibilidades mucho mayores. Hoy en día es muy utilizada, ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

Son muchas las ventajas de la definición de interfaces de usuario utilizando el lenguaje XML, entre las que destacan:

- Lenguaje de fácil aprendizaje.
- Permite definir la interfaz de forma independiente de la lógica y contenido de la aplicación.

Cada lenguaje de programación debe desarrollar las APIs necesarias para trabajar con los documentos XML.

ENLACE DE INTERÉS

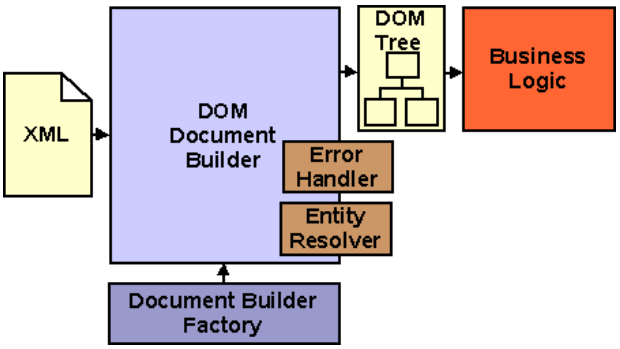
Obtén más información sobre el lenguaje de marcado XML:

Existen varias APIs para el trabajo con ficheros XML, entre ellas cabe destacar **SAX** (Simple API for XML Parsing) y **DOM** (DocumentObjectModel) ambas implementadas por el lenguaje de programación Java mediante la API **JAXP** (Java Api for XML Processing).

La principal diferencia entre ambas es que SAX procesa los documentos XML de manera secuencial y, según se va encontrando partes en XML separadas por su correspondiente etiqueta de inicio y cierre, va generando eventos que son recogidos por el manejador de eventos, cuya función es realizar una acción en función del evento sucedido. Sin embargo, en el caso de DOM, se lee todo el documento y devuelve un objeto del tipo Document que almacena la estructura del documento. La aplicación recorre y procesa esta estructura. Por tanto, DOM tiene acceso al documento entero de esta manera los

elementos y atributos de XML están disponibles simultáneamente. Sin embargo, en SAX solo se tiene acceso al elemento actual, es decir, al que acaba de procesarse.

Como punto positivo que cabe destacar de SAX es que el consumo de memoria es menor.



Diafragma de funcionamiento DOM

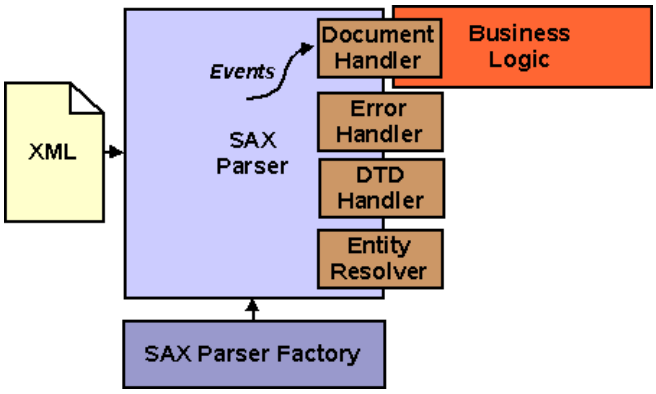


Diagrama de funcionamiento SAX

ENLACE DE INTERÉS

Amplía la información de la API JAXP:

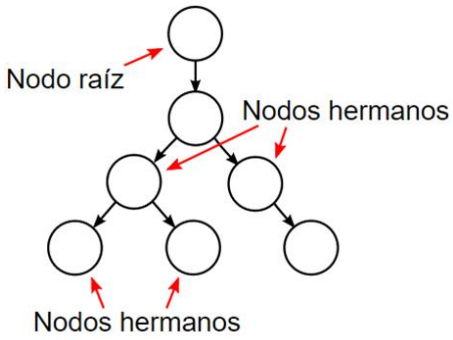
PARA SABER MÁS

Se recomienda visitar la siguiente web donde se analizan algunas APIs, además de mostrar ejemplos de uso:

2.1.1 Procesamiento de XML: XPath (Xml Path Language)

Dentro del procesamiento de ficheros XML, nos encontramos con XPath que es un lenguaje que permite navegar a través de los nodos de una estructura XML y extraer, editar, modificar, añadir y borrar la información. Es un estándar mantenido por el consorcio W3C y, por ello, podemos encontrar toda la información y referencia en la propia web.

Para que el lenguaje XPath pueda procesar un documento XML, este lo transforma o lo considera como un árbol de nodos, tal y como vemos en la siguiente imagen.



Árbol de nodos
Fuente: https://www.mclibre.org/consultar/xml/lecciones/xml-xpath.html

Si nos adentramos dentro de las diferentes versiones y estándares, veremos que estas han evolucionado, pero todas ellas tienen una serie de características en común:

- Definición del documento, nodos.
- Procesado de la información.
    - Mediante la localización de los nodos.

    - Mediante la evaluación del contenido e información.
- Estructuras más complejas y control de errores.

Java tiene un paquete que implementa XPath versión 1 y que, por lo tanto, permite evaluar los nodos, localizarlos, evaluar el contenido y modificarlo.

PARA SABER MÁS

Conoce la versión 3.1 de XML Path Language:

ENLACE DE INTERÉS

También es interesante que comprendas la documentación de la implementación de XPath con Java:

PARA SABER MÁS

Aquí encontrarás una explicación en profundidad del procesamiento de XML con XPath:

2.1.2 Conversiones de Objetos/XML con JAXB

JAXB (Java Architecture for XML Binding) es una API de Java que proporciona una manera sencilla y eficiente de mapear clases Java a representaciones XML y viceversa facilitando el proceso de exportar datos de una aplicación Java a un documento XML (marshalling) y de importar datos desde un documento XML a objetos Java (unmarshalling). Es parte del estándar Java SE y facilita el trabajo con datos XML en aplicaciones Java. Desde la versión 9 de Java SE se decidió eliminar JAXB del estándar para favorecer la modularidad por lo que si queremos usarlo tenemos que importarlo en nuestro proyecto de forma externa o utilizar el estándar de Java EE (actualmente conocido como Jakarta).

JAXB utiliza el sistema de anotaciones que son metadatos (precedidos del símbolo @) que se agregan a las clases y campos de Java para definir cómo se deben mapear a elementos y atributos XML y viceversa. Estas anotaciones proporcionan una forma sencilla de personalizar y controlar el proceso de marshalling (conversión de objetos Java a XML) y unmarshalling (conversión de XML a objetos Java):

- **@XmlRootElement:** define la clase Java como el elemento raíz de un documento XML.
- **@XmlElement:** mapea un campo de la clase Java a un elemento XML.
- **@XmlAttribute:** mapea un campo de la clase Java a un atributo de un elemento XML.
- **@XmlTransient:** indica que un campo de la clase Java no debe ser incluido en el XML.

```
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
```

```
import javax.xml.bind.annotation.XmlTransient;
@XmlRootElement // Define la clase como el elemento raíz del documento
XML
public class Persona {
    private int id;
    private String nombre;
    private String clave;
    // Constructor con parámetros
    public Persona(int id, String nombre, String clave) {
        this.id = id;
        this.nombre = nombre;
        this.apellido = apellido;
        this.clave = clave;
    }

    @XmlAttribute // Mapea el campo id a un atributo del elemento XML
    public int getId() {        return id;    }
    public void setId(int id) {        this.id = id;    }

    @XmlElement // Mapea el campo nombre a un elemento XML
    public String getNombre() {        return nombre;    }
    public void setNombre(String nombre) {        this.nombre =
nombre;    }

    @XmlTransient // Indica que el campo clave no debe ser incluido en
el XML
    public String getClave() {        return clave;    }
    public void setClave(String clave) {        this.clave = clave;
}
}
```

Por tanto, como hemos mencionado anteriormente, JAXB nos proporciona dos tipos de operaciones:

- **Marshalling** es el proceso de convertir un objeto Java en una representación XML.
- **Unmarshalling** es el proceso inverso, donde se convierte un documento XML en un objeto Java. Al igual que el marshalling, JAXB simplifica este proceso mediante el uso de anotaciones.

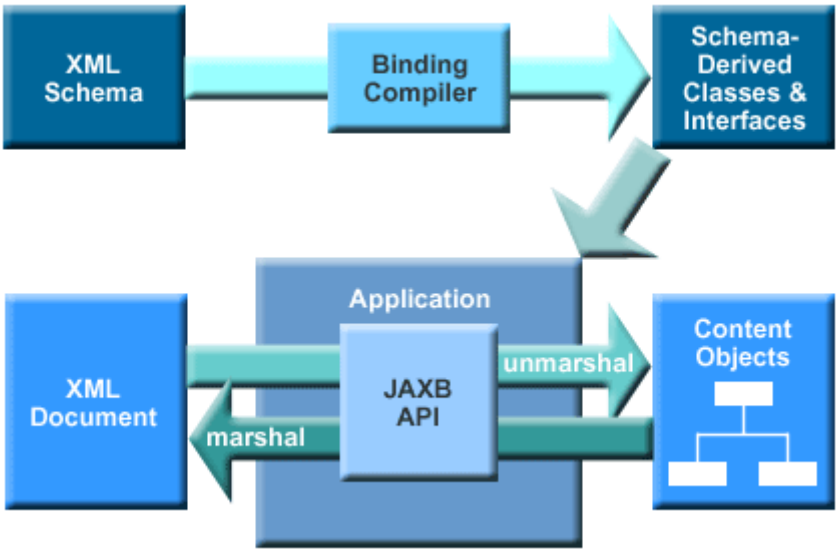


Diagrama de funcionamiento JAXB
Fuente: https://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/231

Por último, JAXB permite personalizar el mapeo de XML a objetos Java mediante configuraciones adicionales y el uso de diferentes anotaciones para controlar cómo se deben serializar y deserializar los datos.

ENLACE DE INTERÉS

Visualiza la guía oficial de la API JAXB

PARA SABER MÁS

Consulta la librería (en formato JAR) para usar JAXB:

EJEMPLO PRÁCTICO

En la aplicación que estamos desarrollando queremos poder importar un fichero XML exportado del control de acceso del almacén. Cuando una persona entra, se genera un fichero con los siguientes datos:

```
<?xml version="1.0"?>
<empresa>
<empleado id="1">
<nombre>Paco Gomez</nombre>
<username>pacogomez</username>
<password>123456</password>
</empleado>
</empresa>
```

A partir de este ejemplo, ¿cómo seríamos capaces de leer e importar el fichero?

El proceso sería el siguiente:

1. Usaremos la clase DocumentBuilder dentro del paquete, javax.xml.parsers.DocumentBuilder, y que nos permite parsear de forma sencilla documentos XML.

```
File archivo = new File("/ruta/almacen.xml");
try{
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder documentBuilder = dbf.newDocumentBuilder();
Document document = documentBuilder.parse(archivo);
document.getDocumentElement().normalize();
System.out.println("Elemento raiz:" +
document.getDocumentElement().getNodeName());
NodeList listaEmpleados = document.getElementsByTagName("empleado");
```

2. A partir de esta lista de empleados podríamos recorrer los datos:

```
for (int temp = 0; temp < listaEmpleados.getLength(); temp++) {
Node nodo = listaEmpleados.item(temp);
System.out.println("Elemento:" + nodo.getNodeName());
if (nodo.getNodeType() == Node.ELEMENT_NODE) {
Element element = (Element) nodo;
System.out.println("id: " + element.getAttribute("id"));
System.out.println("Nombre: " +
element.getElementsByTagName("nombre").item(0).getTextContent());
System.out.println("username: " +
element.getElementsByTagName("username").item(0).getTextContent());
System.out.println("password: " +
element.getElementsByTagName(password).item(0).getTextContent());
}
}
} catch (Exception e) {
e.printStackTrace();
}
```

2.2 Trabajo con ficheros JSON

Dentro del trabajo con diferentes formatos y ficheros en la aplicación que se está desarrollando, se plantea la necesidad de incorporar compatibilidad con el formato JSON.

El formato de intercambio de datos JSON (JavaScript Object Notation) es ligero y fácil de leer/escribir tanto para humanos como para máquinas. Fue diseñado como un formato de intercambio de datos que es simple y fácil de adaptar, basado en una versión simplificada de la sintaxis de JavaScript. Aunque en un inicio fue pensado para JavaScript, JSON es un formato de datos independiente del lenguaje y es compatible con una amplia variedad de lenguajes de programación, incluyendo Java.

Son muchas las ventajas de utilizar el lenguaje JSON, entre las que destacan:

- Simplicidad y Facilidad de Uso
- Interoperabilidad
- Soporte por Multitud de Lenguajes de Programación
- Lenguaje de fácil aprendizaje.

```json
{
  "empresa": {
    "empleados": [
      {
        "id": 1,
        "nombre": "Paco Gomez",
        "username": "pacogomez",
        "password": "123456"
      },
      {
        "id": 2,
        "nombre": "Ana Perez",
        "username": "anaperez",
        "password": "abcdef"
      }
    ]
  }
}

```

Ejemplo Fichero JSON

Fuente: Propia

En este caso el estándar de Java que usamos, Java SE, no incorpora una API estándar para el procesamiento de ficheros JSON, aunque en su versión para empresas (Java EE) tenemos una API estándar para el procesamiento de JSON, que es javax.json que aunque ofrece una solución estándar, existen algunas limitaciones en cuanto a funcionalidad y flexibilidad. Debido a esto es muy común el uso de bibliotecas externas que proporcionan características adicionales para los desarrolladores. Por ello, existen varias bibliotecas para trabajar con este formato de intercambio de datos, entre las que se destacan Jackson y GSON. Estas bibliotecas permiten la serialización y deserialización de objetos Java a JSON y viceversa, facilitando así la manipulación y procesamiento de datos JSON en aplicaciones Java.

- **Jackson:** permite trabajar con JSON en Java incluyendo la capacidad de manejar diferentes formatos de datos permite la conversión de objetos de manera sencilla y eficiente.

- **GSON:** es otra biblioteca popular para trabajar con JSON en Java. Es conocida por su simplicidad y facilidad de uso. GSON también nos permite convertir objetos Java a JSON y viceversa soportando configuraciones y personalizaciones para el mapeo de datos.

2.2.1 Procesamiento de JSON con Jackson

Jackson es una de las bibliotecas más populares y poderosas para trabajar con JSON en Java. Ofrece un conjunto completo de herramientas para la serialización y deserialización de objetos Java a JSON y viceversa. Además, Jackson proporciona capacidades avanzadas para manejar diferentes formatos de datos, como XML y CSV.

La clase ObjectMapper es el núcleo de la biblioteca Jackson. Proporciona métodos para convertir entre objetos Java y JSON.

```
ObjectMapper objectMapper = new ObjectMapper();
```

La clase JsonNode es utilizada para representar y manipular nodos individuales en un árbol JSON. Es útil para navegar y modificar estructuras JSON complejas.

```
JsonNode rootNode = objectMapper.readTree(jsonInput);
```

Con esto ya podremos lanzar consultas sobre el propio JSON.

```
JsonNode nombreNode = rootNode.get("nombre");
System.out.println("Nombre: " + nombreNode.asText());
```

Como podemos ver la clase JsonNode es muy versátil ya que puede ser cualquiera de los nodos que componen el fichero JSON.

ENLACE DE INTERÉS

Aquí encontrarás más información sobre la biblioteca Jackson de Java:

PARA SABER MÁS

Consulta la librería (en formato JAR) para poder empezar con Jackson:

2.2.2 Procesamiento de JSON con GSON

GSON es una biblioteca desarrollada por Google para trabajar con JSON en Java que proporciona un conjunto completo de herramientas para la serialización y deserialización de objetos Java a JSON y viceversa.

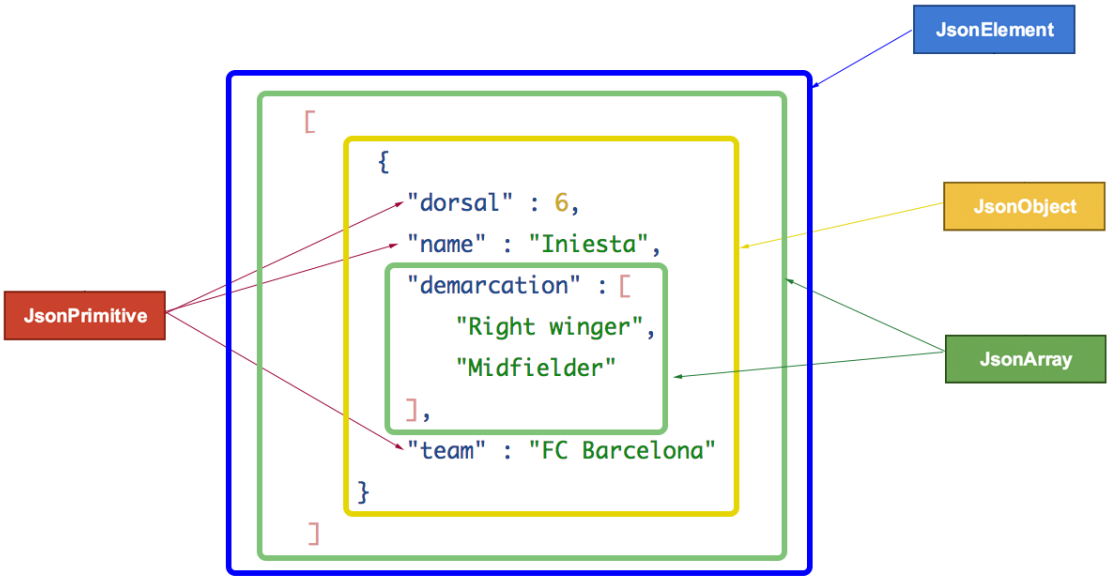
La clase Gson es el núcleo de la biblioteca GSON. Proporciona métodos para convertir entre objetos Java y JSON.

```
Gson gson = new Gson();
```

La serialización es un proceso muy sencillo con la clase Gson. Con tan solo llamar a una función podemos conseguir transformar el contenido de la variable.

```
String json = gson.toJson(miObjeto);
MiClase miObjeto = gson.fromJson(json, MiClase.class);
```

GSON también permite la manipulación de estructuras JSON mediante las clases JsonElement, JsonObject, y JsonArray. Además, La clase JsonParser se utiliza para transformar una cadena JSON y obtener estas estructuras.



Clases para Tratamiento de JSON

Fuente: https://jarroba.com/gson-json-java-ejemplos/

```
JsonElement elemento =
JsonParser.parseString("{\"nombre\":\"Juan\"}");
System.out.println(elemento);  // Imprime el elemento JSON completo
JsonObject objeto =
JsonParser.parseString("{\"nombre\":\"Juan\"}").getAsJsonObject();
String nombre = objeto.get("nombre").getAsString();  // "Juan"
JsonArray array = JsonParser.parseString("[\"manzana\",
\"naranja\"]").getAsJsonArray();
String fruta = array.get(0).getAsString();  // "manzana"
```

ENLACE DE INTERÉS

Conoce la información completa sobre la biblioteca GSON de Java:

PARA SABER MÁS

También es interesante que consultes la librería (en formato JAR) para poder empezar con GSON:

EJEMPLO PRÁCTICO

En la aplicación que estamos desarrollando, queremos poder manejar datos de estudiantes en formato JSON. Necesitamos poder importar y exportar la información de los estudiantes, que incluye su nombre, edad y los cursos en los que están inscritos. Utilizaremos la biblioteca GSON para facilitar la serialización y deserialización de estos datos.

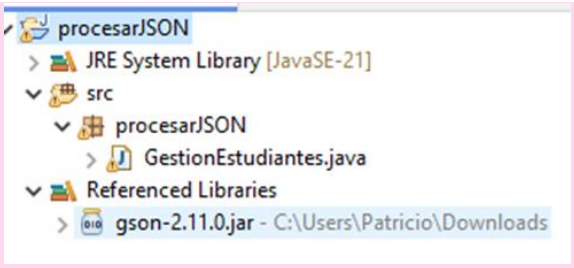
```
[
  {
    "nombre": "Ana",
    "edad": 22,
    "cursos": ["Matemáticas", "Historia"]
  },
  {
    "nombre": "Luis",
    "edad": 25,
    "cursos": ["Física", "Química"]
  }
]
```

A partir de este ejemplo, ¿cómo seríamos capaces de leer e importar el fichero?

El proceso sería el siguiente:

1. Descargar el JAR con GSON:

2. Agregar el JAR a tu proyecto haciendo clic derecho en el proyecto -> Properties -> Build Path -> Add External Archives y seleccionando el archivo JAR descargado.



3. Agregar la clase GestionEstudiantes que se encargará de Leer el fichero JSON, agregar/quitar estudiantes e imprimir la lista de ellos.

```
package procesarJSON;

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

public class GestionEstudiantes {
    public static void main(String[] args) {
        // JSON de ejemplo
        String json =
"[{\"nombre\":\"Ana\",\"edad\":22,\"cursos\":[\"Matemáticas\",\"Histo
ria\"]},{\"nombre\":\"Luis\",\"edad\":25,\"cursos\":[\"Física\",\"Quí
mica\"]}]";

        // Crear una instancia de Gson
        Gson gson = new Gson();

        // Parsear el JSON a un JsonArray
        JsonArray estudiantes =
JsonParser.parseString(json).getAsJsonArray();

        // Imprimir la lista de estudiantes
        imprimirEstudiantes(estudiantes);

        // Agregar un nuevo estudiante
        JsonObject nuevoEstudiante = new JsonObject();
        nuevoEstudiante.addProperty("nombre", "Maria");
        nuevoEstudiante.addProperty("edad", 20);
        JsonArray cursos = new JsonArray();
        cursos.add("Biología");
        cursos.add("Química");
        nuevoEstudiante.add("cursos", cursos);
        estudiantes.add(nuevoEstudiante);

        // Imprimir la lista de estudiantes después de agregar uno
nuevo
        System.out.println("\nDespués de agregar un nuevo
estudiante:");
        imprimirEstudiantes(estudiantes);
```

```
        // Quitar un estudiante (por nombre, por ejemplo)
        quitarEstudiante(estudiantes, "Luis");

        // Imprimir la lista de estudiantes después de quitar uno
        System.out.println("\nDespués de quitar un estudiante:");
        imprimirEstudiantes(estudiantes);
    }

    // Método para imprimir la lista de estudiantes
    private static void imprimirEstudiantes(JsonArray estudiantes) {
        for (JsonElement estudiante : estudiantes) {
            JsonObject obj = estudiante.getAsJsonObject();
            System.out.println("Nombre: " +
obj.get("nombre").getAsString());
            System.out.println("Edad: " +
obj.get("edad").getAsInt());
            System.out.println("Cursos: " +
obj.get("cursos").getAsJsonArray());
            System.out.println();
        }
    }

    // Método para quitar un estudiante por nombre
    private static void quitarEstudiante(JsonArray estudiantes,
String nombre) {
        for (int i = 0; i < estudiantes.size(); i++) {
            JsonObject obj = estudiantes.get(i).getAsJsonObject();
            if (obj.get("nombre").getAsString().equals(nombre)) {
                estudiantes.remove(i);
                break;
            }
        }
    }
}
```

RESUMEN FINAL

El uso de los ficheros dentro de cualquier sistema operativo es una de las funcionalidades más importantes y por ese motivo incorporan un sistema de gestión para el manejo y administración de directorios y ficheros. Como desarrolladores, es importante conocer cómo funcionan los sistemas de gestión dentro del sistema operativo para llevar a cabo las operaciones necesarias.

En todos los lenguajes de programación encontraremos bibliotecas dedicadas a la gestión de los ficheros. En el lenguaje de programación Java se utiliza la clase File para gestionar archivos y directorios. Esta clase permite crear y eliminar archivos o directorios, al igual que obtener información de ellos tal y como el tamaño de un archivo, los archivos que contiene un directorio, etc. La clase File se encuentra en el paquete java.io.

Con la evolución de las redes de ordenadores y de Internet, los sistemas de almacenamiento han evolucionado también, provocando una descentralización. Las librerías de programación dan respuesta a este tipo de circunstancias y, por ejemplo, en java se trabaja con flujos o streams de información. Un flujo es un objeto que hace de intermediario entre un programa y la fuente o el destino de la información. Esto permite que el programa pueda leer o escribir información en el flujo sin importarle el origen o destino de la información ni el tipo de datos. En concreto, en Java encontramos clases para la lectura de datos desde un stream, InputStream, o como salida, OutputStream, que posteriormente se adaptan a la tipología del fichero o la fuente de información.

Uno de los estándares de ficheros más usados es el estándar XML, que es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C), cuyo objetivo es almacenar datos en forma legible. Un estándar que nos permite almacenar y distribuir información y sobre el cual Java también posee librerías que permiten analizar, modificar y crear ficheros de tipo XML. Además de XML, el formato JSON (JavaScript Object Notation) ha ganado mucha popularidad en los últimos años debido a su simplicidad y eficiencia. JSON es un formato ligero de intercambio de datos fácil de leer y escribir para humanos y máquinas. Aunque fue diseñado originalmente para JavaScript, JSON es independiente del lenguaje y es ampliamente compatible con muchos lenguajes de programación, incluyendo Java.