UNIDAD 1: MANEJO DE FICHEROS

Módulo Profesional: Acceso a Datos

RESUMEN INTRODUCTORIO

En esta unidad introduciremos los conceptos básicos sobre la lectura de ficheros, 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 ha realizado toda la programación de la interfaz y funciones de escritura. Ahora, deberá implementar una serie de funciones que le sirvan para guardar el documento, así como para recuperar la información que haya guardado previamente.

¿Qué clases existen en Java para poder manejar ficheros? ¿Permiten realizar todas las funciones habituales o bien hay que incorporar librerías externas?

Con esta unidad seremos capaces 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

El primer paso para comenzar a añadir la funcionalidad de la gestión de ficheros en nuestra aplicación es conocer las clases básicas de gestión de ficheros y directorios.

Te planteas las siguientes preguntas de inicio. ¿Cuál es la clase base de inicio de trabajo dentro de Java? ¿Es independiente del sistema operativo? ¿Es necesario incluir librerías externas? ¿Cómo funcionan las clases necesarias para las operaciones básicas sobre ficheros y directorios?

A partir de estas cuestiones se puede establecer el inicio de trabajo con ficheros y directorios para nuestra aplicación.

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

En el siguiente enlace podrás encontrar información completa sobre esta clase en la documentación oficial de Java:

http://docs.oracle.com/javase/7/docs/api/java/io/File.html

ENLACE DE INTERÉS

Si trabajamos con OpenJDK tenemos compatibilidad con la clase File, además de tener otros paquetes y librerías interesantes:

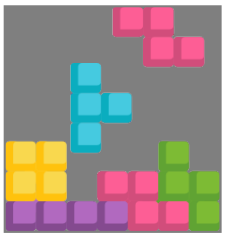
https://devdocs.io/openjdk~8/java/io/file
https://devdocs.io/openjdk~8/java/nio/file/files

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*.

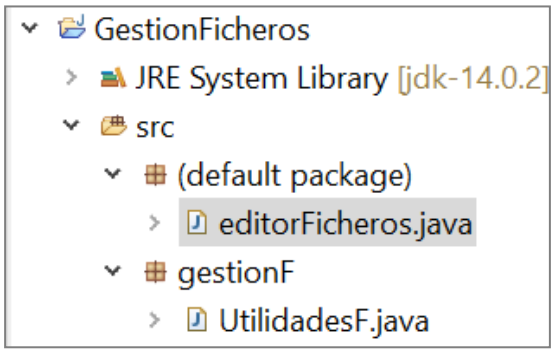


EJEMPLO PRÁCTICO

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.



Estructura del proyecto
Fuente: Elaboración propia

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]);
		}
	}

}
```

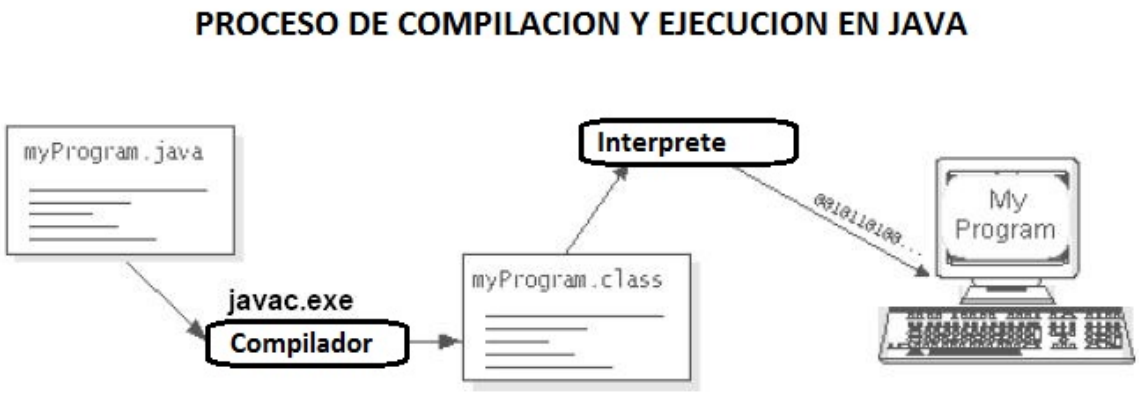
3. La clase editorFicheros será la clase principal.

```
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.).

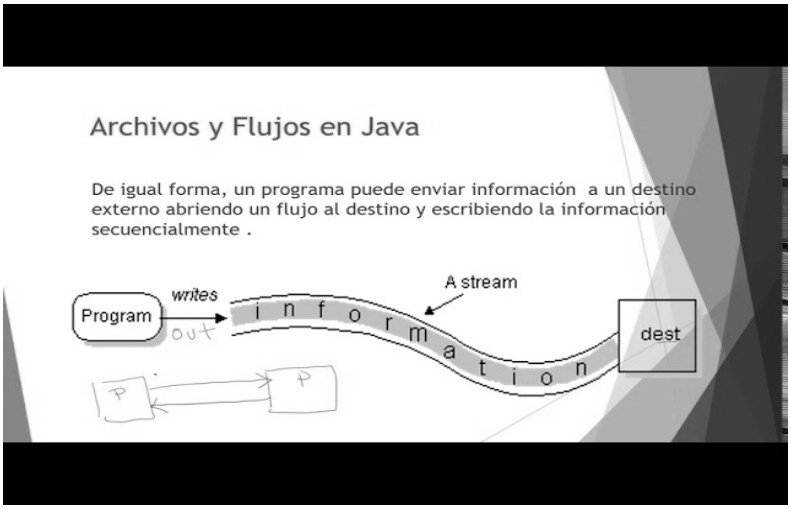
Los algoritmos para leer y escribir datos siguen generalmente el mismo esquema.

Para leer, sería:

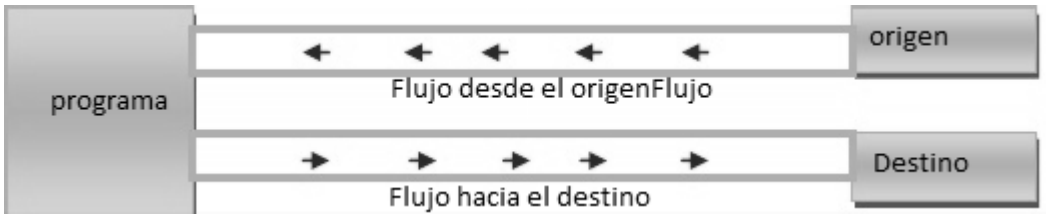
- Abrir un flujo desde un origen.
- Mientras haya información, leer información.
- Cerrar el flujo.

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

Para ampliar información sobre los flujos y algunas clases utilizadas se recomienda visitar:

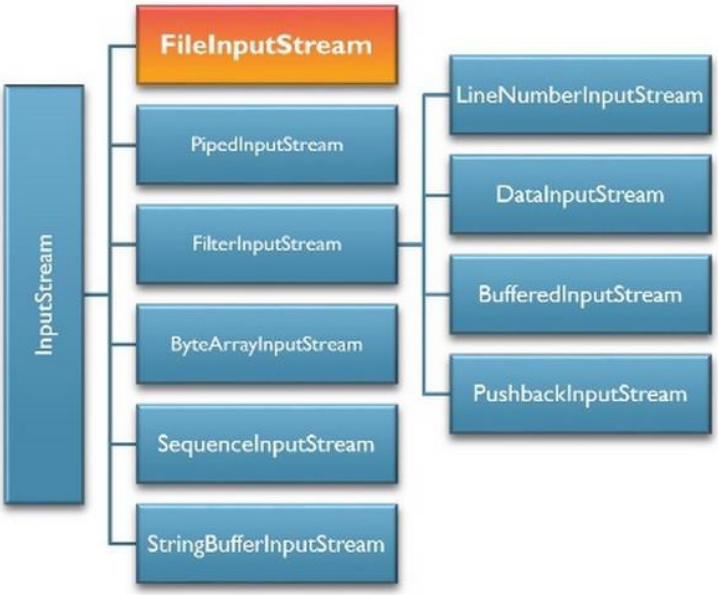
http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/archivos/flujos.htm

Acabamos de estudiar las diferencias entre los flujos de tipo byte y de tipo carácter. ¿Serías capaz de poner un ejemplo de ficheros de ambos tipos? ¿Qué tipo usarías para el almacenamiento de información tipo csv?

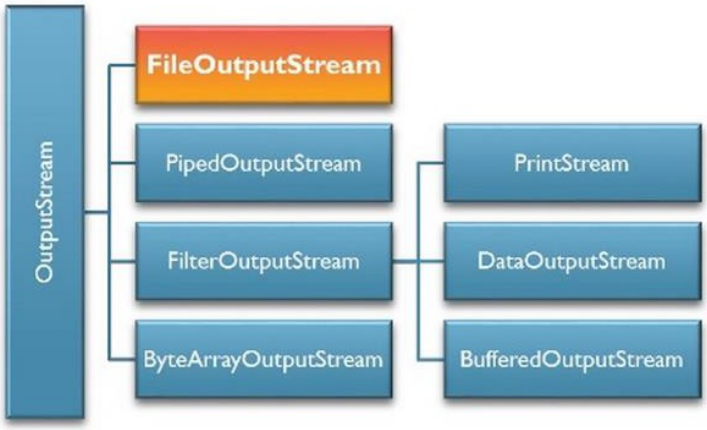
1.3 Flujos de bytes

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

En el siguiente enlace se puede obtener información más detallada sobre InputStream:

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html

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 ((b = in.read()) != -1) {
}
```

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

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)
```

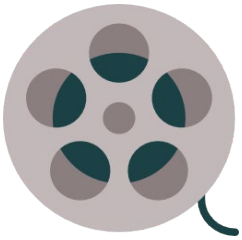
2. En este método incluiremos el siguiente código:

```
	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);
			}

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

	}
```



VIDEO DE INTERÉS

En el siguiente vídeo tenemos un ejemplo de uso de FileInputStream y FileOutputStream:

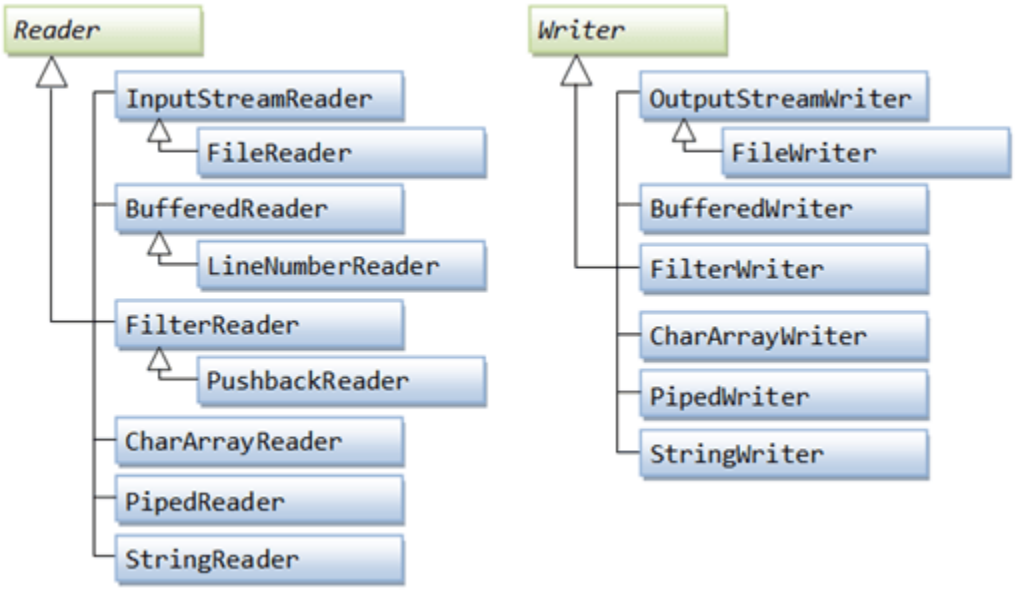
https://www.youtube.com/watch?v=4TFi4tyBQGw

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.



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 ficheroTXT)
```

2. En este método incluiremos el siguiente código:

```
        public void importarTXT(String ficheroActual,String ficheroTXT)
throws IOException
{
        FileReader in=null;
        FileWriter out=null;
        try //crea flujos de entrada y salida
        {
                in=new FileReader("archivooriginal.txt");
                out=new FileWriter("archivocopiado.txt");
                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 caracter
                }
        }
        finally //Cierra los flujos
        {
                if(in!=null) in.close();
               if(out!=null) out.close();
        }
    }
```

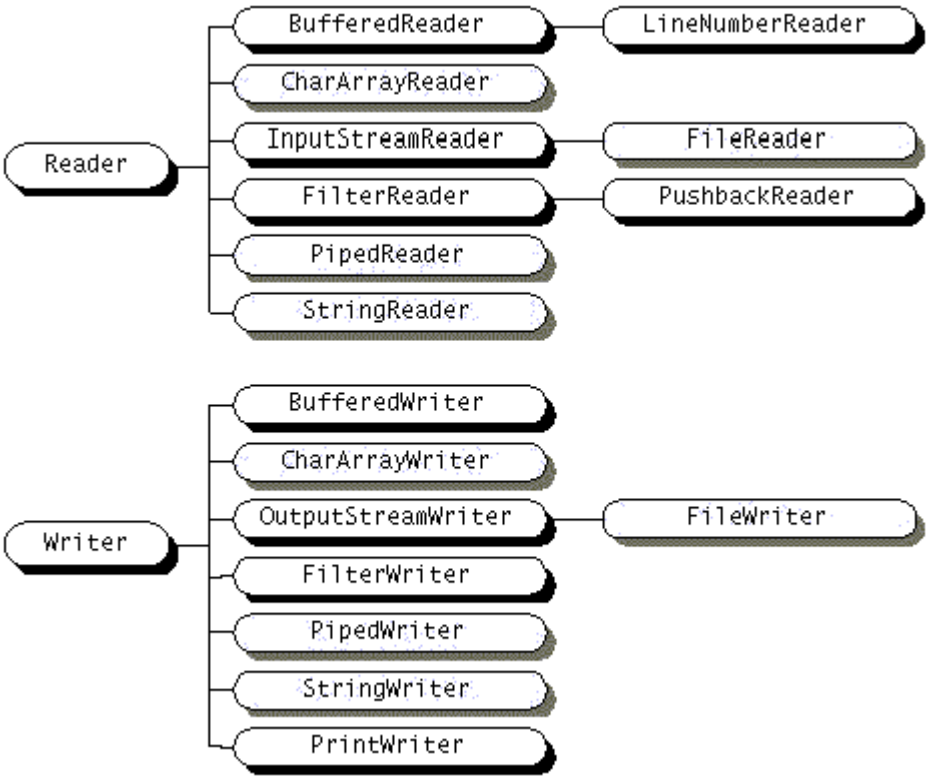
1.5 Las jerarquías de clases

En el lenguaje Java los flujos de datos se describen mediante clases que forman jerarquías según sea el **char** Unicode de 16 bits o **byte** de 8 bits. A su vez, las clases se agrupan en jerarquías según sea su función de lectura o de escritura.

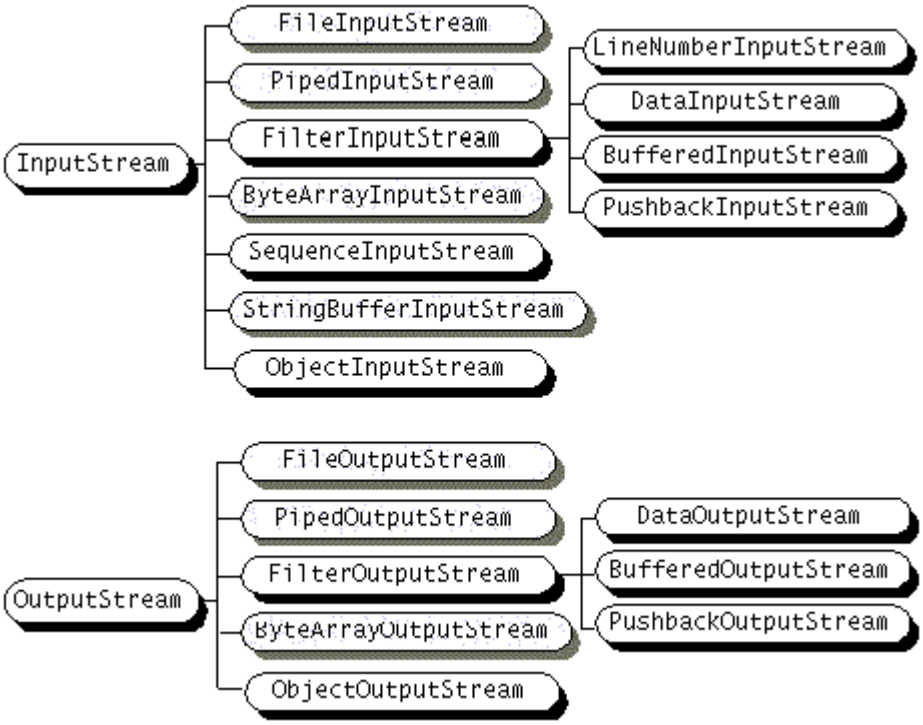
La característica de internacionalización del lenguaje Java es la razón por la que existe una jerarquía separada de clases para la lectura y escritura de caracteres.

Todas estas clases se encuentran en el **java.io**, por lo que al principio del código fuente tendremos que escribir la sentencia

```
import java.io.*;
```

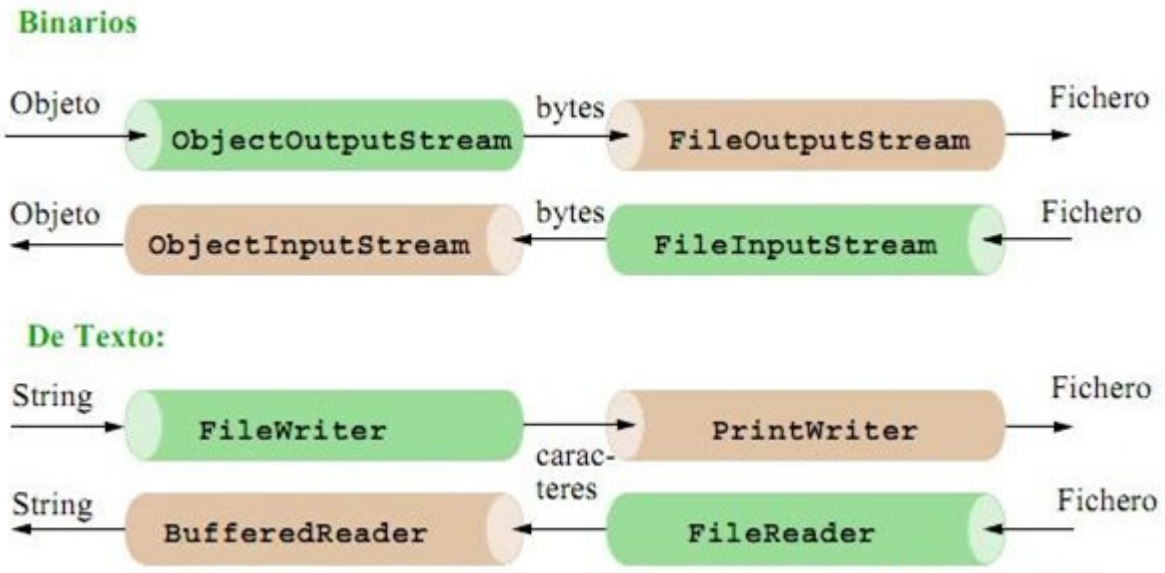


char Unicode, 16 bits



byte, 8 bits.

Reader y *Writer* son las clases bases de la jerarquía para los flujos de caracteres. Para leer o escribir datos binarios tales como imágenes o sonidos, se emplea otra jerarquía de clases cuyas clases base son *InputStream* y *OutputStream*.



VIDEO DE INTERÉS

En el siguiente enlace encontrarás una serie de vídeos sobre la programación en Java, en concreto, sobre el acceso a ficheros:

https://www.youtube.com/watch?v=etQN4EfYN7k

1.6 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.

1.6.1 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.

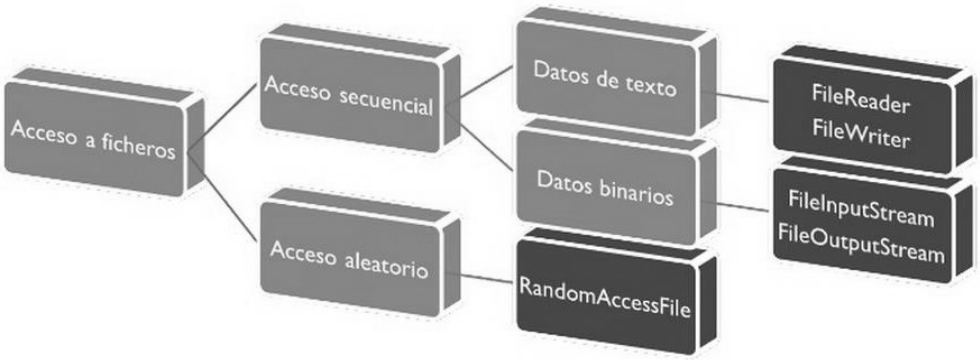
  Para trabajar con ficheros de acceso aleatorio se utiliza la clase java.io.**RandomAccessFile**, que implementa las interfaces DataInput y DataOutput, las cuales permiten leer y escribir en el fichero.
  Para trabajar con la clase **RandomAccessFile** se debe indicar el nombre del archivo que se quiere 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");
```

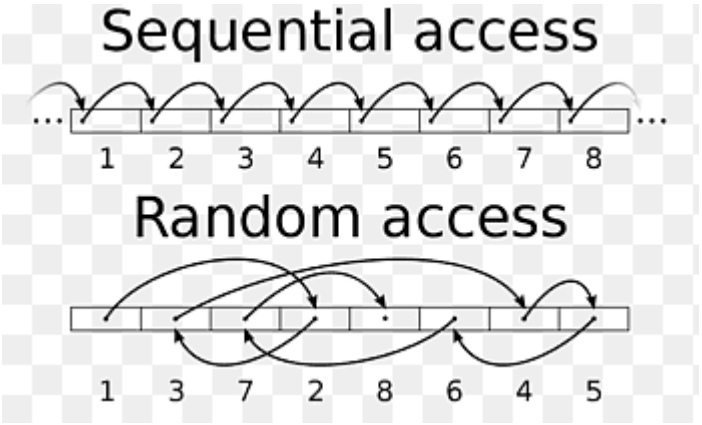


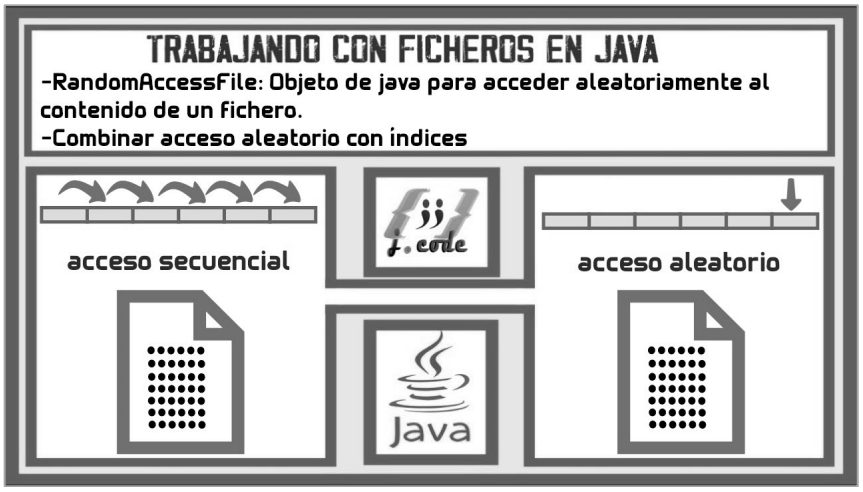
Tipos de accesos y clases en Java

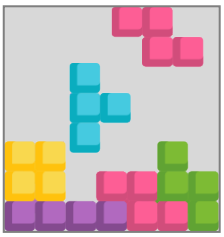
Descripción	Declaración
Constructor. El String `fich` indica el nombre del fichero. El String modo será `"r"` para leer y `"rw"` para leer y escribir	`RandomAccessFile (String fich, String modo) throws FileNotFoundException`
Retorna el tamaño actual en bytes del fichero	`long length() throws IOException`
Cambia el tamaño del fichero al indicado, en bytes	`void setLength(long nueva) throws IOException`
Pone el puntero de lectura/escritura a pos	`void seek(long pos) throws IOException`
Retorna el puntero de lectura/escritura	`long getFilePointer() throws IOException`
Intenta leer un array de bytes. Retorna el número de bytes leídos, o -1 si no quedan más	`int read(byte[] b) throws IOException`
Lee repetidamente hasta rellenar el array de bytes completo. Lanza `EOFException` si se acaba el fichero y no se ha podido leer todo	`void readFully(byte[] b) throws IOException`
Lee un double. Lanza `EOFException` si se acaba el fichero	`double readDouble() throws IOException`
Lee un int Lanza `EOFException` si se acaba el fichero	`int readInt() throws IOException`
Lee bytes convirtiéndolos a caracteres hasta encontrar un final de línea	`String readLine() throws IOException`
Escribe un array de bytes	`void write(byte[] b) throws IOException`
Escribe un double	`void writeDouble(double d) throws IOException`
Escribe un int	`void writeInt(int i) throws IOException`
Escribe los caracteres de un string convirtiéndolos primero a bytes (sólo vale para caracteres de 8 bits)	`void writeBytes(String s) throws IOException`
Cerrar el fichero	`void close() throws IOException`

COMPRUEBA LO QUE SABES

Acabamos de estudiar los diferentes tipos de accesos a ficheros. ¿Tiene sentido acceder a un fichero de forma secuencia y de forma aleatoria?


Sequential access
1 2 3 4 5 6 7 8
Random access
1 3 7 2 8 6 4 5


TRABAJANDO CON FICHEROS EN JAVA
-RandomAccessFile: Objeto de java para acceder aleatoriamente al contenido de un fichero.
-Combinar acceso aleatorio con indices
acceso secuencial
acceso aleatorio
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());
        }
    }
}
```

VIDEO DE INTERÉS

En el siguiente vídeo se muestran ejemplos de entrada y salida en Java:

https://www.youtube.com/watch?v=Ng0_7uZyIoA

1.6.2 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.

1.7 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 externa

1.7.1 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.

1.7.2 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 readUnsignedShort();	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

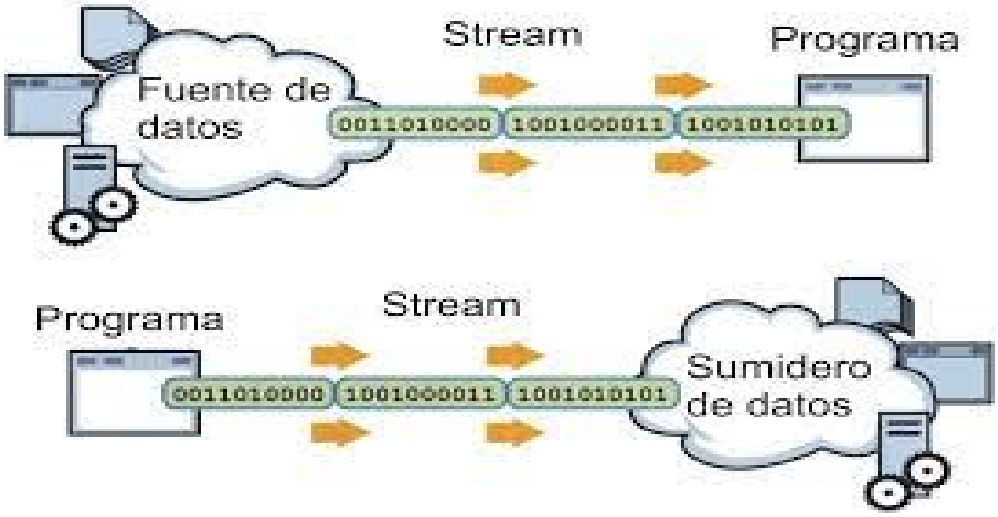
1.7.3 Ficheros de objetos. Serialización

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 .



Despues de serializado un objeto y escrito dentro de un archivo este puede ser leido y deserializado, el tipo de informacion, los bytes que representan al objeto y los datos almacenados pueden ser usados para recrear el objeto en memoria, lo mas llamativo de este proceso es que completamente independiente de la JVM, Java Virtual Machine, lo cual implica que podemos serializar un objeto en una plataforma y deserializar el objeto en otra plataforma, dos clases de este estilo son:

- **ObjectInputStream**
- **ObjectOutputStream**

Las cuales son streams de alto nivel con los metodos necesarios para poder serializar y deserializar un objeto.

La clase **ObjectOutputStream** contiene muchos metodos para escribir varios tipos de datos pero destacaremos el siguiente metodo:

```
public final void writeObject(Objeto x) throws IOException
```

Este metodo serializa un objeto y lo manda a un stream de salida y de forma similar la clase ObjectInputStream contiene el siguiente metodo para deserializar:

```
public final Objeto readObject() throws IOException, ClassNotFoundException
```

Este método recupera el próximo Objeto mas allá del stream y lo deserializa, el valor devuelto es un Objeto por lo que deberás convertirlo a su tipo adecuado, para serializar una clase correctamente debemos tener dos condiciones:

La clase debe poder implementar la interfaz **java.io.Serializable**
Todos los campos de la clase deben poder ser serializable y de no poder ser serializable debe ser marcado como **transient**.

Una forma muy simple de saber si puede serializar o no, es implementando la interfaz y en caso afirmativo ser puede hacer de lo contrario no ahora hablemos como serializar un objeto a través del siguiente ejemplo donde primero crearemos una clase en la cual implementaremos la serialización:

EmpleadoSerial.java:

```
public class EmpleadoSerial implements java.io.Serializable
{
	public String nombre;
	public String direccion;
	public transient int DNI;
	public int numero;

	public void chequeCorreo()
	{
		System.out.println("Mandando cheque a " + nombre
				+ " " + direccion);
	}
}
```

En este caso podemos ver como se implementa a Serializable, luego tendremos las variables para almacenar el nombre, la direccion, el DNI (al cual lo haremos transient) y por ultimo el numero de la direccion, despues tendremos un metodo el cual mostrara un mensaje donde notificara que se mando el cheque al empleado en la direccion indicada, con nuestra clase y la serializacion implementada procederemos a la clase base la cual ira en el siguiente ejemplo:

DemoSerializar.java:

```
import java.io.*;

public class DemoSerializar
{
	public static void main(String[] args)
	{
		EmpleadoSerial e = new EmpleadoSerial();
		e.nombre = "Martin Miranda";
		e.direccion = "Mercedes Sosa, Santa Monica";
		e.DNI = 29087695;
		e.numero = 333;

		try
		{
			FileOutputStream archivo = new
				FileOutputStream("empleado.ser");
			ObjectOutputStream salida = new
				ObjectOutputStream(archivo);
			salida.writeObject(e);
			salida.close();
			archivo.close();
			System.out.println("Datos guardados en empleado.ser");
```

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

En este programa importaremos todos los paquetes de java.io, luego en la clase crearemos un objeto llamado e de la clase creada anteriormente, EmpleadoSerial, donde le pasaremos el nombre, la direccion, el DNI y el numero de la direccion, despues utilizaremos un try / catch para ante cualquier **eventualidad** ver un mensaje de error, primero crearemos un objeto del tipo FileOutputStream el cual nos dara la capacidad de crear un archivo donde le indiquemos, la linea es esta:

```
FileOutputStream archivo = new FileOutputStream("empleado.ser");
```

La siguiente línea es la encargada de crear el objeto para la serializacion al cual llamaremos salida y nuestro stream de salida será el archivo indicado justamente en archivo, después procederemos a escribir en salida por medio de **writeObject**() y le enviaremos el objeto e que antes habíamos creado, para luego cerrar a salida y archivo, y poder escribir correctamente el archivo para finalmente mostrar una notificacion de que fue guardado el empleado, el catch se iniciara si aparece una excepción de tipo **IOException** donde mostrara el volcado de la pila (stack), si lo compilamos y ejecutamos obtendremos la siguiente salida:

```
tinchicus@dbn001vrt:~/programacion/java/codigo$ java DemoSerializar
Datos guardados en empleado.ser
tinchicus@dbn001vrt:~/programacion/java/codigo$
```

Con esto si miramos nuestro archivo generado será de la siguiente forma:

```
tinchicus@dbn001vrt:~/programacion/java/codigo$ cat empleado.ser
 ▒ercedes Sosa, Santa MonicatMartin
Mirandatinchicus@dbn001vrt:~/programacion/java/codigo$ PuTTY
```

Como pueden ver se guardo como una secuencia de bytes y no en formato ASCII directamente, ahora procedamos a deserializar el archivo para obtener la información almacenada y para ello lo haremos a través del siguiente ejemplo:

DemoDeserializar.java:

```
import java.io.*;

public class DemoDeserializar
{
	public static void main(String[] args)
	{
		EmpleadoSerial e = null;
		try
		{
			FileInputStream archivo = new
					FileInputStream("empleado.ser");
			ObjectInputStream entrada = new
					ObjectInputStream(archivo);
			e = (EmpleadoSerial) entrada.readObject();
			entrada.close();
			archivo.close();
		}
		catch(IOException i)
		{
			i.printStackTrace();
			return;
		}
		catch(ClassNotFoundException c)
		{
			System.out.println("Clase EmpleadoSerial no encontrada");
```

```
                c.printStackTrace();
                return;
            }

            System.out.println("Deserializando al empleado...");
            System.out.println("Nombre: " + e.nombre);
            System.out.println("Direccion: " + e.direccion);
            System.out.println("DNI: " + e.DNI);
            System.out.println("Numero: " + e.numero);
        }
}
```

En este ejemplo primero vamos a crear un objeto de **EmpleadoSerial** llamado e, nuestro siguiente paso será crear un objeto llamado archivo de la clase **FileInputStream**, que al igual que antes nos permitirá acceder a un archivo pero esta vez para lectura y le diremos cual es nuestro archivo, nuestra siguiente linea será la encargada de crear el stream de entrada el cual será direccionado a nuestro archivo, nuestro siguiente paso será leer el contenido del archivo por medio de **readObject**() y todo ese contenido lo deserializará y lo asignara a nuestro objeto e, despues cerramos nuestros dos streams, archivo y salida, para ir redondeando los catch estan para si ocurre alguna de las dos excepcions, **IOException** y **ClassNotFoundException**, donde nos mostrara una notificación, por último mostraremos el contenido de nuestro empleado por medio de las variables instanciadas del objeto, compilemos y probemos para ver si funciona correctamente:

```
tinchicus@dbn001vrt:~/programacion/java/codigo$ java DemoDeserializar
Deserializando al empleado…
Nombre: Martin Miranda
Direccion: Mercedes Sosa, Santa Monica
DNI: 0
Numero: 333
tinchicus@dbn001vrt:~/programacion/java/codigo$
```

Como podemos ver funciono perfectamente salvo por el DNI, esto es debido a que usamos la palabra transient y esto hizo que al momento de serializar a nuestro empleado este dato fuera ignorado por esto no lo recuperamos pero por el resto funciono correctamente, si quieren prueben de quitar la palabra **transient** de la clase EmpleadoSerial, compílenlo nuevamente, vuelvan a crear al empleado y

veamos como es la salida de nuestro programa:

```
tinchicus@dbn001vrt:~/programacion/java/codigo$ java DemoDeserializar
Deserializando al empleado…
Nombre: Martin Miranda
Direccion: Mercedes Sosa, Santa Monica
DNI: 29087695
Numero: 333
tinchicus@dbn001vrt:~/programacion/java/codigo$
```

Y en este caso vemos como si nos trajo el documento dado que ahora no lo ignora. En resumen, hemos visto que es la serialización, qué ventajas tiene, como se puede utilizar, como guardar información y como leer información.

PARA SABER MÁS

Se recomienda realizar la siguiente lectura para ampliar información sobre la serialización de objetos en Java:

http://www.javahispano.org/storage/contenidos/serializacion.pdf

Acabamos de estudiar la serialización de objetos en Java. ¿Podrías poner algún ejemplo de serialización y su uso?

1.8 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.

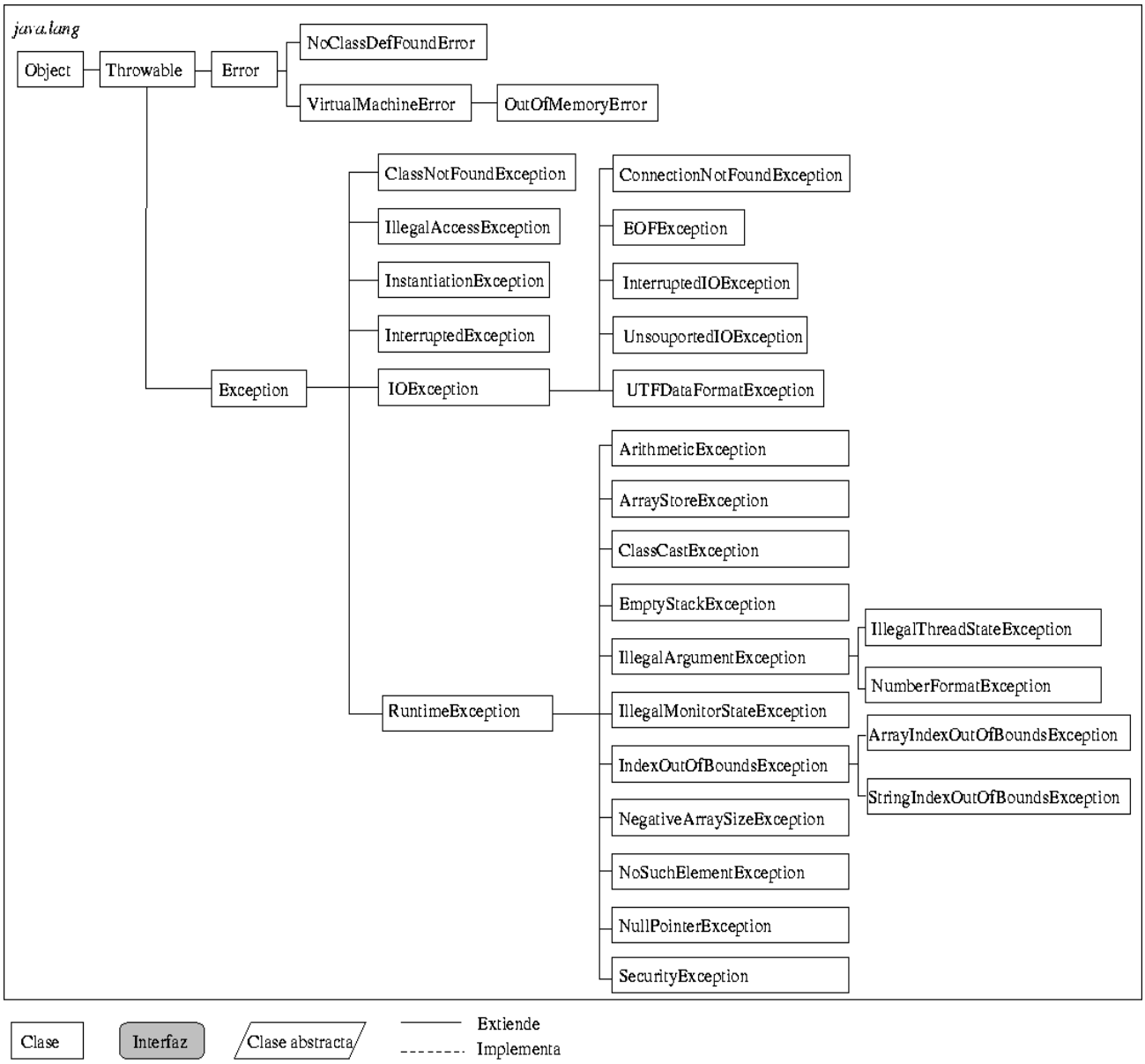
De lo dicho anteriormente, se entiende que un programa siempre debe manejar excepciones porque si no es así, cuando se produzca un error, el programa dejará de ejecutarse o lo hará mal.

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.
}
```

```
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.
}
```

1.9 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 XML

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 XML.

Te planteas las siguientes preguntas de inicio. ¿Qué estándares debemos tener en cuenta para manejar el formato XML? ¿Qué clases y cómo podemos trabajar sobre XML desde Java?

A partir de estas cuestiones, se puede establecer cómo evolucionar nuestra aplicación para que se integre con 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.

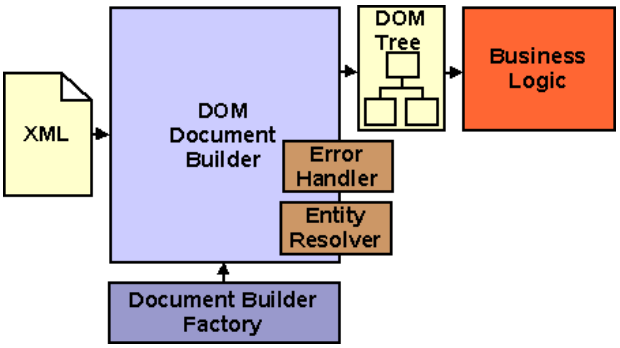
ENLACE DE INTERÉS

En el siguiente enlace se puede obtener más información sobre el lenguaje de marcado XML:

http://www.mclibre.org/consultar/xml/lecciones/xml_quees.html

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

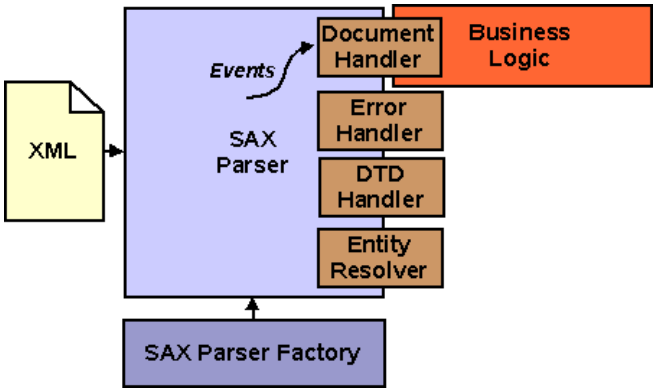


Diagrama de funcionamiento SAX

ENLACE DE INTERÉS

En la siguiente web encontrará información de la API JAXP:

https://jaxp.java.net/

COMPRUEBA LO QUE SABES

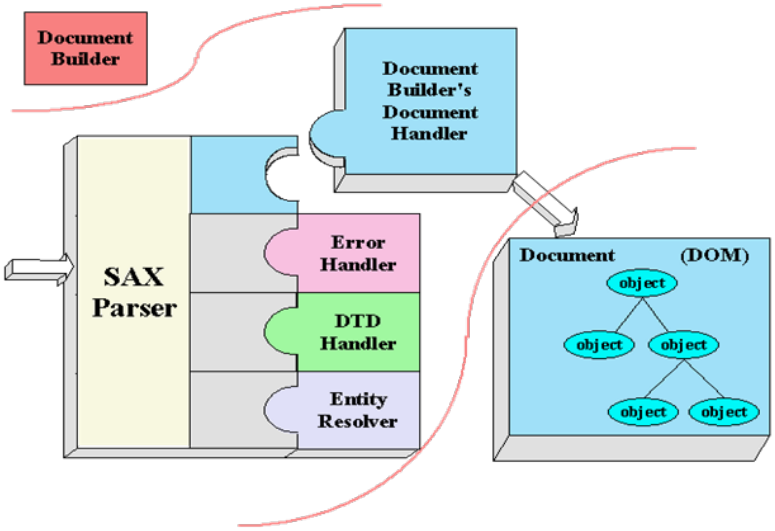
Acabamos de estudiar dos APIs para el procesado de información con Java y XML.

PARA SABER MÁS

Se recomienda visitar la siguiente web donde se analizan las APIs aquí estudiadas y otras más, además de mostrar ejemplos de uso:

http://howtodoinjava.com/2014/07/30/dom-vs-sax-parser-in-java/

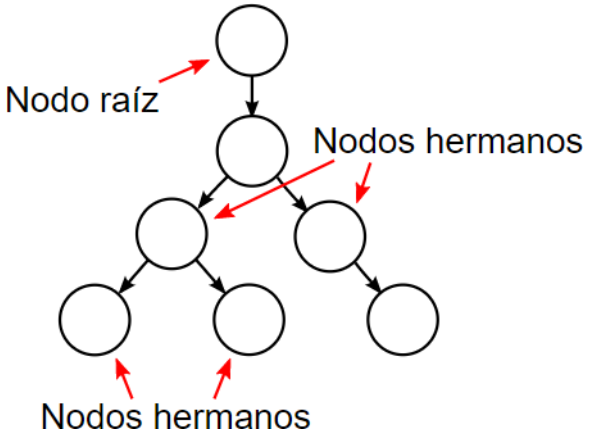
La implementación que hace Sun del **API DOM usa las librerías SAX** para **leer en los datos XML y construye el árbol de objetos de datos que constituye el DOM. Esta implementación también proporciona un marco de trabajo para ayudarnos a sacar el árbol de objetos como datos XML**. El siguiente diagrama muestra cómo opera el **DocumentBuilder de Sun** detrás de la escena.



2.1 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** y que, por lo tanto, permite evaluar los nodos, localizarlos, evaluar el contenido y modificarlo.

ENLACE DE INTERÉS

En el siguiente enlace encontrarás la versión 3.1 de XML Path Language:

https://www.w3.org/TR/2017/REC-xpath-31-20170321/

ENLACE DE INTERÉS

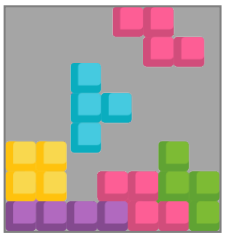
En el siguiente enlace estudiarás la documentación de la implementación de XPath con Java:

https://docs.oracle.com/en/java/javase/13/docs/api/java.xml/javax/xml/xpath/package-summary.html

ARTÍCULO DE INTERÉS

En el siguiente enlace encontrarás un artículo que explica con profundidad el procesamiento de XML con XPath:

https://www.mclibre.org/consultar/xml/lecciones/xml-xpath.html



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:

- 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");
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");
```

- 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("321423").item(0).getTextContent());
 }
 }
 } catch (Exception e) {
 e.printStackTrace();
}
```

3. 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.