UD 1. Manejo de ficheros

Contenido

1 INTRODUCCIÓN

Desde el punto de vista informático, un **fichero** es una colección de información que almacenamos en un soporte magnético, óptico, de estado sólido o de otro tipo para poder manipularla en cualquier momento. Los ficheros tienen un nombre y se ubican en directorios o carpetas. Su principal ventaja es que los datos que se guardan en ellos no son volátiles.

Un **fichero** está formado por un conjunto de **registros** y cada registro por un conjunto de **campos** relacionados, que representan los datos de la entidad de la cual queremos almacenar información

1.1 Clasificación de ficheros

Según el tipo de contenido:

- **Ficheros de caracteres (o de texto):** son aquellos creados exclusivamente con caracteres (código ASCII), por lo que pueden ser creados y visualizados por cualquier editor de texto.
- **Ficheros binarios (o de bytes):** son aquellos que no contienen caracteres reconocibles, sino que los bytes que contienen representan otra información como imágenes, música o vídeo. Estos ficheros solo pueden ser abiertos por aplicaciones concretas que conozcan cómo están organizados los bytes dentro del fichero.

Según el modo de acceso:

- **Secuenciales:** la información se almacena como una secuencia de bytes o caracteres, de manera que para acceder al byte i-ésimo, es necesario pasar antes por todos los anteriores.
- **Aleatorios:** se puede acceder directamente a una posición concreta del fichero, sin necesidad de recorrer los bytes anteriores.

Según la dirección del flujo de datos:

- **De entrada:** el programa lee los datos del archivo.
- **De salida:** el programa escribe datos en el archivo.
- **De entrada/salida**: el programa puede leer o escribir datos en el archivo.

Según la longitud de registro:

- **Longitud variable**: en realidad, en este tipo de archivos no tiene sentido hablar de longitud de registro, podemos considerar cada byte como un registro. También puede suceder que nuestra aplicación conozca el tipo y longitud de cada dato almacenado en el archivo, y lea o escriba los bytes necesarios en cada ocasión. Otro caso es cuando se usa una marca para el final de registro, por ejemplo, en ficheros de texto se usa el carácter de retorno de línea para eso. En estos casos cada registro es de longitud diferente.
- **Longitud constante**: en estos archivos los datos se almacenan en forma de registro de tamaño contante.
- **Mixtos**: en ocasiones pueden crearse archivos que combinen los dos tipos de registros, por ejemplo, dBASE usa registros de longitud constante, pero añade un registro especial de cabecera al principio para definir, entre otras cosas, el tamaño y el tipo de los registros

2 OPERACIONES DE GESTIÓN DE FICHEROS

Las operaciones básicas que se realizan sobre cualquier fichero, independientemente de la forma de acceso al mismo, son las siguientes:

- **Creación del fichero.** El fichero se crea en el disco con un nombre que después se debe usar para acceder a él. Este proceso solo se hace una vez.
- **Apertura del fichero.** Para que un programa pueda operar con un fichero, primero debe realizar su apertura. Para ello, el programa utilizará algún método para identificar el fichero, como usar un descriptor del fichero.
- **Cierre del fichero.** El fichero se debe cerrar cuando el programa no lo vaya a utilizar. Suele ser la última instrucción del programa.
- **Lectura de datos del fichero.** Este proceso consiste en transferir información del fichero a la memoria, a través de variables del programa.
- **Escritura de datos en el fichero**. Este proceso consiste en transferir información de la memoria, por medio de las variables del programa, al fichero.

Una vez abierto, las operaciones típicas que se realizan sobre un fichero son:

- **Altas**: consiste en añadir un nuevo registro al fichero.
- **Bajas**: consiste en eliminar del fichero un registro ya existente.
- **Modificaciones**: consiste en cambiar parte del contenido de un registro. Antes de la modificación, será necesario localizar el registro a modificar dentro del fichero.
- **Consultas**: consiste en buscar en el fichero un registro determinado.

2.1 Clases asociadas con ficheros

En Java, la clase **`File`** proporciona un conjunto de utilidades relacionadas con ficheros que proporcionan información acerca de los mismos. Un objeto de la clase **`File`** puede representar el nombre de un fichero o de un directorio que existe en el sistema de ficheros. También se puede usar para crear un nuevo directorio o una trayectoria de directorios completa si ésta no existe.

Para crear un objeto **`File`**, se puede usar cualquiera de los siguientes constructores:

Constructor	Explicación
`File(String directorioYFichero)`	Recibe como parámetro el camino completo donde está el fichero junto con el nombre. Por defecto, si no se indica, lo busca en la carpeta del proyecto. `directorioYFichero` puede ser también el camino a un directorio, sin indicar al final el nombre de ningún fichero. En este caso se crea un directorio.
`File(String directorio, String nombreFichero)`	Recibe en la instanciación del objeto el camino completo donde está el fichero, como primer parámetro, y el nombre del fichero como segundo parámetro.
`File(File directorio, String fichero)`	Recibe en la instanciación del objeto un objeto de tipo `File`, que hace referencia a un directorio, como primer parámetro y el nombre del fichero como segundo parámetro.

La clase **`File`** tiene los siguientes métodos que sirven para ficheros y directorios:

Método	Explicación
`boolean canRead()`	Informa si se puede leer la información que contiene.
`boolean canWrite()`	Informa si se puede guardar información.
`boolean exists()`	Informa si el fichero o el directorio existe.
`boolean isFile()`	Devuelve true si el objeto File corresponde a un fichero normal.
`boolean isDirectory()`	Devuelve true si el objeto `File` corresponde a un directorio.
`long lastModified()`	Retorna la fecha de la última modificación.
`String getName()`	Devuelve el nombre del fichero o directorio.
`String getPath()`	Devuelve la ruta relativa.
`String getAbsolutePath()`	Devuelve la ruta absoluta del fichero/directorio.
`String getParent()`	Devuelve el nombre del directorio padre o null si no existe.

Algunos de los métodos que tiene la clase **`File`**, exclusivos de manejo de ficheros, son:

Método	Explicación
`boolean delete()`	Borra el fichero o directorio asociado al objeto `File`.
`long length()`	Retorna el tamaño del archivo en bytes.
`boolean renameTo(File nuevo_nombre)`	Cambia el nombre del fichero.

Algunos de los métodos que tiene la clase **`File`**, exclusivos de manejo de directorios, son:

Método	Explicación
`boolean mkdir()`	Crea un directorio con el nombre que se haya indicado en el constructor al crear el objeto `File`. Sólo lo crea si no existe.
`String[] list()`	Devuelve un array de String con los nombres de ficheros y directorios asociados al objeto `File`.
`File[] listFiles()`	Devuelve un array de objetos File que corresponden a los a los objetos `File` que están dentro del directorio representado por el objeto `File`.

Hay que tener en cuenta que se puede:

- indicar el nombre de un fichero **sin la ruta**, se buscará el fichero en el directorio actual.
- indicar el nombre de un fichero con la **ruta relativa**.
- indicar el nombre de un fichero con la **ruta absoluta**.

El siguiente programa Java muestra los nombres de los archivos y directorios que se encuentren en el directorio que se pasa como argumento a **`metodo`**.

```
import java.io.File;

public class _2x2x01
{
    public static void metodo(String[] args)
    {
        File file = new File(args[0]);

        if( file.isDirectory() )
        {
            File[] ficheros = file.listFiles();

            for( File f : ficheros )
                System.out.println(f.getName());
        }
    }

    public static void main(String[] args)
    {
        String[] argumentos = { "C:\\" };
        metodo(argumentos);
    }
}
```

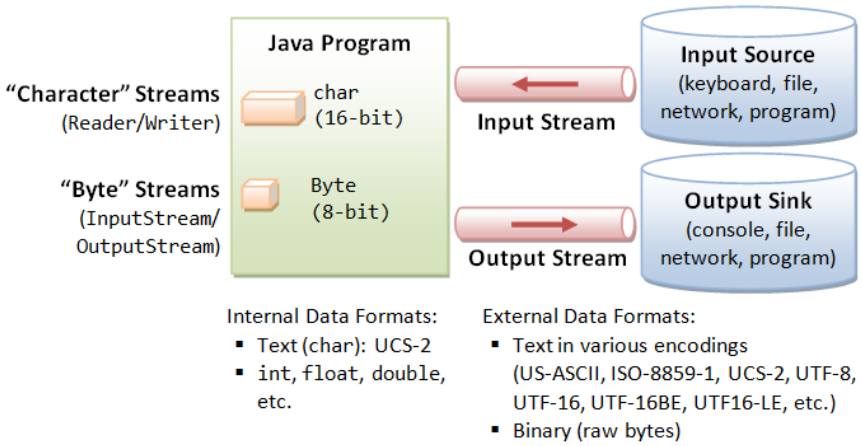
3 ACCESO SECUENCIAL

El sistema de entrada/salida en Java presenta una gran variedad de clases que se implementan en el paquete **`java.io`**. Utiliza la abstracción del flujo o **`stream`** para tratar la comunicación de información entre una fuente y un destino, y por tanto, las operaciones que un programa realiza sobre un flujo son independientes del dispositivo al que esté asociado.

Así pues, un **archivo** es simplemente un flujo externo, es decir, una secuencia de bytes almacenados en un dispositivo externo. Los programas leen o escriben en el flujo, que puede estar conectado a un dispositivo o a otro.

Hay dos tipos de flujos de datos definidos:

- **Flujos de caracteres (16 bits):** realizan operaciones de entradas y salidas de caracteres. El flujo de caracteres viene gobernado por las clases **`Reader`** y **`Writer`**.
- **Flujos de bytes (8 bits):** realizan operaciones de entradas y salidas de bytes y su uso está orientado a la lectura y escritura de datos binarios. Todas las clases de flujos de bytes heredan de las clases **`InputStream`** y **`OutputStream`**.



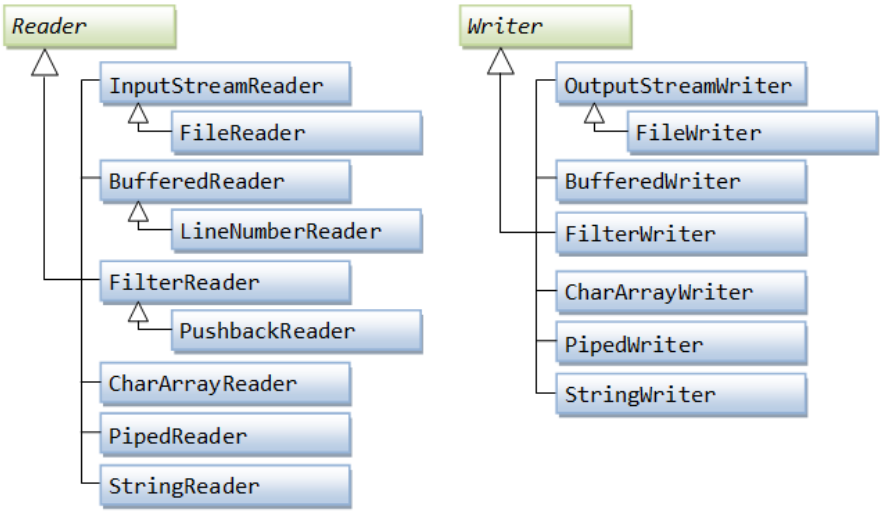
En Java, la entrada desde el teclado y la salida por la pantalla están gestionadas por la clase **`System`**. Esta clase pertenece al paquete **`java.lang`** y tienen tres atributos, los llamados flujos predefinidos: **`in`**, **`out`** y **`err`**, que son **`public`** y **`static`**:

- **`System.in`**: hace referencia a la entrada estándar de datos (teclado).
- **`System.out`**: hace referencia a la salida estándar de datos (pantalla).
- **`System.err`**: hace referencia a la salida estándar de información de errores (pantalla).

3.1 FLUJOS DE CARACTERES

Las clases abstractas **`Reader`** y **`Writer`** manejan flujos de caracteres `Unicode`. Hay ocasiones en las que hay que usar las clases que manejan bytes en combinación con las clases que manejan caracteres. Para lograr esto, existen varias clases "puente" (que convierten los flujos de bytes a flujos de caracteres): **`InputStreamReader`** convierte un **`InputStream`** en un **`Reader`** (lee bytes y los convierte a caracteres) y **`OutputStreamWriter`** convierte un **`OutputStream`** en un **`Writer`**.

La siguiente figura muestra la jerarquía de clases para la lectura y la escritura de flujos de caracteres.



Las clases de flujos de caracteres más importantes son:

- Para acceso a ficheros, lectura y escritura de caracteres en ficheros: **`FileReader`** y **`FileWriter`**.
- Para acceso a caracteres, leen y escriben un flujo de caracteres en un array de caracteres: **`CharArrayReader`** y **`CharArrayWriter`**.
- Para buferización de datos, **`BufferedReader`** y **`BufferedWriter`**, las cuales se utilizan para evitar que cada lectura o escritura acceda directamente al fichero, ya que usan un buffer intermedio entre la memoria y el flujo de datos.

3.1.1 Las Clases FileReader y FileWriter

Los ficheros de texto, que normalmente se generan con un editor de textos, almacenan caracteres alfanuméricos en un formato estándar (ASCII, Unicode, UTF-8, etc.). Para trabajar con ellos en Java, se utilizan la clase **`FileReader`** para leer caracteres y la clase **`FileWriter`** para escribir los caracteres en el fichero. Además, las lecturas o escrituras realizadas sobre un fichero deben escribirse dentro de un manejador de excepciones try-catch.

Al instanciar un objeto con la clase **`FileReader`**, el programa abre el fichero que se envía como argumento para lectura y su información se puede leer de forma secuencial carácter a carácter. Si el fichero no existe o no es válido, se lanza la excepción FileNotFoundException.

Constructores de la clase `FileReader`:

Método	Explicación
`FileReader(File fichero)`	Lanza la excepción `FileNotFoundException` si el fichero pasado no existe.
`FileReader(String fichero)`	Lanza la excepción `FileNotFoundException` si el fichero pasado no existe.

Principales métodos de la clase FileReader:

Método	Explicación
`int read()`	Lee un carácter y lo devuelve.
`int read(char[] buf)`	Lee hasta buf.length caracteres de datos del array buf pasado como parámetro.
`int read(char[] buf, int desplazamiento, int n)`	Lee hasta n caracteres de datos del array buf comenzando por buf[desplazamiento] y devuelve el número leído de caracteres.

Al instanciar un objeto con la clase **`FileWriter`**, el programa abre el fichero especificado con el fin de guardar información, pero carácter a carácter. Si el fichero no existe, el programa lo crea de forma automática. Si el disco está lleno o protegido contra escritura, se lanza la excepción **`IOException`**.

Constructores de la clase FileWriter:

Constructor	Explicación
`FileWriter(String nombreFich)`	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileWriter(File fichero)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabaja y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileWriter(String nombreDeFich, boolean append)`	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
`FileWriter(File fichero, boolean append)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabaja y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Principales métodos de la clase FileWriter:

Método	Explicación
`void write(int c)`	Escribe un carácter.
`void write(char[] buf)`	Escribe un array de caracteres.
`void write(char[] buf, int desplazamiento, int n)`	Escribe n caracteres de datos de un array buf comenzando por buf[desplazamiento].
`void write(String str)`	Escribe una cadena de caracteres.
`append(char c)`	Añade un carácter a un fichero.

```
import java.io.*;
public class Ejemplo01 {

	public static void main(String args[]) throws IOException{
		String cadena;
		int codCaracter;
		char caracter;

		FileWriter fichEsc = new FileWriter("nuevo.txt");
		cadena = Leer.pedirCadena("\nIntroduce una frase: ");
		fichEsc.write(cadena);
		fichEsc.close();

		FileReader fichLect = new FileReader("nuevo.txt");
		cadena = "";
		codCaracter = fichLect.read();
		while(codCaracter!=-1){
			caracter = (char) codCaracter;
			cadena = cadena + caracter;
			codCaracter = fichLect.read();
		}
		fichLect.close();
		System.out.println("\nLa frase leída del fichero es: \"" + cadena + "\"");
	}
  }
```

3.1.2 Las Clases BufferedReader y BufferedWriter

Las clases **`BufferedReader`** y **`BufferedWriter`** se pueden utilizar sobre las clases **`FileReader`** y **`FileWriter`** u otros flujos de caracteres para realizar operaciones de entrada/salida con un buffer, en lugar de carácter a carácter.

La clase **BufferedReader** dispone del método **readLine**(), que lee una línea del fichero y la devuelve, o devuelve **null** si no hay nada que leer o se llega al final del fichero. Para construir un objeto **BufferedReader**, se necesita la clase **FileReader**:

```
FileReader fileR = new FileReader(nombreFichero);
BufferedReader bufferedR = new BufferedReader(fileR);
```

```
import java.io.*;
public class Ejemplo02 {

	public static void main(String[] args) throws IOException{
		String cadena;
		File fich = new File("fichero.txt");
		BufferedReader flujo = new BufferedReader(new FileReader(fich));
		if (fich.exists()){
			System.out.println("\nEsta es la información que contiene el
					fichero: ");
			cadena = flujo.readLine();
			while(cadena!=null){
				System.out.println(cadena);
				cadena = flujo.readLine();
			}
		}
	}
}
```

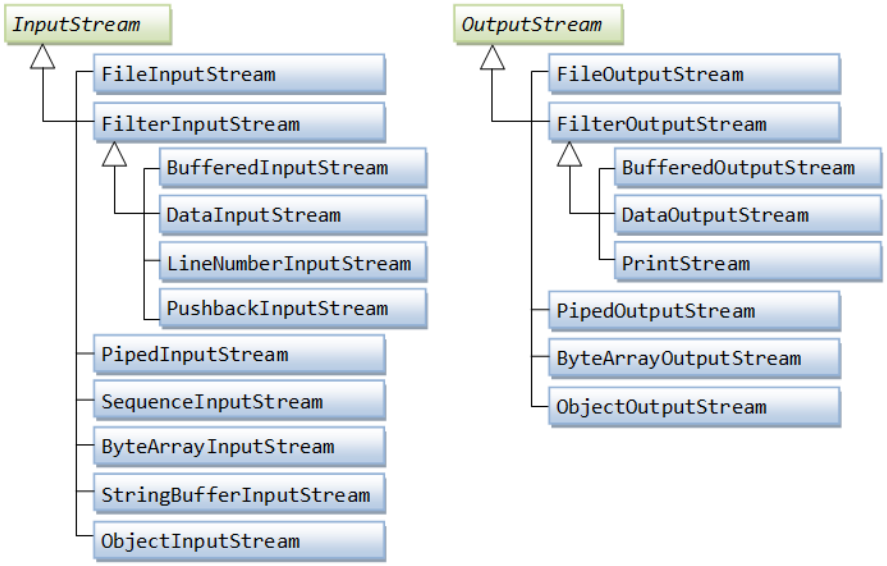
La clase **BufferedWriter** dispone del método **write**(), que escribe una línea en el fichero, y del método **newLine**(), que escribe un salto de línea en el fichero. Para construir un objeto **BufferedWriter**, se necesita la clase **FileWriter**:

```
FileWriter fileW = new FileWriter(nombreFichero);
BufferedWriter bufferedW = new BufferedWriter(fileW);
```

3.2 FLUJOS DE BYTES

La clase abstracta **InputStream** representa las clases que producen entradas de distintas fuentes, como un array de bytes, un objeto **String**, un fichero, una tubería, etc. La clase abstracta **OutputStream** decide donde irá la salida, como a un array de bytes, un fichero o una tubería.

La siguiente figura muestra la jerarquía de clases para la lectura y la escritura de flujos de bytes.



Las clases **FileInputStream** y **FileOutputStream** manipulan los flujos de bytes que provienen o se dirigen hacia ficheros en disco.

3.2.1 Las clases FileInputStream y FileOutputStream

Los ficheros binarios almacenan secuencias de dígitos binarios que no son legibles directamente por el usuario. Tienen la ventaja de que ocupan menos espacio en disco. En Java, las dos clases que permiten trabajar con ficheros son **`FileInputStream`** (para entrada) y **`FileOutputStream`** (para salida), que operan con flujos de bytes y crean un enlace entre el flujo de bytes y el fichero.

Cuando se instancia un objeto con la clase **`FileInputStream`**, el programa abre el fichero (que se debe enviar como argumento del constructor) en modo lectura. Una vez abierto, se podrá leer la información que contiene de forma secuencial byte a byte.

Constructores de la clase **`FileInputStream`**:

Constructor	Explicación	Excepción que lanza
`FileInputStream(String nombreDeFichero)`	Recibe como parámetro el nombre del fichero a abrir.	`FileNotFoundException` si el fichero no existe.
`FileInputStream(File fichero)`	Recibe como parámetro un objeto File que representa al fichero con el que queremos trabajar.	`FileNotFoundException` si el fichero no existe.

Principales métodos de la clase **`FileInputStream`**:

Método	Explicación
`int read()`	Devuelve el código ASCII del siguiente byte que hay después de donde está situado el puntero del fichero. Dicho puntero se va moviendo secuencialmente por el fichero, según vamos leyendo los bytes. Devuelve -1 si no hay ningún byte más que leer.
`int read(byte cadByte[])`	Lee hasta `cadByte.length` bytes guardándolos en la tabla que se envía como parámetro. Devuelve -1 si no hay ningún byte más que leer.
`void close()`	Cierra el fichero.

```
import java.io.*;
public class Ejemplo03 {

	public static void main(String args[]) throws IOException{
		int c;
		try{
			//Se puede poner / o  \\
			FileInputStream f = new FileInputStream("/pedidos.txt");
			/*Al poner / va a buscar el fichero en la raíz del disco donde está
			  el proyecto. Si no se pone la /, busca el fichero en la carpeta
			  donde está ahora mismo*/
			//FileInputStream f = new FileInputStream("pedidos.txt");
			while((c=f.read())!=-1)
				System.out.print((char) c);
			/*Si no se hace la conversión visualizaría el código ASCII
			de cada carácter que hayguardado en el fichero*/
			f.close();
		}catch(FileNotFoundException e){
			System.out.println("El fichero no existe. ");
		}
	}
  }
```

Cuando se instancia un objeto con la clase **`FileOutputStream`**, lo que hace el programa es abrir el fichero para escritura. Una vez abierto, se podrá guardar información byte a byte. Si el fichero no existe, se crea en ese momento.

Constructores de la clase **`FileOutputStream`**:

Constructor	Explicación
`FileOutputStream(String nombreFich)`	Recibe como parámetro el nombre del fichero a abrir y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileOutputStream(File fichero)`	Recibe como parámetro un objeto `File` que representa al fichero con el que queremos trabaja y borra el contenido previo del fichero comenzando a escribir desde el principio.
`FileOutputStream(String nombreDeFich, boolean append)`	Recibe como parámetro el nombre del fichero a abrir y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.
`FileOutputStream(File fichero, boolean append)`	Recibe como parámetro un objeto `File` que representa al fichero con el que queremos trabaja y, si **append** es **true**, se sitúa al final del fichero para añadir contenido desde el final.

Principales métodos de la clase **`FileOutputStream`**:

Método	Explicación
`int write(int byte)`	Escribe el byte que recibe como argumento en el fichero.
`int write(byte cadByte[])`	Escribe todos los bytes que contiene la tabla cadByte.
`void close()`	Cierra el fichero.

```
  import java.io.*;
public class Ejemplo04 {

      public static void main(String args[]) throws IOException{
            String cadena;
            int codCaracter;
            char caracter;

            FileOutputStream f1 = new FileOutputStream("fichejemplo04.txt ");
            cadena=Leer.pedirCadena("Introduce una frase: ");
            for(int pos=0;pos<cadena.length();pos++){
                  f1.write(cadena.charAt(pos));
            }
            f1.write('\n');//Para separar las frases
            f1.close();

            System.out.println(" \nEl contenido del fichero es: ");

            FileInputStream f2 = new FileInputStream("fichejemplo04.txt ");
            codCaracter=f2.read();
            while(codCaracter!=-1){
                  caracter=(char)codCaracter;
                  System.out.print(caracter);
                  codCaracter=f2.read();
            }
            f2.close();
      }
   }
```

3.2.2 Las clases DataInputStream y DataOutputStream

Para leer y escribir datos de tipos primitivos (como `boolean`, `int`, `long`, `float`, `double`, `char`, etc.), se utilizan las clases **`DataInputStream`** y **`DataOutputStream`**. Además de los métodos **`read`**() y **`write`**(), estas clases

proporcionan métodos para la lectura y escritura de tipos primitivos de un modo independiente de la máquina. Algunos de los métodos disponibles son:

Métodos para lectura	Métodos para escritura
boolean readBoolean()	void writeBoolean(boolean b)
byte readByte()	voidwriteByte(int i)
int readUnsignedByte()	void writeBytes(String s)
int readUnsignedShort()	void writeShort(int i)
short readShort()	void writeChars(String s)
char readChar()	void writeChar(int i)
int readInt()	void writeInt(int i)
long readLong()	void writeLong(long l)
float readFloat()	void writeFloat(float f)
double readDouble()	void writeDouble(double d)
String readUTF()	void writeUTF(String s)

Para abrir un objeto **DataInputStream**, se utilizan los mismos métodos que para **FileInputStream**. Por ejemplo:

```
File fichero = new File("C:\\EJERCICIOS\\FichData.dat");
FileInputStream fileIS = new FileInputStream(fichero);
DataInputStream dataIS = new DataInputStream(fileIS);
```

Para abrir un objeto **DataOutputStream**, se utilizan los mismos métodos que para **FileOutputStream**. Por ejemplo:

```
File fichero = new File("C:\\EJERCICIOS\\FichData.dat");
FileOutputStream fileOS = new FileOutputStream(fichero);
DataOutputStream dataOS = new DataOutputStream(fileOS);
```

Hay que tener mucho cuidado con leer un fichero en el mismo formato que se ha escrito porque, de no ser así, daría errores en la ejecución al no corresponderse los tipos.

Para saber que se ha alcanzado el final del fichero, los métodos lanzan la excepción **EOFException**, así que hay que recogerla y tratarla correctamente. Ejemplo:

```
import java.io.*;
public class Ejemplo05 {

	public static void main(String args[]) throws FileNotFoundException{
		int numInt;
		String cadena;
		float numFloat;

		FileOutputStream fichEscrit = new FileOutputStream("prueba.dat");
		DataOutputStream escritTipos = new DataOutputStream(fichEscrit);

		FileInputStream fichLect = new FileInputStream("prueba.dat");
		DataInputStream lectTipos = new DataInputStream(fichLect);

		try {
			numInt=Leer.pedirEntero("Inserta un número entero:");
			escritTipos.writeInt(numInt);

			numFloat=Leer.pedirFloat("Inserta un número con decimales: ");
			escritTipos.writeFloat(numFloat);

			cadena=Leer.pedirCadena("Inserta una cadena:");
			escritTipos.writeChars(cadena);
			escritTipos.close();

			numInt=lectTipos.readInt();
			numFloat=lectTipos.readFloat();
			cadena=lectTipos.readLine();

			System.out.println(" \nLos datos leídos del fichero son: \n");
```

```
                System.out.println("-**:" +numInt+"-**:" +numFloat+"-**:" +cadena);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

3.2.3 Las clases ObjectInputStream y ObjectOutputStream

Se llama **persistencia** al proceso de almacenar toda la información que contiene un objeto, manteniendo su estructura, en un fichero binario. En Java, para poder guardar un objeto en un fichero binario, dicho objeto tiene que implementar la interfaz **`Serializable`**, que dispone de métodos que permiten escribir y leer objetos en ficheros binarios:

Método para escribir un objeto	**`void writeObject(ObjectOutputStream stream) throws IOException`**
Método para leer un objeto	**`void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`**

La **serialización** de objetos de Java permite tomar cualquier objeto que implemente la interfaz `Serializable` y convertirlo en una secuencia de bits, que puede ser posteriormente restaurada para regenerar el objeto original.

Para leer y escribir objetos serializables a un stream se utilizan las clases **`ObjectInputStream`** y **`ObjectOutputStream`** respectivamente.

Por ejemplo, para escribir un objeto "persona" en un fichero binario, se necesita crear un flujo de salida a disco con FileOutputStream, y a continuación, crear el flujo de salida **ObjectOutputStream**, que procesa los datos y está vinculado al fichero. Por último, el método writeObject() escribe el objeto al flujo de salida y lo guarda en el fichero.

```
File fichero = new File("C:\\EJERCICIOS\\FichPersona.dat");
FileOutputStream fileOS = new FileOutputStream(fichero);
ObjectOutputStream objectOS = new ObjectOutputStream(fileOS);
objectOS.writeObject(persona);
```

Por ejemplo, para leer un objeto "persona" de un fichero binario, se necesita crear el flujo de entrada a disco **`FileInputStream`**, y a continuación, crear el flujo de entrada **`ObjectInputStream`**, que procesa los datos y está vinculado al fichero. Por último, el método **`readObject`**() lee el objeto del flujo de entrada y puede lanzar la excepción **`ClassNotFoundException`**.

```
File fichero = new File("C:\\EJERCICIOS\\FichPersona.dat");
FileInputStream fileIS = new FileInputStream(fichero);
ObjectInputStream objectIS = new ObjectInputStream(fileIS);
persona = (Persona) objectIS.readObject();
```

4 ACCESO ALEATORIO

Hasta ahora, todas las operaciones sobre ficheros se han realizado de forma secuencial. Java dispone de la clase **`RandomAccessFile`** para posicionar un cursor en una posición concreta de un fichero y acceder a su contenido de forma aleatoria o directa (no secuencial). Esta clase no es parte de la jerarquía **`Reader/Writer`** ni **`InputStream/OutputStream`**, puesto que permite avanzar y retroceder dentro de un fichero.

Constructores de la clase **`RandomAccessFile`**:

Método	Explicación
`RandomAccessFile(String nombre, String modo)`	nombre es el nombre del fichero y modo es el argumento que determina si el contenido del fichero se va a poder solo leer (**"r"**) o leer y escribir (**"rw"**).
`RandomAccessFile(File fichero, String modo)`	fichero es el objeto fichero y modo es el argumento que determina si el contenido del fichero se va a poder solo leer (**"r"**) o leer y escribir (**"rw"**).

La clase **`RandomAccessFile`** maneja un puntero o cursor que indica la posición actual en el fichero. Cuando se abre un fichero, el puntero se coloca en 0, es decir, apuntando al principio del mismo. Además, esta clase implementa las **interfaces `DataInput`** y **`DataOutput`**. Por tanto, una vez abierto un fichero, se pueden utilizar sus métodos **`readXXX`***()* y **`writeXXX`***()* para cada tipo de dato y las sucesivas llamadas a estos métodos **`read`***()* y **`write`***()* ajustan el puntero según la cantidad de bytes leídos o escritos.

Principales métodos de la clase **`RandomAccessFile`**:

Método	Explicación
`long getFilePointer()`	Devuelve la posición actual del puntero del fichero.
`void seek(longposicion)`	Coloca el puntero del fichero en una posición determinada desde el comienzo del mismo.
`long length()`	Devuelve el tamaño del fichero en bytes y marca el final del fichero.
`int skipBytes(int desplazamiento)`	Desplaza el puntero del fichero desde la posición actual el número de bytes indicados en desplazamiento.

Cada tipo de dato tiene un tamaño concreto:

`boolean` (1 bit)	**`byte`** (1 byte)	**carácter `Unicode`** (2 bytes)
`short` (2 bytes)	**`int`** (4 bytes)	**`long`** (8 bytes)
`float` (4 bytes)	**`double`** (8 bytes)	

El siguiente programa Java abre un fichero para lectura y escritura, escribe los nombres que se leen por teclado en el fichero al final, y después los lee del fichero para mostrarlos en pantalla.

```
import java.io.*;
public class Ejemplo06 {

	public static void main(String args[]) throws IOException{
		RandomAccessFile fichAleatorio;
		String nomNuevo,nombre;
		fichAleatorio = new RandomAccessFile("directo06.txt","rw");

		nomNuevo=Leer.pedirCadena("Introduce un nombre (\"*\" para salir):");
		while (!nomNuevo.equals("*")){
			fichAleatorio.seek(fichAleatorio.length());
			fichAleatorio.writeBytes(nomNuevo);
			fichAleatorio.write('\n');//Para separar los nombres
			nomNuevo=Leer.pedirCadena("Introduce nombre (\"*\" para salir):");
		}

		fichAleatorio.seek(0);
		nombre=fichAleatorio.readLine();
		while(nombre!=null) {
			System.out.println("Nombre leído: " +nombre);
			nombre=fichAleatorio.readLine();
		}
		fichAleatorio.close();
	}
 }
```

El siguiente ejemplo inserta datos de empleados (apellido, departamento y salario) en un fichero aleatorio. Estos datos se introducen de manera secuencial (no se usa el método *seek()*), y además, por cada empleado, se inserta un identificador que coincide con el índice+1 con el que se recorren los vectores. La longitud del registro de cada empleado es la misma (36 bytes).

```
import java.io.*;
public class EjemCreaEmpleAleatorio {

	public static void main(String[] args) throws IOException {
		File fichero = new File("AleatorioEmple.dat");
		//declara el fichero de acceso aleatorio
		RandomAccessFile file = new RandomAccessFile(fichero, "rw");
		//arrays con los datos
		String apellido[] = {"FERNANDEZZZZZ","GIL","LOPEZ","MARTINEZ",
				"SEVILLA","CEREZO", "REY"}; //apellidos
		int dep[] = {10, 20, 10, 10, 30, 30, 20};	//departamentos
		Double salario[]={1000.45, 2400.60, 3000.0, 1500.56,
				2200.0, 1435.87, 2000.0};//salarios

		StringBuffer buffer = null;//buffer para almacenar apellido
		int n=apellido.length;//numero de elementos del array

		for (int i=0;i<n; i++){ //recorro los arrays
			file.writeInt(i+1); //uso i+1 para identificar empleado
			buffer = new StringBuffer( apellido[i] );
			buffer.setLength(10); //10 caracteres para el apellido
			file.writeChars(buffer.toString());//insertar apellido
			file.writeInt(dep[i]);	//insertar departamento
			file.writeDouble(salario[i]);//insertar salario
		}
		file.close();  //cerrar fichero
	}

 }
```

El siguiente ejemplo usa el fichero anterior y visualiza todos sus registros. El posicionamiento para empezar a recorrer los registros empieza en 0, y para recuperar los siguientes registros hay que sumar 36 (tamaño del registro) a la variable utilizada para el posicionamiento.

```
import java.io.*;
public class EjemLeeEmpleAleatorio {

	public static void main(String[] args) throws IOException {
		File fichero = new File("AleatorioEmple.dat");
		//declara el fichero de acceso aleatorio
		RandomAccessFile file = new RandomAccessFile(fichero, "r");
		//
		int  id, dep, posicion;
		Double salario;
		char apellido[] = new char[10], aux;

		posicion = 0;  //para situarnos al principio
		file.seek(posicion); //nos posicionamos en posicion

		//recorro el fichero
		while(file.getFilePointer() < file.length()){  //Si he recorrido todos los bytes
salgo del bucle
			id = file.readInt();   // obtengo id de empleado
			//recorro uno a uno los caracteres del apellido
			for (int i = 0; i < apellido.length; i++) {
				aux = file.readChar();
				apellido[i] = aux;    //los voy guardando en el array
			}

			//convierto a String el array
			String apellidos = new String(apellido);
			dep = file.readInt();        //obtengo dep
			salario = file.readDouble(); //obtengo salario

			System.out.printf("ID: %s, Apellido: %s, Departamento: %d, Salario: %.2f %n",
					id,   apellidos.trim(), dep, salario);

			//me posiciono para el sig empleado, cada empleado ocupa 36 bytes
			posicion= posicion + 36;
			file.seek(posicion); //nos posicionamos en posicion
		}//fin bucle while
		file.close();  //cerrar fichero
	}

}
```

Para consultar un empleado determinado no es necesario recorrer todos los registros del fichero. Conociendo su identificador, se puede acceder a la posición que ocupa dentro del fichero y obtener sus datos.

5 FICHEROS XML

XML (eXtensible Markup Language) es un metalenguaje, es decir, un lenguaje para la definición de lenguajes de marcas. Permite jerarquizar y estructurar la información así como describir los contenidos dentro del propio documento, de forma independiente del lenguaje de programación y del sistema operativo empleados.

Los ficheros XML son ficheros de textos escritos en lenguaje XML donde la información está organizada de forma secuencial y en orden jerárquico. Existe una serie de marcas especiales, como los símbolos "<" y ">", que se usan para delimitar las marcas que dan la estructura al documento. Cada marca tiene un nombre y puede tener atributos.

El uso intensivo de los ficheros XML en el desarrollo de aplicaciones hace que sean necesarias herramientas específicas (librerías) para acceder y manipular este tipo de archivos de forma potente, flexible y eficiente. Estas herramientas reducen los tiempos de desarrollo de aplicaciones y optimizan los propios accesos a XML, sin cargar innecesariamente el sistema.

```
▼<Libros xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="LibrosEsquema.xsd">
  ▼<Libro publicado_en="1840">
     <Titulo>El Capote</Titulo>
     <Autor>Nikolai Gogol</Autor>
   </Libro>
  ▼<Libro publicado_en="2008">
     <Titulo>El Sanador de Caballos</Titulo>
     <Autor>Gonzalo Giner</Autor>
   </Libro>
  ▼<Libro publicado_en="1981">
     <Titulo>El Nombre de la Rosa</Titulo>
     <Autor>Umberto Eco</Autor>
   </Libro>
 </Libros>
```

Un **analizador sintáctico** o **parser** es una herramienta que lee un fichero XML y que comprueba si el documento es válido sintácticamente. Un parser XML es un módulo, biblioteca o programa encargado de leer un fichero XML y acceder a su contenido y estructura, mediante un modelo interno que optimiza su acceso.

Algunos de los analizadores sintácticos más empleados son DOM y SAX, los cuales son independientes del lenguaje de programación, tienen implementaciones particulares para Java, C, VisualBasic y .NET, y utilizan dos enfoques muy distintos:

- **DOM (Document Object Model).** Almacena toda la estructura del documento en memoria en forma de árbol, con nodos padre, nodos hijo y nodos finales (no tienen descendientes). Una vez creado el árbol, se recorren los diferentes nodos y se analiza a qué tipo particular pertenecen. Este procesamiento necesita más recursos de memoria y tiempo, sobre todo si los ficheros XML son grandes y complejos.
- **SAX (Simple API for XML).** Lee un fichero XML de forma secuencial y produce una secuencia de eventos (comienzo/fin del documento, comienzo/fin de una etiqueta, etc.) en función de los resultados de lectura. Cada evento invoca a un método definido por el programador. Este procesamiento consume poca memoria, pero impide tener una visión global del fichero.

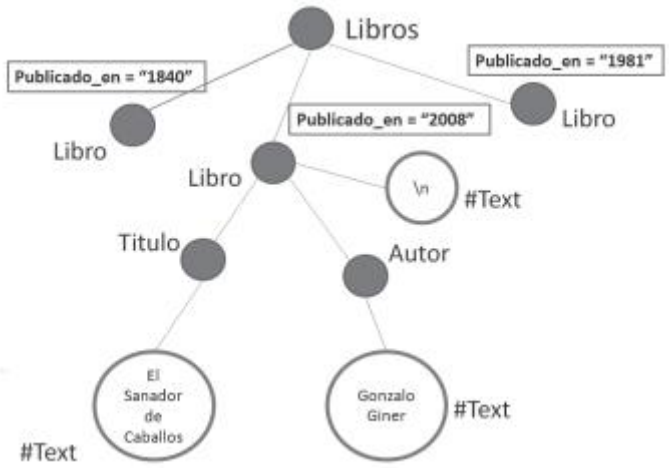
Estas herramientas DOM y SAX no validan los documentos XML. Es decir, solo comprueban que el documento está bien formado según la definición XML, pero no necesitan de un esquema asociado para comprobar si es válido con respecto a ese esquema.

5.1 Acceso a datos con DOM

DOM (Document Object Model) es una herramienta de programación que permite analizar y manipular dinámicamente y de manera global el contenido, el estilo y la estructura de un documento XML. Tiene su origen en el Consorcio World Wide Web (W3C).

A partir de un fichero XML, se almacena en memoria en forma de árbol un modelo del fichero, con nodos padre, nodos hijo y nodos finales (que son aquellos que no tienen descendientes). En este modelo, todas las estructuras de datos del fichero XML se transforman en algún tipo de nodo, y después, se organizan dichos nodos jerárquicamente en forma de árbol para representar la estructura descrita por el fichero XML.

Una vez creada en memoria esta estructura, los métodos de DOM permiten recorrer los distintos nodos del árbol y analizar a qué tipo particular pertenecen. En función del tipo de nodo, DOM ofrece una serie de funcionalidades para trabajar con la información contenida.



Para trabajar con DOM en Java, se necesitan las clases e interfaces que componen el paquete **org.w3c.dom** y el paquete **`javax.xml.parsers`** de la API estándar de Java. Estas clases ofrecen métodos para cargar documentos desde una fuente de datos (fichero, **`InputStream`**, etc.). Contiene dos clases fundamentales: **`DocumentBuilderFactory`** y **`DocumentBuilder`**.

DOM no define ningún mecanismo para generar un fichero XML a partir de un árbol DOM. Para eso se usa el paquete **`javax.xml.transform`**, que permite especificar una fuente y un resultado. La fuente y el resultado pueden ser ficheros, flujos de datos o nodos DOM entre otros.

Las principales interfaces que necesita un programa Java que utiliza DOM son:

- **Document**. Es un objeto que equivale a un ejemplar de un documento XML. Permite crear nuevos nodos en el documento.
- **Element**. Cada elemento del documento XML tiene un equivalente en un objeto de este tipo. Expone propiedades y métodos para manipular los elementos del documento y sus atributos.
- **Node**. Representa a cualquier nodo del documento.
- **NodeList**. Contiene una lista con los nodos hijos de un nodo.
- **Attr**. Permite acceder a los atributos de un nodo.
- **Text**. Son los datos carácter de un elemento.
- **CharacterData**. Representa a los datos carácter presentes en el documento. Proporciona atributos y métodos para manipular los datos de caracteres.
- **DocumentType**. Proporciona información contenida en la etiqueta <!DOCTYPE>.

El siguiente programa Java crea un fichero XML "empleados.xml" a partir de un fichero aleatorio de empleados y muestra el documento XML resultante en pantalla:

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class CrearEmpleadoXml {
    public static void main(String args[]) throws IOException{
        File fichero = new File("AleatorioEmple.dat");
        RandomAccessFile file = new RandomAccessFile(fichero, "r");
        int  id, dep, posicion=0; //para situarnos al principio del fichero
        Double salario;
        char apellido[] = new char[10], aux;
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

        try{
            DocumentBuilder builder = factory.newDocumentBuilder();
            DOMImplementation implementation = builder.getDOMImplementation();
            Document document = implementation.createDocument(null, "Empleados", null);
            document.setXmlVersion("1.0");

            for(;;) {
                file.seek(posicion); //nos posicionamos
                id=file.readInt();   // obtengo id de empleado
                for (int i = 0; i < apellido.length; i++) {
                    aux = file.readChar();
                    apellido[i] = aux;
                }
                String apellidos = new String(apellido);
                dep = file.readInt();
                salario = file.readDouble();

                if(id > 0) { //id validos a partir de 1
                    Element raiz = document.createElement("empleado"); //nodo empleado
                    document.getDocumentElement().appendChild(raiz);
                    CrearElemento("id",Integer.toString(id), raiz, document); //añadir ID
                    CrearElemento("apellido",apellidos.trim(), raiz, document); //Apellido
                    CrearElemento("dep",Integer.toString(dep), raiz, document); //añadir DEP
                    CrearElemento("salario",Double.toString(salario), raiz, document); //añadir salario
                }
                posicion = posicion + 36; // me posiciono para el sig empleado
                if (file.getFilePointer() == file.length()) break;
            }//fin del for que recorre el fichero

            Source source = new DOMSource(document);
            Result result = new StreamResult(new java.io.File("Empleados.xml"));
            Transformer transformer = TransformerFactory.newInstance().newTransformer();
            transformer.transform(source, result);
        }
        catch(Exception e){ System.err.println("Error: "+ e); }
        file.close();  //cerrar fichero
    }//fin de main

    //Inserción de los datos del empleado
    static void  CrearElemento(String datoEmple, String valor, Element raiz, Document document){
        Element elem = document.createElement(datoEmple);
        Text text = document.createTextNode(valor); //damos valor
        raiz.appendChild(elem); //pegamos el elemento hijo a la raiz
        elem.appendChild(text); //pegamos el valor
    }
}//fin de la clase
```

En este programa Java se realizan los siguientes pasos:

1) En primer lugar, se crea una instancia de ***DocumentBuilderFactory*** para construir el analizador sintáctico (parser). Se debe encerrar en un bloque **try-catch** porque se puede producir la excepción ***ParserConfigurationException***.
2) Luego se crea un documento vacío de nombre ***document*** con el nodo raíz de nombre*Empleados* y se le asigna la versión de XML.
3) El siguiente paso consiste en recorrer el fichero con los datos de los empleados y para cada registro se crea un nodo **empleado** con 4 hijos (**id**, **apellido**, **dep** y **salario**):

- Para crear un elemento se utiliza el método ***createElement(String)***, que lleva como parámetro el nombre que se pone entre las etiquetas < y >.
- A continuación se añaden los hijos de ese nodo (raíz) con la función definida ***crearElemento(String, String, Element, Document)***. Esta función crea el nodo hijo (*<id>, <apellido>, <dep>* o *<salario>*), lo añade a la raíz (*<empleado>*) y le asigna un valor (texto).

4) Por último, se crea la fuente XML a partir del documento y se crea el resultado en el fichero "empleados.xml". Se utiliza una instancia de ***TransformerFactory*** para realizar la transformación del documento a fichero.

El siguiente programa Java lee el fichero XML "empleados.xml" y lo visualiza en pantalla:

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

public class LecturaEmpleadoXml  {

    public static void main(String[] args) {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document document = builder.parse(new File("Empleados.xml"));
            document.getDocumentElement().normalize();
            System.out.println ("Elemento raíz: " +
                document.getDocumentElement().getNodeName());

            // crear una lista con todos los nodos empleado
            NodeList empleados = document.getElementsByTagName("empleado");
            System.out.println("Nodos empleado a recorrer: " + empleados.getLength());

            // recorrer la lista de empleados
            for (int i = 0; i < empleados.getLength(); i++) {
                Node emple = empleados.item(i); // obtener un nodo empleado
                if (emple.getNodeType() == Node.ELEMENT_NODE) { // tipo de nodo
                    // obtener los elementos del nodo
                    Element elemento = (Element) emple;
                    System.out.println("ID: " + getNodo("id", elemento));
                    System.out.println("Apellido: " + getNodo("apellido", elemento));
                    System.out.println("Departamento: " + getNodo("dep", elemento));
                    System.out.println("Salario: " + getNodo("salario", elemento));
                }
            }
        }
        catch (Exception e) { e.printStackTrace(); }
    } // fin de main

    // obtener la información de un nodo
    private static String getNodo(String etiqueta, Element elem) {
        NodeList nodos = elem.getElementsByTagName(etiqueta).item(0).getChildNodes();
        Node nodo = (Node) nodos.item(0);
        return nodo.getNodeValue(); // devolver el valor del nodo
    }

} // fin de la clase
```

En este programa Java se realizan los siguientes pasos:

1) Se crea una instancia de ***DocumentBuilderFactoy*** para construir el analizador sintáctico (parser) y se carga el documento XML con el método *parse()*.
2) Se obtiene la lista de nodos con nombre *empleado* de todo el documento mediante el método ***getElementsByTagName(String)***.
3) Se realiza un bucle para recorrer esta lista de nodos *empleados*. Para cada nodo, se obtienen su etiqueta y su valor llamando a la función definida ***getNodo(String, Element)***,y se muestran en pantalla.

5.2 Acceso a datos con SAX

SAX (Simple API for XML) es otra herramienta de programación que permite el acceso a documentos XML desde lenguajes de alto nivel. Dado que aborda el procesamiento de datos desde una óptica diferente a la de DOM, por lo general, SAX se usa cuando la información almacenada en un documento XML es clara, está bien estructurada y no se necesita hacer modificaciones.

La principales características de SAX son:

- SAX ofrece una alternativa para leer documentos XML de manera secuencial. El documento solo se lee una vez: se recorre el secuencialmente el fichero hasta el final. A diferencia de DOM, el programador no se puede mover por el documento a su antojo.
- SAX no carga el documento en memoria, sino que lo lee directamente desde el fichero. Esto es especialmente útil cuando el fichero XML es muy grande.

Para trabajar con SAX en Java, se necesitan las clases e interfaces que componen el paquete **`org.xml.sax`**. Estas clases ofrecen métodos para leer secuencialmente documentos desde una fuente de datos (fichero, **`InputStream`**, etc.) y generar eventos relacionados. Contiene una clase fundamental: **`InputSource`**.

Cuando SAX detecta un elemento propio de XML, entonces lanza un evento, que puede deberse a que:

- Se haya detectado el comienzo del documento XML.
- Se haya detectado el final del documento XML.
- Se haya detectado una etiqueta de comienzo de un elemento. Por ejemplo <libro>.
- Se haya detectado una etiqueta de final de un elemento. Por ejemplo </libro>.
- Se haya detectado un atributo.
- Se haya detectado una cadena de caracteres que puede ser un texto.
- Se haya detectado un error (en el documento, de Entrada/Salida, etc.).

Cuando SAX devuelve que ha detectado un evento, entonces dicho evento puede ser manejado con algún método de control (*callback*) de la clase **`DefaultHandler`**. Esta clase es la que se usa por defecto y contiene métodos que pueden ser sobrecargados por el programador. También se pueden implementar los *callbacks* de la interfaz **`ContentHandler`** para la gestión de dichos eventos.

Cuando SAX detecta un evento de error o un final de documento, entonces se termina el recorrido del fichero XML.

Las principales **interfaces** que necesita un programa Java que utiliza SAX son:

- **`ContentHandler`**. Recibe las notificaciones de los eventos que ocurren en el documento.
- **`DTDHandler`**. Recoge eventos relacionados con la DTD.
- **`ErrorHandler`**. Define métodos de tratamiento de errores.
- **`EntityResolver`**. Sus métodos llaman cada vez que se encuentra una referencia a una entidad.
- **`DefaultHandler`**. Provee una implementación por defecto para todos sus métodos, en la que el programador definirá los métodos a usar en un programa. Por ejemplo:
  1) *startDocument*. Se invoca al comenzar el procesado del fichero XML.
  2) *endDocument*. Se invoca al finalizar el procesado del fichero XML.
  3) *startElement*. Se invoca al comenzar el procesado de una etiqueta XML. Aquí se leen los atributos de la etiqueta.
  4) *endElement*. Se invoca al finalizar el procesado de una etiqueta XML.
  5) *characters*. Se invoca cuando se encuentra una cadena de texto.
- **`XMLReader`**. Realiza la lectura de un documento XML usando varios métodos (***`setContentHandler()`***, ***`setDTDHandler()`***, ***`setEntityResolver()`*** y ***`setErrorHandler()`***). Cada método trata un tipo de evento y está asociado con una interfaz determinada.

El siguiente programa Java lee el fichero XML "alumnos.xml" mediante la generación de eventos y su posterior tratamiento en los métodos definidos para ello.

```
import java.io.*;
import org.xml.sax.Attributes;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.helpers.XMLReaderFactory;

public class PruebaSax1 {
    public static void main(String[] args)
    throws FileNotFoundException, IOException, SAXException {
        XMLReader  procesadorXML = XMLReaderFactory.createXMLReader();
        GestionContenido gestor= new GestionContenido();
        procesadorXML.setContentHandler(gestor);
        InputSource fileXML = new InputSource("alumnos.xml");
        procesadorXML.parse(fileXML);
    }
}//fin PruebaSax1

class GestionContenido extends DefaultHandler {
    public GestionContenido() {
        super();
    }
    public void startDocument() {
        System.out.println("Comienzo del Documento XML");
    }
    public void endDocument() {
        System.out.println("Final del Documento XML");
    }
    public void startElement(String uri, String nombre, String nombreC, Attributes atts) {
        System.out.printf("\t Principio Elemento: %s %n",nombre);
    }
    public void endElement(String uri, String nombre, String nombreC) {
        System.out.printf("\tFin Elemento: %s %n", nombre);
    }
    public void characters(char[] ch, int inicio, int longitud)
    throws SAXException {
        String car = new String(ch, inicio, longitud);
        car = car.replaceAll("[\t\n]","");  //quitar saltos de línea
        System.out.printf ("\t Caracteres: %s %n", car);
    }
}//fin GestionContenido
```

En este programa Java se realizan los siguientes pasos:

1) Se crea un objeto procesador de XML, es decir, un ***XMLReader***. Durante la creación de este objeto se puede producir una excepción (***SAXException***) que es necesario capturar.
2) A continuación, hay que indicar al ***XMLReader*** qué objetos poseen los métodos que tratarán los eventos. Se ha definido la clase ***GestionContenido***, en la que se tratan solo los eventos básicos: inicio y fin de documento, inicio y fin de etiqueta encontrada y cadena de caracteres encontrada. En este caso, se usa el método **setContentHandler**() para tratar los eventos que ocurren en el documento.
3) Luego se define el fichero "alumnos.xml" que se va a leer mediante un objeto ***InputSource***.
4) Por último, se procesa dicho documento XML mediante el método ***parse()*** del objeto ***XMLReader***.

6 DETECCIÓN Y TRATAMIENTO DE EXCEPCIONES

Una **excepción** es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias. Cuando no es capturada por el programa, se captura por el gestor de excepciones por defecto, que retorna un mensaje y detiene el programa.

La ejecución del siguiente programa produce una excepción y visualiza un mensaje por pantalla indicando el error:

```
public class ejemploExcepcion {
    public static void main(String[] args) {
        int nume = 10, denom = 0, cociente;
        cociente = nume / denom;
        System.out.println("Resultado:" + cociente);
    }
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ejemploExcepcion.main(ejemploExcepcion.java:4)
```

Cuando dicho error ocurre dentro de un método Java, el método crea un objeto **`Exception`** y lo maneja fuera, en el sistema de ejecución. El manejo de excepciones en Java está diseñado pensando en situaciones en las que el método que detecta el error no es capaz de manejarlo. En este caso, el método lanzará una excepción.

Las excepciones en Java son objetos de clases derivadas de la clase base **`Exception`**, que a su vez es una clase derivada de la clase base **`Throwable`**.

6.1 Captura de excepciones

Para capturar una excepción se utiliza el bloque **`try-catch`**. Se encierra en un bloque **`try`** el código que puede generar una excepción. Este bloque **`try`** va seguido de uno o más bloques catch. Cada bloque **`catch`** especifica un tipo de excepción que puede atrapar (capturar) y contiene un manejador de excepciones.

Después del último bloque **`catch`** puede aparecer un bloque **`finally`** (opcional) que siempre se ejecuta, haya ocurrido o no la excepción. Este bloque **`finally`** se utiliza para cerrar ficheros o liberar otros recursos del sistema después de que ocurra una excepción.

```
try {
    // código que puede generar excepciones
}
catch (Excepcion1 e1) {
    // manejo de la Excepcion1
}
catch (Excepcion2 e2) {
    // manejo de la Excepcion2
}
// etc...
finally {
    // se ejecuta después de try o catch
}
```

El siguiente programa Java muestra la captura de 3 tipos de excepciones que se pueden producir. Cuando se encuentra un error, se produce un salto al bloque **`catch`** que maneja dicho error. En este caso, al ejecutar ***`arraynum[10]=20;`*** se lanza la excepción ***`ArrayIndex-OutOfBoundsException`***. Las sentencias posteriores a la que ha causado el error dentro del bloque `try` no se ejecutarán.

```
public class ejemploExcepciones1 {
    public static void main(String[] args) {
        String cad1 = "20", cad2 = "0", mensaje = "";
        int nume, denom, cociente;
        int[] arraynum = new int[4];
        try {
            //codigo que puede producir errores
            arraynum[10] = 20;                  // sentencia que produce la excepción
            nume = Integer.parseInt(cad1);      // no se ejecuta
            denom = Integer.parseInt(cad2);     // no se ejecuta
            cociente = nume / denom;            // no se ejecuta
            mensaje = String.valueOf(cociente); // no se ejecuta
        }
        catch (NumberFormatException ex){
            mensaje = "Caracteres no numéricos.";
        }
        catch (ArithmeticException ex){
            mensaje = "Division por cero.";
        }
        catch (ArrayIndexOutOfBoundsException ex){
            mensaje = "Fuera de rango en el array.";
        }
        finally {
            System.out.println("SE EJECUTA SIEMPRE");
        }
        System.out.println(mensaje);  // sí se ejecuta
    } // fin de main
} // fin de la clase
```

Para capturar cualquier excepción se utiliza la clase base **`Exception`**. Si se usa, hay que ponerla al final de la lista de manejadores para evitar que los manejadores que van detrás queden ignorados.

Para obtener más información sobre la excepción producida, se pueden invocar los métodos de la clase base **`Throwable`**. Algunos de estos métodos son:

- **`StringgetMessage().`** Devuelve la cadena de error del objeto.
- **`StringgetLocalizedMessage().`** Crea una descripción local de este objeto.
- **`StringtoString().`** Devuelve una breve descripción del objeto,
- **`voidprintStackTrace()`**, **`printStackTrace(PrintStream)`** o **`printStackTrace(PrintWriter).`** Imprime el objeto y la traza de pila de llamadas lanzada.

```
public class ejemploExcepciones2 {
    public static void main(String[] args) {
        String cad1 = "20", cad2 = "0", mensaje;
        int nume, denom, cociente;
        int[] arraynum = new int[4];
        try {
            arraynum[10] = 20;
            nume = Integer.parseInt(cad1);
            denom = Integer.parseInt(cad2);
            cociente = nume / denom;
            mensaje = String.valueOf(cociente);
        }
        catch (Exception ex){
            System.err.println("toString           => " + ex.toString());
            System.err.println("getMessage         => " + ex.getMessage());
            System.err.println("getLocalizedMessage=> " + ex.getLocalizedMessage());
            ex.printStackTrace();
        }
        finally {
            System.out.println("SE EJECUTA SIEMPRE");
        }
    } // fin de main
} // fin de la clase
```

El anterior programa Java muestra la siguiente salida al ejecutarse:

```
toString           => java.lang.ArrayIndexOutOfBoundsException: 10
getMessage         => 10
getLocalizedMessage=> 10
java.lang.ArrayIndexOutOfBoundsException: 10
    at ejemploExcepciones2.main(ejemploExcepciones2.java:8)
SE EJECUTA SIEMPRE
```

Una sentencia try puede estar dentro de un bloque de otra sentencia try. Si la sentencia try interna no tiene un manejador catch, se buscará el manejador en las sentencias try más externas.

6.2 Especificación de excepciones

Para especificar una o varias excepciones se utiliza la palabra clave **`throws`**, seguida de la lista de todos los tipos de excepciones potenciales. Si un método decide no gestionar una excepción (mediante **`try-catch`**), debe especificar que puede lanzar esa excepción.

El siguiente ejemplo indica que el método ***`main()`*** puede lanzar las excepciones ***`IOException`*** y ***`ClassNotFoundException`***:

```
Public static void main(String[] args)
throwsIOException, ClassNotFoundException{}
```

Aquellos métodos que pueden lanzar excepciones deben saber cuáles son esas excepciones que se pueden producir durante su ejecución e indicarlas en su declaración. Una forma típica de conocerlas es compilando el programa.

La siguiente figura muestra las excepciones que se pueden producir durante la ejecución de los diferentes métodos del paquete **`java.io`**:

Exception	Description
CharConversionException	Base class for character conversion exceptions.
EOFException	Signals that an end of file or end of stream has been reached unexpectedly during input.
FileNotFoundException	Signals that an attempt to open the file denoted by a specified pathname has failed.
InterruptedIOException	Signals that an I/O operation has been interrupted.
InvalidClassException	Thrown when the Serialization runtime detects one of the following problems with a Class.
InvalidObjectException	Indicates that one or more deserialized objects failed validation tests.
IOException	Signals that an I/O exception of some sort has occurred.
NotActiveException	Thrown when serialization or deserialization is not active.
NotSerializableException	Thrown when an instance is required to have a Serializable interface.
ObjectStreamException	Superclass of all exceptions specific to Object Stream classes.
OptionalDataException	Exception indicating the failure of an object read operation due to unread primitive data, or the end of data belonging to a serialized object in the stream.
StreamCorruptedException	Thrown when control information that was read from an object stream violates internal consistency checks.
SyncFailedException	Signals that a sync operation has failed.
UnsupportedEncodingException	The Character Encoding is not supported.
UTFDataFormatException	Signals that a malformed string in modified UTF-8 format has been read in a data input stream or by any class that implements the data input interface.
WriteAbortedException	Signals that one of the ObjectStreamExceptions was thrown during a write operation.