

UD 1

ALMACENAMIENTO DE LA INFORMACIÓN

Clases asociadas a las operaciones de gestión de ficheros y directorios: creación, borrado, copia, movimiento, entre otras

En **Java SE 8** las clases que permiten trabajar con ficheros están en el paquete **java.io**. Pueden consultarse en <https://docs.oracle.com/javase/8/docs/api/>

Clase **File**

Permite obtener información relativa a directorios y ficheros y realizar operaciones sobre ellos como borrar, renombrar, ...

Constructores

El constructor más sencillo tiene esta sintaxis:

- **File**(**String** ruta)

Crea un nuevo objeto **File** (puede ser un directorio o un fichero) en la ruta indicada.

Métodos accesoros

<code>public boolean canRead()</code>	devuelve true si el fichero o directorio tiene permiso de lectura
<code>public boolean canWrite()</code>	devuelve true si el fichero o directorio tiene permiso de escritura
<code>public boolean canExecute()</code>	devuelve true si el fichero tiene permiso de ejecución o si el directorio se puede establecer como el directorio actual
<code>public boolean exists()</code>	devuelve true si el fichero o directorio existe
<code>public boolean isDirectory()</code>	devuelve true si es un directorio

<code>public boolean isFile()</code>	devuelve true si es un fichero
<code>public long length()</code>	devuelve el tamaño en bytes si el objeto es un fichero
<code>public String getName()</code>	devuelve el nombre del fichero
<code>public File getParent()</code>	devuelven el nombre del directorio del fichero o el nombre del directorio padre del directorio
<code>public File getParentFile()</code>	

Consulta de ficheros en un directorio

<code>public String[] list()</code>	devuelve un array con los nombres de directorios y ficheros dentro de un directorio
<code>public File[] listFiles()</code>	devuelve un array con los ficheros y directorios dentro de un directorio
<code>public java.nio.file.Path toPath()</code>	devuelve un objeto que permite acceder a información y funcionalidad adicional

Creación, borrado y renombrado

<code>public boolean createNewFile()</code>	crea un nuevo fichero
<code>public static File createTempFile()</code>	crea un nuevo fichero temporal y devuelve dicho objeto File
<code>public boolean delete()</code>	elimina el fichero o directorio
<code>public boolean renameTo()</code>	renombra el fichero o directorio
<code>public boolean mkdir()</code>	crea un directorio

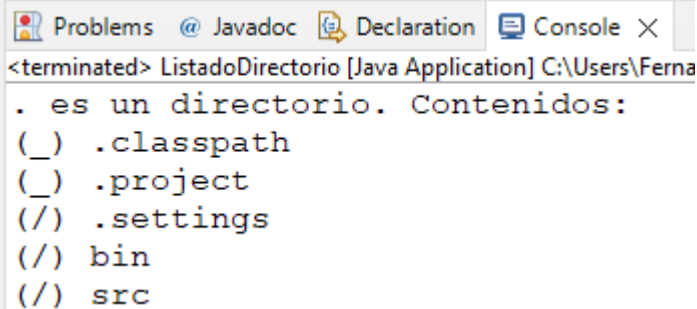
Ejemplo: creamos la clase **ListadoDirectorio** con un método **main()** para probar algunos de estos métodos en el objeto **fich** de clase **File**

Probamos algunos métodos de la clase **File**:

- **Constructor** → línea 12
- **exists()** → línea 14
- **isFile()** → líneas 18 y 25
- **listFiles()** → línea 23
- **isDirectory()** → línea 25
- **getName()** → línea 26

```
1 package listadodirectorio;
2
3 import java.io.File;
4
5 public class ListadoDirectorio {
6
7     public static void main(String[] args) {
8
9         String ruta = ".";
10        if (args.length >= 1) ruta = args[0];
11
12        File fich = new File(ruta);
13
14        if (! fich.exists()) {
15            System.out.println("No existe el fichero o directorio (" + ruta + ").");
16        }
17        else {
18            if (fich.isFile()) {
19                System.out.println(ruta + " es un fichero.");
20            }
21            else {
22                System.out.println(ruta + " es un directorio. Contenidos: ");
23                File[] ficheros = fich.listFiles(); // Ojo, ficheros o directorios
24                for (File f: ficheros) {
25                    String textoDescr = f.isDirectory() ? "/" : f.isFile() ? "_" : "?";
26                    System.out.println("(" + textoDescr + " " + f.getName());
27                }
28            }
29        }
30    }
31 }
32
33 }
```

Probamos desde Eclipse el método **main** sin parámetros y observamos este resultado:

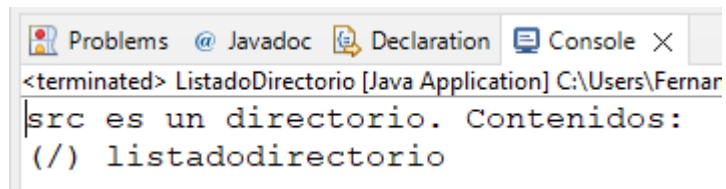
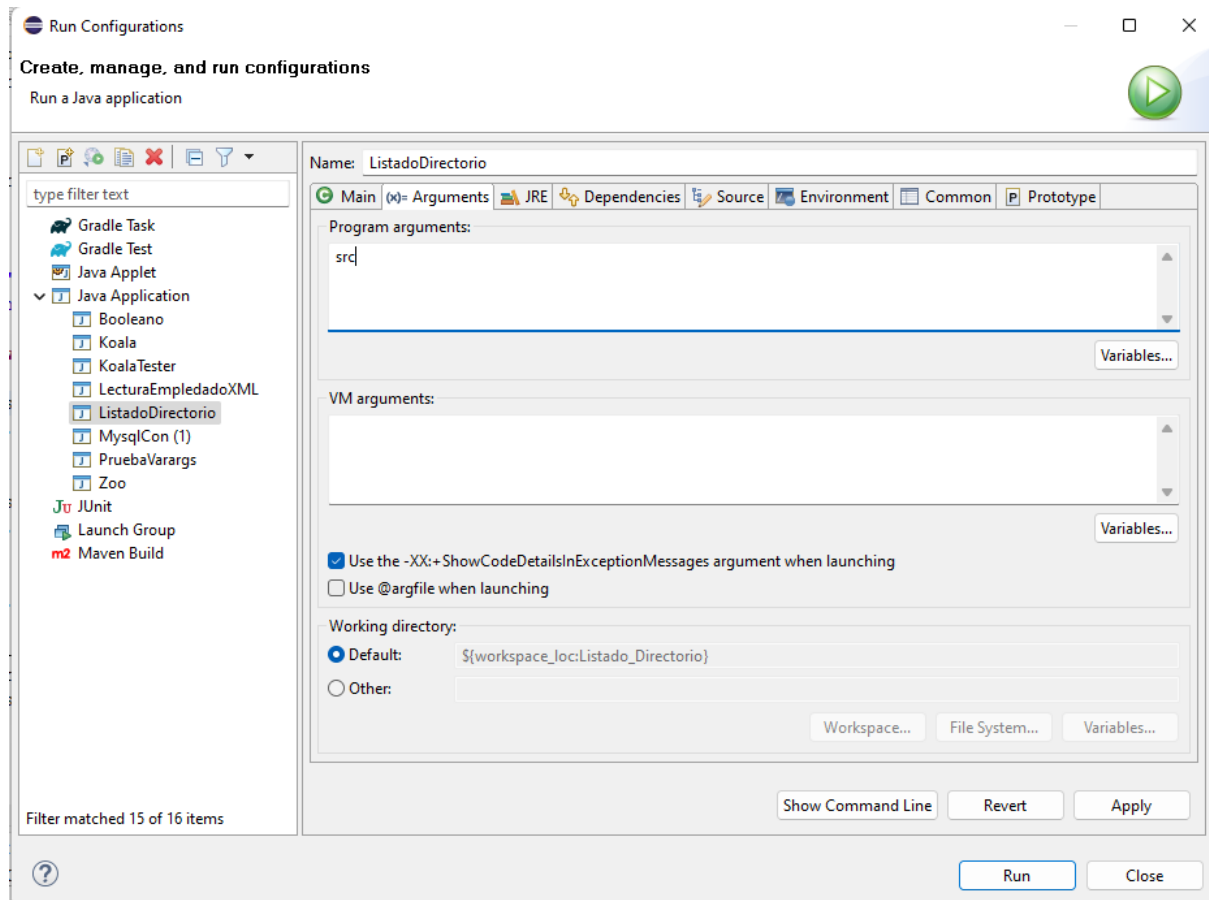


The screenshot shows the Eclipse IDE's Console window. The title bar includes 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console output is as follows:

```
<terminated> ListadoDirectorio [Java Application] C:\Users\Ferna
. es un directorio. Contenidos:
(_) .classpath
(_) .project
(/) .settings
(/) bin
(/) src
```

Coincidente con el contenido de **C:\Users\usuario\eclipse-workspace\proyecto**

Si lo probamos desde pasando como “src” **parámetro** entonces observaremos este resultado:



Coincidente con el contenido de **C:\Users\usuario\eclipse-workspace\proyecto\src**

Excepciones: detección y tratamiento

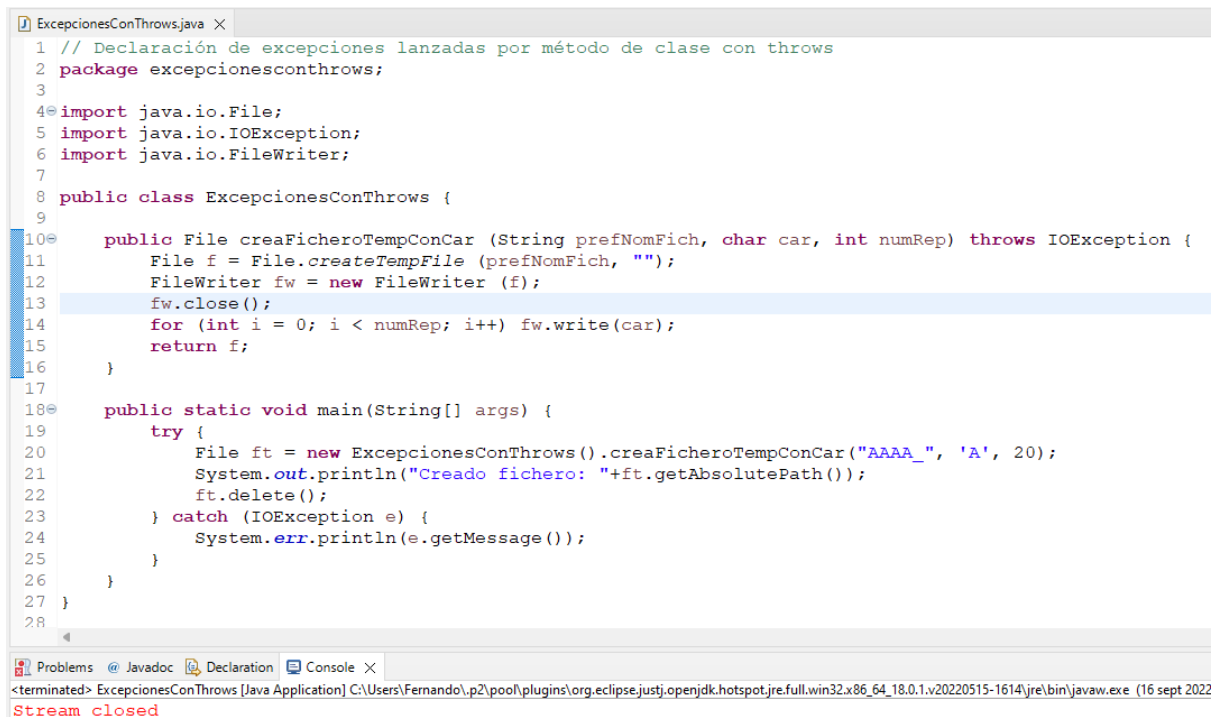
Si el compilador de Java detecta que un método de una clase puede originar un tipo de excepción pero no lo gestiona entonces la compilación terminará con un error.

Opciones:

1. Capturar y gestionar la excepción en un bloque **catch(){}**
2. Lanzar la excepción mediante **throws** seguido de la clase de la excepción

Ejemplo: la siguiente clase tiene un método que crea un fichero temporal con un nombre que empieza por un prefijo dado, y escribe en él un carácter dado, un número dado de veces (las clases y procedimientos utilizados los veremos más adelante)

- Durante este proceso puede saltar la excepción **IOException** en varias ocasiones, pero no se quiere gestionarla en el método **creaFicheroTempConCar()**. Por ello se incluye **throws IOException** en la declaración del método
- La captura de la excepción se gestionará en el **main()**



```
1 // Declaración de excepciones lanzadas por método de clase con throws
2 package excepcionesconthrows;
3
4 import java.io.File;
5 import java.io.IOException;
6 import java.io.FileWriter;
7
8 public class ExcepcionesConThrows {
9
10     public File creaFicheroTempConCar (String prefNomFich, char car, int numRep) throws IOException {
11         File f = File.createTempFile (prefNomFich, "");
12         FileWriter fw = new FileWriter (f);
13         fw.close();
14         for (int i = 0; i < numRep; i++) fw.write(car);
15         return f;
16     }
17
18     public static void main(String[] args) {
19         try {
20             File ft = new ExcepcionesConThrows().creaFicheroTempConCar("AAAA_", 'A', 20);
21             System.out.println("Creado fichero: "+ft.getAbsolutePath());
22             ft.delete();
23         } catch (IOException e) {
24             System.err.println(e.getMessage());
25         }
26     }
27 }
28
```

Problems @ Javadoc Declaration Console X
<terminated> ExcepcionesConThrows [Java Application] C:\Users\Fernando\AppData\Local\Temp\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64.18.0.1.v20220515-1614\jre\bin\javaw.exe (16 sept 2022)
Stream closed

Lo que ocurre es:

1. En la línea 13 se cierra un objeto **fw** de clase **FileWriter**
2. Al intentar ejecutarse en la línea 14 el método **write(car)** se produce una **excepción** de la clase **IOException** porque el objeto fw está cerrado
3. El método **creaFicheroTempConCar()** lanza esa **excepción** de la clase **IOException** al método **main()** que lo invocó
4. El método **main()** captura la excepción de la clase **IOException** e imprime el mensaje **Stream closed**

Inicialización y liberación de recursos en el tratamiento de excepciones: bloques **finally** y **try con recursos**

Es muy frecuente que un bloque de programa de Java esté estructurado de la siguiente forma:

```
Inicialización y asignación de recursos
Cuerpo
Finalización y liberación de recursos
```

La **asignación** y la **liberación** de **recursos** deben ejecutarse siempre, independientemente de la ejecución del cuerpo.

El cuerpo con gestión de excepciones podría ser algo así:

```
Inicialización y asignación de recursos
try {
    Cuerpo
} catch (Excepcion_Tipo_1 e1) {
    Gestión de excepción de tipo 1
} catch (Excepcion_Tipo_2 e2) {
    Gestión de excepción de tipo 1
} catch (Exception e) {
    Gestión del resto de tipos de excepciones
}
Finalización y liberación de recursos
```

Si durante el tratamiento de alguna de esas excepciones se termina la ejecución del flujo normal del código (por ejemplo con **return**, **break** o **continue**) entonces no se alcanzará el bloque de finalización y liberación de recursos.

Para evitar este problema Java implementa dos opciones:

- Codificar un **bloque finally** (es la opción más antigua en Java)
- Implementar un **bloque try con recursos** (la opción más moderna en Java)

Bloque **finally**

Como el bloque **finally** se ejecuta siempre (se ejecuta “sí o sí”) **después** de cualquier conjunto de **bloques catch** permite finalizar y liberar los recursos no liberados en los bloques catch.

Inicialización y asignación de recursos

```
try {  
    Cuerpo  
} catch (Excepcion_Tipo_1 e1) {  
    Gestión de excepción de tipo 1  
} catch (Excepcion_Tipo_2 e2) {  
    Gestión de excepción de tipo 1  
} catch (Exception e) {  
    Gestión del resto de tipos de excepciones  
}  
finally {  
    Finalización y liberación de recursos  
}
```

Ejemplo: La siguiente clase de ejemplo simula un programa que

1. abre para lectura dos ficheros **f1.dat** y **f2.dat** y crea dos ficheros temporales **f1.info.tmp** y **f2.info.tmp** (que borrará al final del programa)
2. introducimos un par de sentencias **return** que pueden finalizar la ejecución del código dejando recursos sin cerrar
3. creamos la variable aleatoria **falloTras** que solamente podrá tomar valores 1, 2 y 3
4. si **falloTras** no vale **3** entonces se ejecutará un **return** pudiendo quedar recursos sin cerrar
5. pero **pase lo que pase** se ejecutará el **bloque finally** y ahí se cerrarán los recursos que hayan quedado abiertos

```
// Liberación de recursos en bloque finally
package excepcionesconfinally2;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.File;
import java.io.IOException;
import java.util.Random;

public class ExcepcionesConFinally2 {
    public static void main(String[] args) {
        File fml=null;
        File fm2=null;
        FileInputStream ifs1=null;
        FileInputStream ifs2=null;

        try {
            Random aleat = new Random();
            int falloTras = aleat.nextInt(3) + 1;

            if (falloTras <= 1) {
                System.out.println("SE SALE AL LLEGAR A "+falloTras);
                return;
            }

            ifs1 = new FileInputStream("f1.dat");
            System.out.println("Abierto f1.dat");
            fml=new File("f1.info.tmp"); fml.createNewFile();
            System.out.println("Creado "+fml.getAbsolutePath());

            if (falloTras <= 2) {
                System.out.println("SE SALE AL LLEGAR A "+falloTras);
                return;
            }
        }
    }
}
```

m


```

        ifs2 = new FileInputStream("f2.dat");
        System.out.println("Abierto f2.dat");
        fm2=new File("f2.info.tmp"); fm2.createNewFile();
        System.out.println("Creado "+fm2.getAbsolutePath());

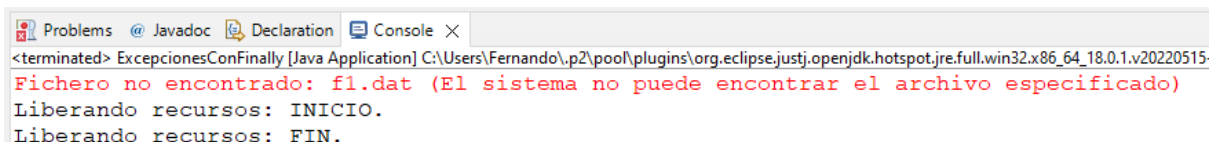
        System.out.println("Ejecutado hasta el final");

    } catch (FileNotFoundException e) {
        System.err.println("Fichero no encontrado: "+e.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        System.out.println("Liberando recursos: INICIO.");
        if(ifs1!=null) {
            try {
                ifs1.close();
                System.out.println("Cerrado f1.dat");
            }
            catch (IOException e) { System.err.println("Error al cerrar
                fichero: "+e.getMessage()); }
        }
        if(ifs2!=null) {
            try {
                ifs2.close();
                System.out.println("Cerrado f2.dat");
            }
            catch (IOException e) { System.err.println("Error al cerrar
                fichero: "+e.getMessage()); }
        }
        if(fml!=null) {
            fml.delete();
            System.out.println("Borrado "+fml.getName());
        }
        if(fm2!=null) {
            fm2.delete();
            System.out.println("Borrado "+fm2.getName());
        }
        System.out.println("Liberando recursos: FIN.");
    }
}
}
}

```

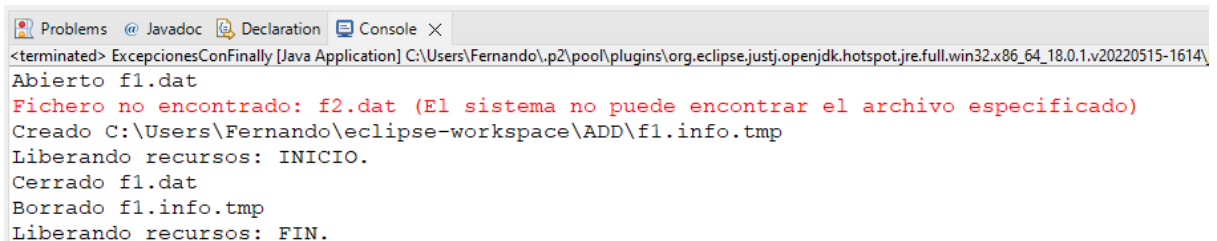
Se debe probar la ejecución en este orden:

1º. Sin crear previamente **f1.dat** ni **f2.dat**



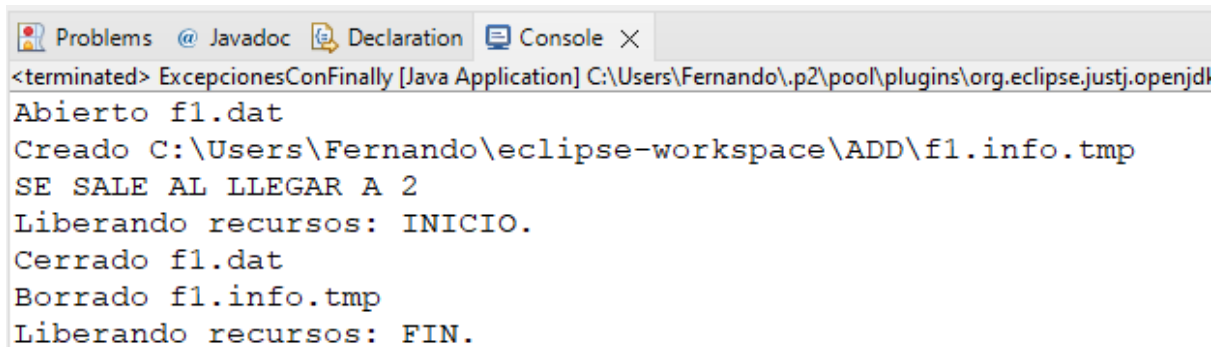
Problems @ Javadoc Declaration Console X
 <terminated> ExcepcionesConFinally [Java Application] C:\Users\Fernando\.p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_18.0.1.v20220515-1614\
 Fichero no encontrado: f1.dat (El sistema no puede encontrar el archivo especificado)
 Liberando recursos: INICIO.
 Liberando recursos: FIN.

2º. Creando solamente **f1.dat** pero no **f2.dat**



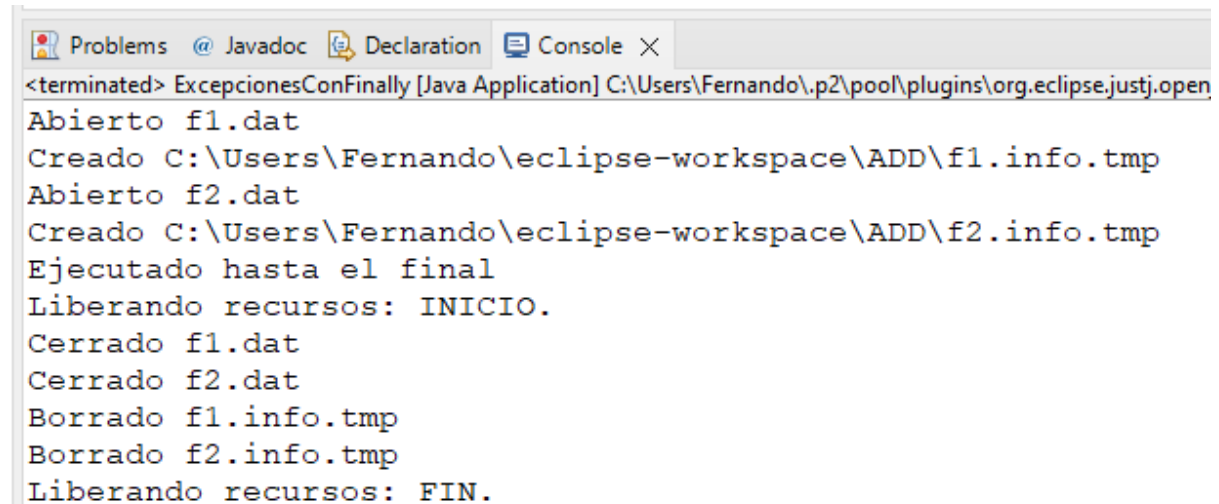
Problems @ Javadoc Declaration Console X
 <terminated> ExcepcionesConFinally [Java Application] C:\Users\Fernando\.p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_18.0.1.v20220515-1614\
 Abierto f1.dat
 Fichero no encontrado: f2.dat (El sistema no puede encontrar el archivo especificado)
 Creado C:\Users\Fernando\eclipse-workspace\ADD\f1.info.tmp
 Liberando recursos: INICIO.
 Cerrado f1.dat
 Borrado f1.info.tmp
 Liberando recursos: FIN.

3.º Creando también **f2.dat**

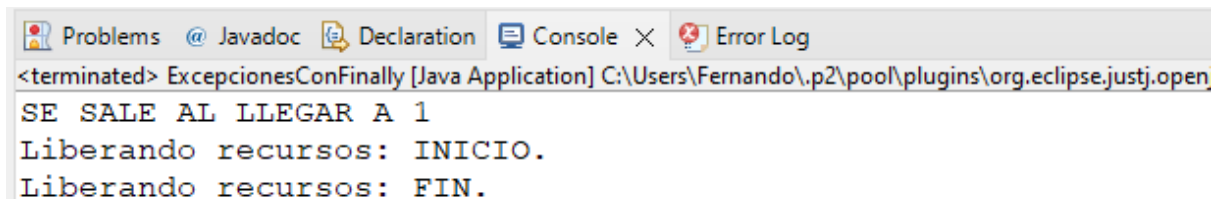


```
<terminated> ExcepcionesConFinally [Java Application] C:\Users\Fernando\.p2\pool\plugins\org.eclipse.justj.openjdk
Abierto f1.dat
Creado C:\Users\Fernando\eclipse-workspace\ADD\f1.info.tmp
SE SALE AL LLEGAR A 2
Liberando recursos: INICIO.
Cerrado f1.dat
Borrado f1.info.tmp
Liberando recursos: FIN.
```

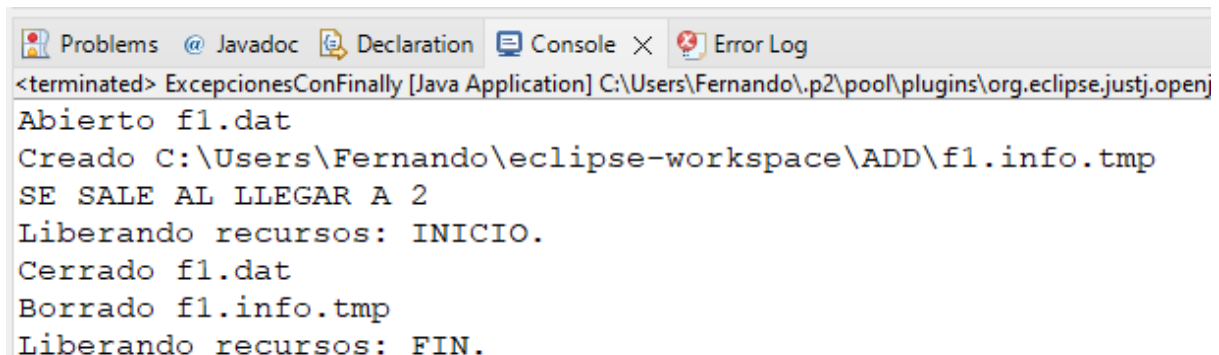
4º. Probarlo varias veces observando que valga lo que valga **falloTras** se ejecuta **siempre** el **bloque finally** liberando recursos



```
<terminated> ExcepcionesConFinally [Java Application] C:\Users\Fernando\.p2\pool\plugins\org.eclipse.justj.open
Abierto f1.dat
Creado C:\Users\Fernando\eclipse-workspace\ADD\f1.info.tmp
Abierto f2.dat
Creado C:\Users\Fernando\eclipse-workspace\ADD\f2.info.tmp
Ejecutado hasta el final
Liberando recursos: INICIO.
Cerrado f1.dat
Cerrado f2.dat
Borrado f1.info.tmp
Borrado f2.info.tmp
Liberando recursos: FIN.
```



```
<terminated> ExcepcionesConFinally [Java Application] C:\Users\Fernando\.p2\pool\plugins\org.eclipse.justj.open
SE SALE AL LLEGAR A 1
Liberando recursos: INICIO.
Liberando recursos: FIN.
```



```
<terminated> ExcepcionesConFinally [Java Application] C:\Users\Fernando\.p2\pool\plugins\org.eclipse.justj.open
Abierto f1.dat
Creado C:\Users\Fernando\eclipse-workspace\ADD\f1.info.tmp
SE SALE AL LLEGAR A 2
Liberando recursos: INICIO.
Cerrado f1.dat
Borrado f1.info.tmp
Liberando recursos: FIN.
```

Bloque try con recursos

A diferencia de un bloque try habitual se añaden **paréntesis** después de try y antes del bloque.

- Bloque **try normal**: `try {}`
- Bloque **try con recursos**: `try () {}`

Dentro de esos paréntesis se pueden incluir (separados por **punto y coma**) **recursos** de clases que implementen las interfaces **Closeable** o **AutoCloseable**.

- Si **hay** un **bloque finally** entonces **no hace falta llamar al método close()** de estos recursos porque se hará la llamada automáticamente dentro del bloque
- Si **no hay** un **bloque finally** tampoco hará falta **llamar al método close()** porque el compilador crea un bloque finally vacío donde se harán dichas llamadas automáticamente

Inicialización y asignación de recursos

```
try (T1 r1=new T1(); T2 r2=new T2()) {  
    // T1 y T2 implementan Closeable o AutoCloseable  
    Cuerpo  
} catch (Excepcion_Tipo_1 e1) {  
    Gestión de excepción de tipo 1  
} catch (Excepcion_Tipo_2 e2) {  
    Gestión de excepción de tipo 1  
} catch (Exception e) {  
    Gestión del resto de tipos de excepciones  
}  
finally {  
    Finalización y liberación de recursos  
    // No es necesario r1.close()  
    // No es necesario r2.close()  
}
```

Ejemplo: En el ejemplo anterior hemos cerrado los objetos de la clase **FileInputStream** en el bloque **finally** pero

- como la clase **FileInputStream** implementa las interfaces **Closeable** y **AutoCloseable** (bastaría con que implementara una de las dos) entonces se las puede incluir en un **bloque try con recursos**

```
// Bloque try con recursos
package ExcepcionesConTryConRecursos;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.File;
import java.util.Random;
public class ExcepcionesConTryConRecursos {
    public static void main(String[] args) {
        File fml=null;
        File fm2=null;
        try(
            FileInputStream ifs1 = new FileInputStream("f1.dat");
            FileInputStream ifs2 = new FileInputStream("f2.dat")) {
            System.out.println("Abierto f1.dat: "+ifs1.getFD().toString());
            System.out.println("Abierto f2.dat: "+ifs2.getFD().toString());
            Random aleat = new Random();
            int falloTras = aleat.nextInt(3) + 1;
            if (falloTras <= 1) {
                System.out.println("SE SALE AL LLEGAR A "+falloTras);
                return;
            }
            fml=new File("f1.info.tmp"); fml.createNewFile();
            System.out.println("Creado "+fml.getAbsolutePath());
            if (falloTras <= 2) {
                System.out.println("SE SALE AL LLEGAR A "+falloTras);
                return;
            }
            fm2=new File("f2.info.tmp"); fm2.createNewFile();
            System.out.println("Creado "+fm2.getAbsolutePath());
            System.out.println("Ejecutado hasta el final");
        } catch (FileNotFoundException e) {
            System.err.println("Fichero no encontrado: "+e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println("Liberando recursos: INICIO.");
            if(fml!=null) {
                fml.delete();
                System.out.println("Borrado "+fml.getName());
            }
        }

        if(fm2!=null) {
            fm2.delete();
            System.out.println("Borrado "+fm2.getName());
        }
        System.out.println("Liberando recursos: FIN.");
    }
}
```

Nota: en el bloque **finally** seguimos eliminando los objetos de clase **File** (que no implementan las interfaces **Closeable** ni **AutoCloseable**)

1º. Sin crear previamente **f1.dat** ni **f2.dat**

2°. Creando solamente **f1.dat** pero no **f2.dat**

3.º Creando también f2.dat

4º. Probarlo varias veces observando que valga lo que valga **falloTras** se ejecuta **siempre** el **bloque finally**

Problems @ Javadoc Declaration Console X Error Log

<terminated> ExcepcionesTryConRecursos [Java Application] C:\Users\Fernando.p2\pool\plugins\org.eclipse.justj.openj

```
Abierto f1.dat: java.io.FileDescriptor@35fb3008
Abierto f2.dat: java.io.FileDescriptor@3551a94
Creado C:\Users\Fernando\eclipse-workspace\ADD\f1.info.tmp
Creado C:\Users\Fernando\eclipse-workspace\ADD\f2.info.tmp
Ejecutado hasta el final
Liberando recursos: INICIO.
Borrado f1.info.tmp
Borrado f2.info.tmp
Liberando recursos: FIN.
```

Flujos. Flujos basados en bytes y flujos basados en caracteres

El sistema de entrada/salida en Java representa una gran cantidad de clases que se implementan en el sistema **java.io**.

Java llama **flujo (stream)** a cualquier transmisión de información desde una fuente a un destino. La fuente y el destino pueden ser un fichero en disco duro, un array en memoria, información en red, datos en otro programa, ...

Cualquier programa que tenga que obtener información de cualquier fuente necesita abrir un stream, igualmente si necesita enviar información abrirá un stream y se escribirá la información en serie.

Flujos basados en **bytes**

Realizan operaciones de entrada y salida de **bytes** (8 bits) para la lectura/escritura de **datos binarios**.

Flujos de **entrada** basados en **bytes**

Todas las clases derivan de **InputStream**. Las clases más importantes son:

ByteArrayInputStream	permite usar un espacio de almacenamiento intermedio de memoria
StringBufferInputStream	convierte un String en un InputStream
FileInputStream	para leer información de un fichero
PipedInputStream	implementa el concepto de pipe (tubería)
FilterInputStream	proporciona funcionalidad a otras clases derivadas de InputStream
SequenceInputStream	convierte varios InputStream en un único InputStream

Flujos de **salida** basados en **bytes**

Todas las clases derivan de **OutputStream**. Las clases más importantes son:

ByteArrayOutputStream	se envía un flujo de datos a un espacio de almacenamiento intermedio de memoria
FileOutputStream	para escribir información en un fichero
PipedOutputStream	implementa el concepto de pipe (tubería): la información enviada aquí será usada como entrada en un PipedInputStream
FilterOutputStream	proporciona funcionalidad a otras clases derivadas de OutputStream

Flujos basados en **caracteres**

Realizan operaciones de entrada y salida de **caracteres Unicode** (16 bits) para la lectura/escritura de **datos alfanuméricos**.

Flujos de **entrada** basados en **caracteres**

Todas las clases derivan de **Reader**. Las clases más importantes son:

FileReader	para lectura de caracteres de ficheros
CharArrayReader	para lectura de caracteres de arrays de caracteres
BufferedReader	para lectura de caracteres de un búfer intermedio

Flujos de **salida** basados en **caracteres**

Todas las clases derivan de **Writer**. Las clases más importantes son:

FileWriter	para escritura de caracteres en ficheros
CharArrayWriter	para escritura de caracteres en arrays de caracteres
BufferedWriter	para escritura de caracteres en un búfer intermedio

Resumen de las **clases básicas** para **entrada y salida a flujos**

	Fuente de datos	Lectura	Escritura
Flujo Binario	Ficheros	FileInputStream	FileOutputStream
	Memoria (byte[])	ByteArrayInputStream	ByteArrayOutputStream
	Tuberías	PipedInputStream	PipedOutputStream
Flujo de Texto	Ficheros	FileReader	FileWriter
	Memoria (char[])	CharArrayReader	CharArrayWriter
	Memoria (String)	StringReader	StringWriter
	Tuberías	PipedReader	PipedWriter

- El principio del nombre indica el tipo de fuente:
 - **File** → fichero
 - **ByteArray** → memoria
 - **Piped** → tubería
- El medio del nombre indica si es de entrada o de salida
 - **Input / Reader** → Entrada
 - **Output / Writer** → Salida
- El final del nombre indica si es de bytes o de texto
 - **Stream** → binario
 - En otro caso → texto

Clases de **enlace entre flujos binarios y de texto**

Las clases **InputStreamReader** y **OutputStreamWriter** sirven de enlace entre los flujos binarios y los flujos de texto y permiten **recodificar texto** (cambiar la codificación del texto de entrada o del texto de salida)

- **InputStreamReader** → convierte flujos binarios de entrada en flujos de texto de entrada
- **OutputStreamWriter** → convierte flujos binarios de salida en flujos de texto de salida

Clases para **buffering**

El **buffering** es una técnica que utiliza una zona de memoria intermedia llamada **buffer** que permite:

- acelerar las operaciones de lectura y escritura
- leer y escribir **líneas de texto** (en vez de bytes)

	Lectura	Escritura
Flujo Binario	BufferedInputStream	BufferedOutputStream
Flujo de Texto	BufferedReader	BufferedWriter

Podemos convertir **flujos sin buffering** a **flujos con buffering** pasando como parámetro al constructor de los flujos con buffering el constructor de los flujos sin buffering:

Flujo sin buffering	Flujo con buffering
<code>new FileInputStream("f.bin")</code>	<code>new BufferedInputStream(new FileInputStream("f.bin"))</code>
<code>new FileOutputStream("f.bin")</code>	<code>new BufferedOutputStream(new FileOutputStream("f.bin"))</code>
<code>new FileReader("f.txt")</code>	<code>new BufferedReader(new FileReader("f.txt"))</code>
<code>new FileWriter("f.txt")</code>	<code>new BufferedWriter(new FileWriter("f.txt"))</code>

Las clases para flujos de texto con buffering tienen métodos que permiten leer y **escribir** líneas:

Clase	Método	Funcionalidad
BufferedReader	<code>String readLine()</code>	Lee hasta el final de una línea
BufferedWriter	<code>void newLine()</code>	Escribe un separador de líneas (depende del sistema operativo)

Métodos (operaciones) de las clases de flujos de entrada

- Los métodos **read()** de las funciones que heredan de **InputStream** leen **bytes**
 - Si no tienen parámetros leen 1 byte
 - Si reciben un array sin offset ni longitud → leen bytes hasta llenarlo
 - Si recibe un offset y una longitud → leen los bytes necesarios para escribir en el array desde la posición offset hasta longitud
- Los métodos **read()** de las funciones que heredan de **Reader** leen **caracteres**
 - Si no tienen parámetros leen 1 carácter
 - Si reciben un array leen caracteres hasta llenarlo
 - Si recibe un offset y una longitud → leen los caracteres necesarios para escribir en el array desde la posición offset hasta longitud
 - Si recibe un objeto de la clase **CharBuffer** lee caracteres hasta llenarlo
- Los métodos **skip()** saltan el número indicado de bytes o de caracteres
- El método **readLine()** lee líneas de texto de un **BufferedReader** construido sobre un **FileReader**

InputStream	Reader
int read()	int read()
int read(byte[] buffer)	int read(char[] buffer)
int read(byte[] buffer, int offset, int longitud)	int read(char[] buffer, int offset, int longitud)
	int read(CharBuffer buffer)
long skip(long n)	long skip(long n)
	BufferedReader
	String readLine()

Ejemplo 1: clase que lee línea a línea un fichero usando el método `readLine()` de la clase `BufferedReader()`

```
1 // Clase para probar el método readLine() de BufferedReader
2 import java.io.FileReader;
3 import java.io.BufferedReader;
4 import java.io.FileNotFoundException;
5 import java.io.IOException;
6
7 public class EscribeConNumeroDeLineas {
8
9     public static void main(String[] args) {
10         if (args.length < 1) {
11             System.out.println("Indicar por favor nombre de fichero.");
12             return;
13         }
14         String nomFich = args[0];
15
16         try (BufferedReader fbr = new BufferedReader(new FileReader(nomFich))) {
17             int i = 0;
18             String linea = fbr.readLine();
19             while (linea != null) {
20                 System.out.format("[%5d] %s", i++, linea);
21                 System.out.println();
22                 linea = fbr.readLine();
23             }
24         } catch (FileNotFoundException e) {
25             System.out.println("No existe fichero " + nomFich);
26         } catch (IOException e) {
27             System.out.println("Error de E/S: " + e.getMessage());
28         } catch (Exception e) {
29             e.printStackTrace();
30         }
31     }
32 }
```

Podemos probar la ejecución pasando como parámetro de entrada el fichero de texto **Empleados.xml** (por ejemplo):

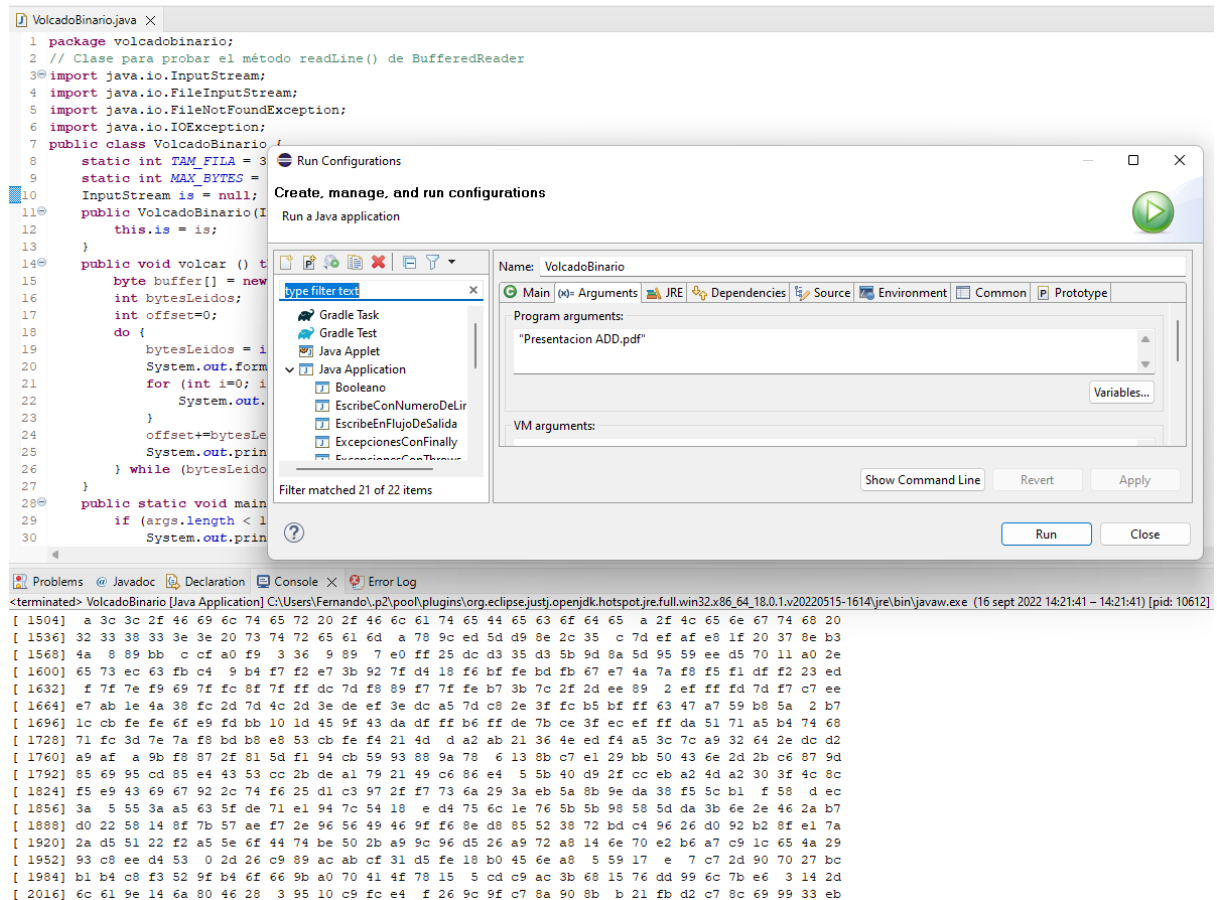


```
<terminated> EscribeConNumeroDeLineas [Java Application] C:\Users\Fernando\p2\pool\plugins\org.eclipse.justj.openjdk.  
[ 0] <?xml version="1.0"?>  
[ 1] <Empleados>  
[ 2]     <empleado>  
[ 3]         <id>1</id>  
[ 4]         <apellido>FERNANDEZ</apellido>  
[ 5]         <dep>10</dep>  
[ 6]         <salario>1000.45</salario>  
[ 7]     </empleado>  
[ 8]     <empleado>  
[ 9]         <id>2</id>  
[10]         <apellido>GIL</apellido>  
[11]         <salario>2400.6</salario>  
[12]         <dep>20</dep>  
[13]     </empleado>  
[14]     <empleado>  
[15]         <id>3</id>  
[16]         <apellido>LOPEZ</apellido>  
[17]         <salario>3000.0</salario>  
[18]         <dep>10</dep>  
[19]     </empleado>  
[20] </Empleados>  
[21]
```

Ejemplo 2: clase que lee un fichero binario en bloques de 32 bytes y muestra en hexadecimal por pantalla hasta un máximo de 2 kilobytes

```
1 package volcadobinario;
2 // Clase para probar el método readLine() de BufferedReader
3 import java.io.InputStream;
4 import java.io.FileInputStream;
5 import java.io.FileNotFoundException;
6 import java.io.IOException;
7 public class VolcadoBinario {
8     static int TAM_FILA = 32;
9     static int MAX_BYTES = 2048;
10    InputStream is = null;
11    public VolcadoBinario(InputStream is) {
12        this.is = is;
13    }
14    public void volcar () throws IOException {
15        byte buffer[] = new byte[TAM_FILA];
16        int bytesLeidos;
17        int offset=0;
18        do {
19            bytesLeidos = is.read(buffer);
20            System.out.format("[%5d]", offset);
21            for (int i=0; i<bytesLeidos; i++) {
22                System.out.format(" %2x", buffer[i]);
23            }
24            offset+=bytesLeidos;
25            System.out.println();
26        } while (bytesLeidos==TAM_FILA && offset<MAX_BYTES);
27    }
28    public static void main(String[] args) {
29        if (args.length < 1) {
30            System.out.println("No se ha indicado ningún fichero");
31            return;
32        }
33        String nomFich = args[0];
34        //El objeto fis se cerrará aunque se produzca una excepción por estar dentro de un try-with-resources
35        try (FileInputStream fis = new FileInputStream(nomFich)) {
36            VolcadoBinario vb = new VolcadoBinario(fis);
37            vb.volcar();
38        } catch (FileNotFoundException e) {
39            System.out.println("ERROR: No existe fichero " + nomFich);
40        } catch (IOException e) {
41            System.out.println("Error de E/S: " + e.getMessage());
42        } catch (Exception e) {
43            e.printStackTrace();
44        }
45    }
46 }
```

Podemos probar la ejecución pasando como parámetro de entrada el fichero binario **Empleados.xml** (por ejemplo) y obtendremos una salida similar a esta:



The screenshot shows an IDE with a Java file named `VolcadoBinario.java` and its Run Configuration dialog. The Java file contains the following code:

```
1 package volcadorbinario;
2 // Clase para probar el método readLine() de BufferedReader
3 import java.io.InputStream;
4 import java.io.FileInputStream;
5 import java.io.FileNotFoundException;
6 import java.io.IOException;
7 public class VolcadoBinario {
8     static int TAM_FILA = 3;
9     static int MAX_BYTES = 1024;
10    InputStream is = null;
11    public VolcadoBinario(InputStream is) {
12        this.is = is;
13    }
14    public void volcar () throws IOException {
15        byte buffer[] = new byte[1024];
16        int bytesLeidos;
17        int offset=0;
18        do {
19            bytesLeidos = is.read(buffer, offset, MAX_BYTES);
20            System.out.write(buffer, offset, bytesLeidos);
21            for (int i=0; i<bytesLeidos; i++) {
22                System.out.print(buffer[i] + " ");
23            }
24            offset+=bytesLeidos;
25            System.out.println();
26        } while (bytesLeidos>0);
27    }
28    public static void main (String[] args) {
29        if (args.length < 1) {
30            System.out.println("Uso: java VolcadoBinario <archivo>");
31        } else {
32            try {
33                VolcadoBinario vb = new VolcadoBinario(new FileInputStream(args[0]));
34                vb.volcar();
35            } catch (FileNotFoundException e) {
36                System.out.println("Archivo no encontrado: " + args[0]);
37            } catch (IOException e) {
38                System.out.println("Error al leer el archivo: " + args[0]);
39            }
40        }
41    }
42 }
```

The Run Configuration dialog is titled "Create, manage, and run configurations" and "Run a Java application". It shows the following settings:

- Name: VolcadoBinario
- Program arguments: "Presentacion ADD.pdf"
- VM arguments: (empty)
- Buttons: Show Command Line, Revert, Apply, Run, Close

The console output shows the following hex dump:

```
<terminated> VolcadoBinario [Java Application] C:\Users\Fernando\p2\poo\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_18.0.1.v20220515-1614\jre\bin\javaw.exe (16 sept 2022 14:21:41 - 14:21:41) [pid: 10612]
[ 1504] a 3c 3c 2f 46 69 6c 74 65 72 20 2f 46 6c 61 74 65 44 65 63 6f 64 65 a 2f 4c 65 6e 67 74 68 20
[ 1536] 32 33 38 33 3e 3e 20 73 74 72 65 61 6d a 78 9c ed 5d d9 8e 2c 35 c 7d ef af e8 1f 20 37 8e b3
[ 1568] 4a 8 89 bb c cf a0 f9 3 36 9 89 7 e0 ff 25 dc d3 35 d3 5b 9d 8a 5d 95 59 ee d5 70 11 a0 2e
[ 1600] 65 73 ec 63 fb c4 9 b4 f7 f2 e7 3b 92 7f d4 18 f6 bf fe bd fb 67 e7 4a 7a f8 f5 f1 df f2 23 ed
[ 1632] f 7f 7e f9 69 7f fc 8f 7f ff dc 7d f8 89 f7 7f fe b7 3b 7c 2f 2d ee 89 2 ef ff fd 7d f7 c7 ee
[ 1664] e7 ab 1e 4a 38 fc 2d 7d 4c 2d 3e de ef 3e dc a5 7d c8 2e 3f fc b5 bf ff 63 47 a7 59 b8 5a 2 b7
[ 1696] 1c cb fe fe 6f e9 fd bb 10 1d 45 9f 43 da df ff b6 ff de 7b ce 3f ec ef ff da 51 71 a5 b4 74 68
[ 1728] 71 fc 3d 7e 7a f8 bd b8 e8 53 cb fe f4 21 d4 d a2 ab 21 36 4e ed f4 a5 3c 7c a9 32 64 2e dc d2
[ 1760] a9 af a 9b f8 87 2f 81 5d f1 94 cb 59 93 88 9a 78 6 13 8b c7 e1 29 bb 50 43 6e 2d 2b c6 87 9d
[ 1792] 85 69 95 cd 85 e4 43 53 cc 2b de a1 79 21 49 c6 86 e4 5 5b 40 d9 2f cc eb a2 4d a2 30 3f 4c 8c
[ 1824] f5 e9 43 69 67 92 2c 74 f6 25 d1 c3 97 2f f7 73 6a 29 3a eb 5a 8b 9e da 38 f5 5c b1 f 58 d ec
[ 1856] 3a 5 55 3a a5 63 5f de 71 e1 94 7c 54 18 e d4 75 6c 1e 76 5b 5b 98 58 5d da 3b 6e 2e 46 2a b7
[ 1888] d0 22 58 14 8f 7b 57 ae f7 2e 96 56 49 46 9f fe 8e d8 85 52 38 72 bd c4 96 26 d0 92 b2 8f e1 7a
[ 1920] 2a d5 51 22 f2 a5 5e 6f 44 74 be 50 2b a9 9c 96 d5 26 a9 72 a8 14 6e 70 e2 b6 a7 c9 1c 65 4a 29
[ 1952] 93 c8 ee d4 53 0 2d 26 c9 89 ac ab cf 31 d5 fe 18 b0 45 6e a8 5 59 17 e 7 c7 2d 90 70 27 bc
[ 1984] b1 b4 c8 f3 52 9f b4 6f 66 9b a0 70 41 4f 78 15 5 cd c9 ac 3b 68 15 76 dd 99 6c 7b e6 3 14 2d
[ 2016] 6c 61 9e 14 6a 80 46 28 3 95 10 c9 fc e4 f 26 9c 9f c7 8a 90 8b b 21 fb d2 c7 8c e9 99 33 eb
```

Métodos (operaciones) de las clases de flujos de salida

Una característica de las clases **FileOutputStream** y **Writer** (y clases derivadas) es que tienen constructores con un parámetro que si es true permite añadir contenido al final del fichero definido como flujo de salida

FileOutputStream	Writer
FileOutputStream (File file, boolean append)	Writer (File file, boolean append)
FileOutputStream (String nombreFichero, boolean append)	Writer (String nombreFichero, boolean append)

Los métodos de escritura de los flujos de salida son:

- Los métodos **write()** de las funciones que heredan de **OutputStream** escriben **bytes**
 - Si el parámetro es un byte → escribe 1 byte al final del flujo de salida
 - Si reciben un array sin offset ni longitud → escriben todos los bytes del array al final del flujo de salida
 - Si recibe un offset y una longitud → escriben los bytes necesarios para escribir al final del flujo de salida el contenido del array desde la posición offset hasta la posición longitud del array (no escribe el array completo)
- Los métodos **write()** de las funciones que heredan de **Writer** escriben **caracteres**
 - Si el parámetro es un carácter → escribe 1 carácter al final del flujo de salida
 - Si reciben un **String** sin offset ni longitud → escribe todo el contenido del **String** al final del flujo de salida
 - Si recibe un offset y una longitud → escribe al final del flujo de salida el contenido del **String** desde la posición offset hasta la posición longitud del **String** (no escribe el **String** completo)
- Los métodos **append()** añaden **caracteres** al final de un fichero
 - Si el parámetro es un carácter → añade 1 carácter al final del fichero
 - Si el parámetro es un objeto de clase **CharSequence** → añade 1 objeto de clase **CharSequence** al final del fichero
 - Si recibe un offset y una longitud → escriben al final del fichero los caracteres del objeto **CharSequence** desde la posición offset del objeto hasta la posición longitud (no escribe el objeto completo)
- El método **newLine()** escribe saltos de línea en un **BufferedWriter** construido sobre un **FileWriter**

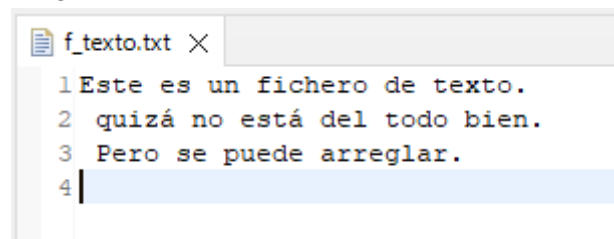
OuputStream	Writer
<code>void write(int b)</code>	<code>void write(int c)</code>
<code>void write(byte[] buffer)</code>	<code>void write(char[] buffer)</code>
<code>void write(byte[] buffer, int offset, int longitud)</code>	<code>void write(char[] buffer, int offset, int longitud)</code>
	<code>void write(String buffer)</code>
	<code>void write(String buffer, int offset, int longitud)</code>
	<code>Writer append(char c)</code>
	<code>Writer append(CharSequence csq)</code>
	<code>Writer append(CharSequence csq, int offset, int longitud)</code>
	BufferedWriter
	<code>void newLine()</code>

Ejemplo 1: clase **EscribeEnFlujoDeSalida**

1. Escribe texto en un fichero con el método **write()**
2. Añade saltos de línea con el método **newLine()**
3. Cierra el fichero y lo vuelve a abrir en modo **append true** (para poder añadir líneas de texto)
 - a. Como **BufferedWriter** deriva de **Writer** mantiene ese constructor con el parámetro **append**
 - b. Si no abro el fichero con ese parámetro entonces se sobrescribirá el contenido del fichero
4. Añade más texto con **write()** y saltos de línea con **newLine()** y cierra el fichero definitivamente

```
1 // Clase para añadir contenidos al final de un fichero de texto
2 package escribenflujodesalida;
3
4 import java.io.File;
5 import java.io.FileWriter;
6 import java.io.BufferedWriter;
7 import java.io.IOException;
8
9 public class EscribeEnFlujoDeSalida {
10
11     public static void main(String[] args) {
12
13         String nomFichero = "f_texto.txt";
14         File f = new File(nomFichero);
15         if (f.exists()) {
16             System.out.println("Fichero "+nomFichero+" ya existe. No se hace nada");
17             return;
18         }
19
20         try {
21             BufferedWriter bfw = new BufferedWriter(new FileWriter(f));
22             bfw.write("Este es un fichero de texto. ");
23             bfw.newLine();
24             bfw.write(" quizá no está del todo bien.");
25             bfw.newLine();
26             bfw.close();
27             bfw = new BufferedWriter(new FileWriter(f, true));
28             bfw.write(" Pero se puede arreglar.");
29             bfw.newLine();
30             bfw.close();
31         } catch (IOException e) {
32             System.out.println(e.getMessage());
33         } catch (Exception e) {
34             e.printStackTrace();
35         }
36     }
37 }
```

Si visualizamos el fichero generado **f_texto.txt** vemos el resultado de la ejecución



```
f_texto.txt X
1 Este es un fichero de texto.
2 quizá no está del todo bien.
3 Pero se puede arreglar.
4 |
```


Ejemplo 2: clase `ArreglaFicheroTexto`

1. Elimina espacios en blanco al inicio de un fichero de texto
2. Sustituye secuencias de blancos por un solo blanco
3. Hace que todas las línea comiencen por mayúscula

```

1 // Clase para añadir contenidos al final de un fichero de texto
2 package arreglaficherotexto;
3
4 import java.io.File;
5 import java.io.BufferedReader;
6 import java.io.BufferedWriter;
7 import java.io.FileReader;
8 import java.io.FileWriter;
9 import java.io.IOException;
10 import java.util.Date;
11 import java.text.SimpleDateFormat;
12
13 public class ArreglaFicheroTexto {
14
15     public static void main(String[] args) {
16
17         String nomFichero = "f_texto.txt";
18         File f = new File(nomFichero);
19         if (!f.exists()) {
20             System.out.println("Fichero "+nomFichero+" no existe.");
21             return;
22         }
23
24         try (BufferedReader bfr = new BufferedReader(new FileReader(f))) {
25             File fTemp = File.createTempFile(nomFichero, "");
26             System.out.println("Creado fich. temporal "+fTemp.getAbsolutePath());
27             BufferedWriter bfw = new BufferedWriter(new FileWriter(fTemp));
28             String linea = bfr.readLine();
29             while (linea != null) { // En resumen, lee bfr y escribe en bfw
30                 boolean principioLinea = true, espacios = false, primerAlfab = false;
31                 for (int i = 0; i < linea.length(); i++) {
32                     char c = linea.charAt(i);
33                     if (Character.isWhitespace(c)) {
34                         if (!espacios && !principioLinea) {
35                             bfw.write(c);
36                             espacios = true;
37                         }
38                     } else if (Character.isAlphabetic(c)) {
39                         if (!primerAlfab) {
40                             bfw.write(Character.toUpperCase(c));
41                             primerAlfab = true;
42                         }
43                     } else {
44                         bfw.write(c);
45                         espacios = false;
46                         principioLinea = false;
47                     }
48                 }
49                 bfw.newLine();
50                 linea = bfr.readLine();
51             }
52             bfw.close();
53             //Copia de seguridad
54             f.renameTo(new File(nomFichero+"."+new SimpleDateFormat("yyyyMMddHHmmss").format(new Date())+".bak"));
55             fTemp.renameTo(new File(nomFichero));
56         } catch (IOException e) {
57             System.out.println(e.getMessage());
58         } catch (Exception e) {
59             e.printStackTrace();
60         }
61     }
62 }

```

Comprobamos que se genera un fichero temporal (en la ruta predefinida para ficheros temporales) con las mejoras realizadas:

[illegible]

Formas de acceso a un fichero

Hay dos formas de acceder a un fichero:

1. **Acceso secuencial:**
 - los datos o líneas de caracteres de un fichero se leen en orden
 - sólo es posible insertar nuevos datos al final del fichero
2. **Acceso aleatorio:**
 - los datos se almacenan en registros (filas) de tamaño conocido
 - se puede acceder directamente a cualquier dato del fichero
 - se puede acceder en cualquier orden a todos los datos del fichero
 - se puede modificar cualquier dato del fichero

Clases para gestión de flujos de datos desde/hacia ficheros

Como hemos visto en Java los ficheros son un caso particular de flujos de entrada o salida de datos o caracteres.

Las clases específicas para ficheros que hemos visto son:

- **FileInputStream** → gestiona ficheros binarios de entrada
- **FileOutputStream** → gestiona ficheros binarios de salida
- **FileReader** → gestiona ficheros de texto de entrada
- **FileWriter** → gestiona ficheros de texto de salida
- **BufferedReader** (deriva de **FileReader**) → permite leer líneas de texto con el método **readLine()**
- **BufferedWriter** (deriva de **FileWriter**) → permite escribir líneas de texto con el método **newLine()**

A continuación estudiaremos la clase **RandomAccessFile** para el manejo de ficheros de acceso aleatorio

Operaciones básicas sobre ficheros de acceso secuencial

- En **Java** el acceso secuencial se lleva a cabo con clases que ya conocemos:
 - **FileInputStream**
 - **FileOutputStream**
 - **FileReader**
 - **FileWriter**
 - **BufferedReader**
 - **BufferedWriter**

Operaciones básicas sobre ficheros de acceso aleatorio

- En **Java** el acceso aleatorio se lleva a cabo con la clase **RandomAccessFile**
 1. La clase **RandomAccessFile** no deriva de **File** y por tanto la gestión de un objeto de esta clase es completamente diferente
 2. Para leer y escribir datos en un objeto **RandomAccessFile** tiene sus propios métodos de lectura y escritura
 3. Tiene múltiples operaciones **lectura** y **escritura** binarias y algunas para texto:
 - a. **readLine()** → lectura de texto
 - b. **writeChar()** → escritura de texto

Métodos más importantes de **RandomAccessFile**

RandomAccessFile (File file, String mode)	Abre el fichero en alguno de estos modos: <ul style="list-style-type: none">• r: lectura• rw: lectura y escritura• rwd: escritura síncrona de datos• rws: escritura síncrona de datos y metadatos
RandomAccessFile (String file, String mode)	
void close()	Cierra el fichero
void seek(long pos)	Posiciona el puntero en la posición indicada
int skipBytes(int n)	Intenta avanzar el número de bytes indicado y devuelve el número de bytes que realmente se ha avanzado (será inferior al valor indicado si se ha llegado al final del fichero)
int read()	Lee y devuelve 1 byte del fichero
int read(byte[] buffer)	Lee bytes del fichero hasta rellenar el array y devuelve el número de bytes leídos
int read(byte[] buffer, int offset, int longitud)	Lee el contenido del array en el fichero desde la posición offset hasta la posición longitud y devuelve el número de bytes leídos
void readFully(byte[] buffer)	Lee bytes del fichero hasta rellenar el array pero si se llega al final del fichero arroja una excepción IOException
void readFully(byte[] buffer, int offset, int longitud)	Lee el contenido del array en el fichero desde la posición offset hasta la posición longitud pero si se llega al final del fichero arroja una excepción IOException

int readInt()	Lee un dato de tipo int
double readDouble()	Lee un dato de tipo double
String readLine()	Lee hasta el final de la línea en el texto actual
void write(int b)	Escribe 1 byte en el fichero
void write(byte[] buffer)	Escribe el contenido del array en el fichero
void write(byte[] buffer, int offset, int longitud)	Escribe el contenido del array en el fichero desde la posición offset hasta la posición longitud
void writeInt(int b)	Escribe un dato de tipo int
void writeDouble(double b)	Escribe un dato de tipo double
void writeChars(String s)	Escribe un String como cadena de caracteres

Ejemplo 1: Clase `EscribirFichAleatorio`

- Creamos un objeto **RandomAccessFile** con permisos de lectura y escritura
- Usamos los métodos **writeInt()**, **writeChars()** y **writeDouble()** para insertar datos en el fichero binario *AleatorioEmple.dat*
- Todas las filas y campos del fichero van a tener el **mismo tamaño**

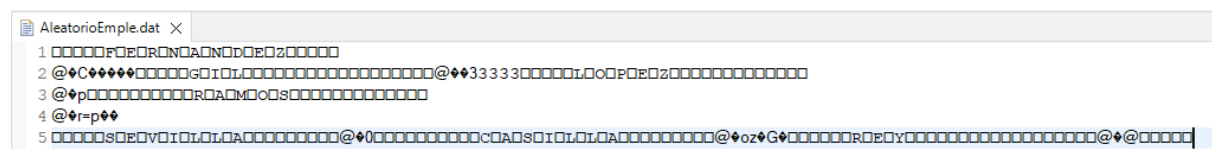
Nota: observamos que escribimos datos de diferentes tipos en el fichero (**int**, **String** y **double**) es decir no insertamos únicamente texto → se trata de un **fichero binario**

```

1 import java.io.*;
2
3 public class EscribirFichAleatorio {
4
5     public static void main(String[] args) throws IOException {
6         File fichero = new File("AleatorioEmple.dat");
7         //declaramos el objeto de acceso aleatorio
8         RandomAccessFile raf = new RandomAccessFile(fichero, "rw");
9
10        //arrays con los datos
11        String apellido[] = {"FERNANDEZ", "GIL", "LOPEZ", "RAMOS", "SEVILLA", "CASILLA", "REY"}; //apellidos
12        int dep[] = {10, 20, 10, 10, 30, 30, 20}; //departamentos
13        Double salario[] = {1000.45, 2400.60, 3000.0, 1500.56, 2200.0, 1435.87, 2000.0}; //salarios
14
15        StringBuffer sf = null; //buffer para almacenar el apellido
16        int n = apellido.length; //número de elementos del array
17
18        for (int i=0; i<n; i++) { //recorro los arrays
19            raf.writeInt(i+1); //usamos i+1 para identificar empleado
20
21            sf = new StringBuffer(apellido[i]);
22            sf.setLength(10); //10 caracteres para el apellido
23            raf.writeChars(sf.toString()); //insertar apellido
24            raf.writeInt(dep[i]); //insertar departamento
25            raf.writeDouble(salario[i]); //insertar salario
26        }
27        raf.close(); //cerramos el fichero
28    }
29 }

```

Si intentamos visualizar el archivo observamos que los datos no se han almacenado codificados como texto:



Ejemplo 2: Clase EscribirFichAleatorio

- Abrimos con el fichero **AleatorioEmple.dat** creado en el ejemplo anterior
- Creamos un objeto **RandomAccessFile** con permisos de lectura sobre el fichero **AleatorioEmple.dat**
- Con el método **seek()** nos ubicamos en la primera posición del fichero y en un bucle vamos leyendo cada campo de cada registro
 1. Leemos el campo **número de registro** con **readInt()**
 2. En un bucle **for** leemos todos los caracteres del campo **apellidos** con **readChar()**
 3. Leemos el campo **departamento** con **readInt()**
 4. Leemos el campo **departamento** con **readDouble()**
 5. Mostramos por pantalla el registro leído
- **Adelantamos** la posición del puntero en **36 bytes** y leemos el **siguiente registro** del fichero en el bucle

```
1 import java.io.*;
2
3 public class LeerFichAleatorio {
4     public static void main(String[] args) throws IOException {
5         //declaramos el fichero de acceso aleatorio
6         File fichero = new File("AleatorioEmple.dat");
7         //creamos un RandomAccessFile para lectura
8         RandomAccessFile raf = new RandomAccessFile(fichero, "r");
9         try {
10             int id, dep, posicion;
11             Double salario;
12             char apellido[] = new char[10], aux;
13             posicion = 0; //comenzamos aleatoriamente al principio
14
15             for(;;) { //recorremos el fichero
16                 raf.seek(posicion);
17                 id = raf.readInt();
18
19                 for (int i = 0; i < apellido.length; i++) {
20                     aux = raf.readChar();
21                     apellido[i] = aux;
22                 }
23
24                 //Convertimos el String en array
25                 String apellidos = new String(apellido);
26                 dep = raf.readInt();
27                 salario = raf.readDouble();
28
29                 if (id > 0)
30                     System.out.printf("ID: %s, Apellido: %s, Departamento: %d, Salario: %.2f %n", id, apellidos.trim(), dep, salario);
31
32                 //Accedemos aleatoriamente al siguiente registro
33                 posicion = posicion + 36;
34
35                 //Si hemos llegado al final salimos del for
36                 if (raf.getFilePointer() == raf.length()) break;
37             }
38         } catch (Exception e) {
39             e.printStackTrace();
40         } finally {
41             raf.close();
42         }
43     }
44 }
```

Que nos mostrará por pantalla el contenido del fichero **AleatorioEmple.dat**

```
<terminated> LeerFichAleatorio [Java Application] C:\Users\Fernando\.p2\pool\plugins\org.eclipse.justj.openjdk
ID: 1, Apellido: FERNANDEZ, Departamento: 10, Salario: 1000,45
ID: 2, Apellido: GIL, Departamento: 20, Salario: 2400,60
ID: 3, Apellido: LOPEZ, Departamento: 10, Salario: 3000,00
ID: 4, Apellido: RAMOS, Departamento: 10, Salario: 1500,56
ID: 5, Apellido: SEVILLA, Departamento: 30, Salario: 2200,00
ID: 6, Apellido: CASILLA, Departamento: 30, Salario: 1435,87
ID: 7, Apellido: REY, Departamento: 20, Salario: 2000,00
```

Trabajo con ficheros XML: analizadores sintácticos (parser) y vinculación (binding)

XML (eXtensible Markup Language - Lenguaje de Etiquetado Extensible)

- es un **metalenguaje**, es decir, un lenguaje para la definición de lenguajes de marcado
- permite **jerarquizar** y **estructurar** la información y **describir** los contenidos **dentro del propio documento**
- los ficheros XML **son ficheros de texto** escritos **en lenguaje XML**
- la información está organizada de forma **secuencial** y en **orden jerárquico**
- existen una serie de **marcas especiales**
 - Ejemplo: los símbolos < y > que se usan para delimitar las marcas que dan la estructura al documento
- cada marca tiene un nombre y puede tener 0 o más atributos

Un fichero XML sencillo tiene la siguiente estructura (lo llamamos **Empleados.xml**)

```
1  <?xml version="1.0"?>
2  <Empleados>
3      <empleado>
4          <id>1</id>
5          <apellido>FERNANDEZ</apellido>
6          <dep>10</dep>
7          <salario>1000.45</salario>
8      </empleado>
9      <empleado>
10         <id>2</id>
11         <apellido>GIL</apellido>
12         <salario>2400.6</salario>
13         <dep>20</dep>
14     </empleado>
15     <empleado>
16         <id>3</id>
17         <apellido>LOPEZ</apellido>
18         <salario>3000.0</salario>
19         <dep>10</dep>
20     </empleado>
21 </Empleados>
```

Los ficheros XML tienen muchos usos:

- **proporcionar datos** a una **BDD**
- para almacenar **copias** de partes del contenido de la **BDD**
- para escribir **ficheros de configuración** de programas
- en el protocolo **SOAP** (Simple Object Access Protocol)
- para **ejecutar comandos en servidores remotos** la información enviada al servidor remoto y el resultado de la ejecución del comando se envían en ficheros XML.

Analizadores sintácticos (parser)

Para leer los ficheros XML y acceder a su contenido y estructura, se utiliza un **analizador sintáctico** de XML o **parser**. El analizador lee los documentos y proporciona acceso a su contenido y estructura.

Los **analizadores sintácticos** más empleados son **DOM** (Modelo de Objetos de Documento) y **SAX** (API Simple para XML): son **independientes del lenguaje de programación** y existen versiones particulares para Java, Visual Basic, C, etc.

Son muy diferentes:

- **DOM**
 - Almacena toda la estructura del documento en memoria en forma de **árbol con nodos padre, nodos hijo y nodos finales** (que son aquellos que no tienen descendientes). Una vez creado el árbol, se van recorriendo los diferentes nodos y se analiza a qué tipo particular pertenecen.
 - Tiene su origen en el **W3C**. Este tipo de procesamiento
 - Necesita **más recursos** de memoria y tiempo sobre todo si los ficheros XML a procesar son bastante grandes y complejos
- **SAX**
 - 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 la lectura.
 - Cada **evento** invoca un **método definido** por el programador.
 - Prácticamente **no consume memoria**
 - Impide tener una visión global del documento por el que navegar

Parsing con DOM

Para poder trabajar con DOM en Java necesitamos métodos para cargar documentos desde una fuente de datos (fichero, InputStream, ...) contenidos en las clases e interfaces en:

- el paquete **org.w3c.dom** contenido en el **JSDK** (Java Servlet Developers Kit)
- el paquete **javax.xml.parsers** del API estándar de Java que proporciona un par de clases abstractas que toda implementación DOM para Java debe extender:
 - **DocumentBuilder**
 - **DocumentBuilderFactory**

Algunas de las interfaces son:

- **Document**: permite asociar **un documento XML a un objeto**
- **Element**: permite asociar **cada elemento** de un documento XML a **un objeto** (cada elemento tendrá unos métodos y atributos que se asociarán al objeto)
- **Node**: permite acceder a **cualquier nodo** del documento XML
- **NodeList**: permite acceder a la **lista de nodos hijo** de otro nodo
- **Attr**: permite acceder a los **atributos** (propiedades) de cada nodo
- **Text**: permite acceder a los datos de texto de un nodo
- **CharacterData**: permite acceder a los datos de carácter de un nodo y proporciona atributos y métodos para cada carácter
- **DocumentType**: permite acceder a la información contenida en la etiqueta **<!DOCTYPE>**

Ejemplo: vamos a utilizar el fichero XML de empleados del ejemplo anterior

1. Importamos los paquetes necesarios
2. Creamos tres objetos del paquete **javax.xml.parsers**
 - a. una instancia **factory** de **DocumentBuilderFactory** que nos pueda fabricar un parser
 - b. una instancia **builder** de **DocumentBuilder** (debe ir dentro de un bloque try-catch para capturar la excepción **ParserConfigurationException**) que realiza el parsing del documento XML
 - c. una instancia **document** de **Document** que tendrá el **fichero XML parseado**
3. El método **getDocumentElement()** del objeto **document** nos permite acceder al nodo raíz del fichero XML
4. El método **getElementsByTagName()** del objeto **document** nos permite acceder a todos los nodos "empleado" del fichero XML
5. Realizamos un bucle para recorrer la lista de nodos que llamamos **NodeList**
 - a. El método **item()** de **NodeList** nos permite acceder a cada nodo (cada empleado) individual
 - b. El método **getTextContent()** de **Node** nos permite obtener el contenido de cada etiqueta de un nodo

```

LecturaEmpleadoXML.java X
1 import org.w3c.dom.*;
2 import javax.xml.parsers.*;
3
4 public class LecturaEmpleadoXML {
5
6     public static void main(String args[]) {
7
8         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
9
10        try {
11            DocumentBuilder builder = factory.newDocumentBuilder();
12            Document document = builder.parse("Empleados.xml");
13            document.getDocumentElement().normalize();
14
15            // %s --> cualquier tipo    %n --> salto de línea
16            System.out.printf("Elemento raíz: %s %n", document.getDocumentElement().getNodeName());
17
18            // crea una lista con todos los nodos del empleado
19            NodeList empleados = document.getElementsByTagName("empleado");
20            // %d --> int                %n --> salto de línea
21            System.out.printf("Nodos empleado a recorrer: %d %n", empleados.getLength());
22
23            for (int i = 0; i < empleados.getLength(); i++) {
24                Node emple = empleados.item(i); //obtener un nodo empleado
25                if (emple.getNodeType() == Node.ELEMENT_NODE) { //tipo de nodo
26                    //obtener los elementos del nodo
27                    Element elemento = (Element) emple;
28                    System.out.printf("ID = %s %n", elemento.getElementsByTagName("id").item(0).getTextContent());
29                    System.out.printf(" * Apellido = %s %n", elemento.getElementsByTagName("apellido").item(0).getTextContent());
30                    System.out.printf(" * Departamento = %s %n", elemento.getElementsByTagName("dep").item(0).getTextContent());
31                    System.out.printf(" * Salario = %s %n", elemento.getElementsByTagName("salario").item(0).getTextContent());
32                }
33            }
34        } catch (Exception e) {
35            {e.printStackTrace();}
36        } //fin de main
37
38    } //fin de la clase

```

Obtenemos este resultado tras la ejecución:

```

Problems @ Javadoc Declaration Console X
<terminated> LecturaEmpleadoXML [Java Application] C:\Users\Fernando\
Elemento raíz: Empleados
Nodos empleado a recorrer: 3
ID = 1
 * Apellido = FERNANDEZ
 * Departamento = 10
 * Salario = 1000.45
ID = 2
 * Apellido = GIL
 * Departamento = 20
 * Salario = 2400.6
ID = 3
 * Apellido = LOPEZ
 * Departamento = 10
 * Salario = 3000.0

```

Parsing con SAX

SAX es una API escrita 100% en **Java** e incluida dentro del JRE

- permite analizar los documentos de forma secuencial (no carga todo el fichero en memoria como hace **DOM**)
 - **Ventajas:**
 - poco consumo de memoria aunque los documentos sean de gran tamaño
 - podemos crearnos nuestro propio parser
 - **Inconvenientes:**
 - más difícil de programar
 - no proporciona una visión global del documento (como sí proporciona DOM)

Funciona con **métodos** que **se ejecutan cuando se dispara un evento** a medida que se va leyendo el documento XML, algunos de estos métodos son:

- **startDocument()** → cuando se dispara la lectura de la etiqueta de **inicio del documento**
- **endDocument()** → cuando se dispara la lectura de la etiqueta de **fin del documento**
- **startElement()** → cuando se dispara la lectura de la etiqueta de **inicio de un elemento**
- **endElement()** → cuando se dispara la lectura de la etiqueta de **fin de un elemento**
- **characters()** → cuando se dispara la lectura de **caracteres entre etiquetas**

Ejemplo: vamos a parsear el contenido del documento XML llamado **alumnos.xml** (a la derecha de cada etiqueta vemos el evento que se dispara)

alumnos.xml	Métodos asociados a los eventos
<pre><?xml version="1.0"?> <listadealumnos> <alumno> <nombre> Juan </nombre> <edad> 19 </edad> </alumno> <alumno> <nombre> Maria </nombre> <edad> 20 </edad> </alumno> </listadealumnos></pre>	<pre>startDocument() startElement() startElement() startElement() characters() endElement() startElement() characters() endElement() startElement() characters() endElement() endElement() endDocument()</pre>

Creamos en el fichero PruebaSax1.java dos clases:

- **GestionContenido** que deriva de **org.xml.sax.helpers.DefaultHandler** (la clase que implementa los métodos para parsear un fichero con SAX) → implementaremos nuestro parseador personalizado
 - La clase **DefaultHandler** contiene los métodos que nosotros implementaremos en nuestro parseador
 - **startDocument()**
 - **endDocument()**
 - **startElement()**
 - **endElement()**
 - **characters()**
- **PruebaSax1**
 - crea un objeto procesador XML de la clase **XMLReader**
 - crea un objeto **GestionContenido** al que le pasaremos el fichero **alumnos.xml** para que lo parsee como queremos

El fichero completo queda así:

```
1 import java.io.*;
2 import org.xml.sax.Attributes;
3 import org.xml.sax.InputSource;
4 import org.xml.sax.SAXException;
5 import org.xml.sax.XMLReader;
6 import org.xml.sax.helpers.DefaultHandler;
7 import org.xml.sax.helpers.XMLReaderFactory; //deprecated
8
9 public class PruebaSax1 {
10     public static void main(String[] args) throws FileNotFoundException, IOException, SAXException
11     {
12         XMLReader procesadorXML = XMLReaderFactory.createXMLReader(); //deprecated
13         GestionContenido gestor = new GestionContenido();
14         procesadorXML.setContentHandler(gestor);
15         InputSource fileXML = new InputSource("alumnos.xml");
16         procesadorXML.parse(fileXML);
17     }
18 } //fin PruebaSax1
19
20 class GestionContenido extends DefaultHandler {
21     public GestionContenido() {
22         super();
23     }
24     public void startDocument() {
25         System.out.println("Comienzo del Documento XML");
26     }
27     public void endDocument() {
28         System.out.println("Final del Documento XML");
29     }
30     public void startElement(String uri, String nombre, String nombreC, Attributes atts) {
31         System.out.printf("\tPrincipio Elemento: %s %n", nombre);
32     }
33     public void endElement(String uri, String nombre, String nombreC) {
34         System.out.printf("\tFin Elemento: %s %n", nombre);
35     }
36     public void characters(char[] ch, int inicio, int longitud) throws SAXException {
37         String car = new String(ch, inicio, longitud);
38         //quitar saltos de línea
39         car = car.replace("\n", "");
40         System.out.printf("\tCaracteres: %s %n", car);
41     }
42 } // fin GestionContenido
```

Nota: vemos que la clase **org.xml.sax.helpers.XMLReaderFactory** está obsoleta

Si ejecutamos el método main() de PruebaSax1 veremos el resultado de nuestro parseador:

```
Problems @ Javadoc Declaration Console X Error Log
<terminated> PruebaSax1 [Java Application] C:\Users\Fernando\.p2\pool\plugins\
Comienzo del Documento XML
    Principio Elemento: listadealumnos
    Caracteres:

    Principio Elemento: alumno
    Caracteres:

    Principio Elemento: nombre
    Caracteres:
        Juan

    Fin Elemento: nombre
    Caracteres:

    Principio Elemento: edad
    Caracteres:
        19

    Fin Elemento: edad
    Caracteres:

    Fin Elemento: alumno
    Caracteres:
```

```
Problems @ Javadoc Declaration Console X Error Log
<terminated> PruebaSax1 [Java Application] C:\Users\Fernando\.p2\pool\plugins\
    Principio Elemento: alumno
    Caracteres:

    Principio Elemento: nombre
    Caracteres:
        Maria

    Fin Elemento: nombre
    Caracteres:

    Principio Elemento: edad
    Caracteres:
        20

    Fin Elemento: edad
    Caracteres:

    Fin Elemento: alumno
    Caracteres:

    Fin Elemento: listadealumnos
Final del Documento XML
```

Nota: si no queremos usar la clase obsoleta **org.xml.sax.helpers.XMLReaderFactory** podemos usar la clase PruebaSax2 en su lugar

```
1@import java.io.*;
2 import org.xml.sax.InputSource;
3 import org.xml.sax.SAXException;
4 import org.xml.sax.XMLReader;
5 import javax.xml.parsers.SAXParserFactory;
6 import javax.xml.parsers.SAXParser;
7 import javax.xml.parsers.ParserConfigurationException;
8
9
10 public class PruebaSax2 {
11
12@   public static void main(String[] args) throws FileNotFoundException, IOException, SAXException, ParserConfigurationException
13   {
14       GestionContenido gestor = new GestionContenido();
15
16       SAXParserFactory parserFactory = SAXParserFactory.newInstance();
17       SAXParser parser = parserFactory.newSAXParser();
18       XMLReader procesadorXML = parser.getXMLReader();
19
20       procesadorXML.setContentHandler(gestor);
21       InputSource fileXML = new InputSource("alumnos.xml");
22       procesadorXML.parse(fileXML);
23   }
24 }//fin PruebaSax2
```

Vinculación (binding)

- Hacer **binding** de un objeto a XML es "conectar" propiedades de ese objeto a etiquetas del fichero XML
- En **Java** podemos usar el paquete **java.xml.transform**

Ejemplo: vamos a utilizar el fichero **AleatorioEmple.dat** de empleados de un ejemplo anterior para transformar un fichero binario en un fichero XML

- fichero binario **AleatorioEmple.dat** → fichero XML **EmpleadosAmpliado.xml**

1. Importamos los paquetes necesarios:

```
1 import org.w3c.dom.*;
2 import javax.xml.parsers.*;
3 import javax.xml.transform.*;
4 import javax.xml.transform.dom.*;
5 import javax.xml.transform.stream.*;
6 import java.io.*;
7
8 public class CrearEmpleadoXML {
9
10 }
```

2. Creamos dos objetos del paquete **javax.xml.parsers**

- a. una instancia **factory** de **DocumentBuilderFactory** que nos pueda fabricar un parser
- b. el parser que se llamará **builder** y será una instancia de **DocumentBuilder** (debe ir dentro de un bloque try-catch para capturar la excepción **ParserConfigurationException**)

```
8 public class CrearEmpleadoXML {
9
10     public static void main(String args[]) throws IOException {
11         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
12
13         try {
14             DocumentBuilder builder = factory.newDocumentBuilder();
15         }
16         catch (Exception e) {System.out.println("Error: "+e);}
17     }
18
19 }
```

3. Creamos un **objeto documento** vacío
 - a. Creamos primero una instancia de la clase **DOMImplementation** para poder crear objetos de tipo **Document** con nodo raíz
 - b. A continuación creamos el objeto **document** de la clase **Document** dándole el nombre **Empleados** al nodo raíz
 - c. Le decimos al documento la **versión de XML** que necesitamos usar

```

13     try {
14         DocumentBuilder builder = factory.newDocumentBuilder();
15         DOMImplementation implementation = builder.getDOMImplementation();
16         Document document = implementation.createDocument(null, "Empleados", null);
17         document.setXmlVersion("1.0");
18     }

```

4. El siguiente paso sería recorrer el fichero con los datos de los empleados y por cada registro crear un **nodo** empleado

```

//creamos el nodo empleado
Element raiz = document.createElement("empleado");
//lo pegamos a la raíz del documento
document.getDocumentElement().appendChild(raiz);

```

5. El siguiente paso sería asociar a **cada nodo empleado 4 hijos**: *id*, *apellido*, *dep* y *salario* (por ejemplo: 1, FERNANDEZ, 10, 1000.45)

```

//añadir ID
CrearElemento("id", Integer.toString(id), raiz, document);
//Apellido
CrearElemento("apellido", apellidos.trim(), raiz, document);
//añadir DEP
CrearElemento("dep", Integer.toString(dep), raiz, document);
//añadir salario
CrearElemento("salario", Double.toString(salario), raiz, document);

```

6. El método **CrearElemento()** se encarga de
 - a. crear cada uno de los nodos hijos usando el método **createElement()** de **Document**:

```
Element elem = document.createElement(datoEmple);
```

- b. darle valor a los nodos hijos usando el método **createTextNode()** de **Text**:

```
Text text = document.createTextNode(valor); //damos valor
```

- c. añadir cada uno de los nodos hijo al nodo raíz y añadir el valor a cada uno de los nodos hijo (ambos con **appendChild**)

```
Transformer transformer = TransformerFactory.newInstance().newTransformer();
transformer.transform(source, result);
```

7. Una vez generados todos los nodos los volcamos al fichero de destino

```

Source source = new DOMSource(document);
Result result = new StreamResult(new java.io.File("EmpleadosAmpliado.xml"));
Transformer transformer = TransformerFactory.newInstance().newTransformer();
transformer.transform(source, result);

```


La clase completa queda así:

```
1 import org.w3c.dom.*;
2 import javax.xml.parsers.*;
3 import javax.xml.transform.*;
4 import javax.xml.transform.dom.*;
5 import javax.xml.transform.stream.*;
6
7 import java.io.*;
8
9 public class CrearEmpleadoXML {
10
11     public static void main(String args[]) throws IOException {
12         File fichero = new File("AleatorioEmple.dat");
13         RandomAccessFile file = new RandomAccessFile(fichero, "r");
14
15         int id, dep, posicion=0; //para situarnos al principio del fichero
16         Double salario;
17         char apellido[] = new char[10], aux;
18
19         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
20
21         try {
22             DocumentBuilder builder = factory.newDocumentBuilder();
23             DOMImplementation implementation = builder.getDOMImplementation();
24             Document document = implementation.createDocument(null, "Empleados", null);
25             document.setXmlVersion("1.0");
26
27             for (;;) {
28                 file.seek(posicion); //nos posicionamos
29                 id = file.readInt(); //obtengo id de empleado
30                 for (int i = 0; i < apellido.length; i++) {
31                     aux = file.readChar();
32                     apellido[i] = aux;
33                 }
34                 String apellidos = new String(apellido);
35                 dep = file.readInt();
36                 salario = file.readDouble();
37
38                 if (id > 0) { //id válidos a partir de 1
39                     //creamos el nodo empleado
40                     Element raiz = document.createElement("empleado");
41                     //lo pegamos a la raíz del documento
42                     document.getDocumentElement().appendChild(raiz);
43
44                     //añadir ID
45                     CrearElemento("id", Integer.toString(id), raiz, document);
46                     //Apellido
47                     CrearElemento("apellido", apellidos.trim(), raiz, document);
48                     //añadir DEP
49                     CrearElemento("dep", Integer.toString(dep), raiz, document);
50                     //añadir salario
51                     CrearElemento("salario", Double.toString(salario), raiz, document);
52                 }
53                 posicion = posicion + 36; //nos posicionamos para el siguiente empleado
54
55                 if (file.getFilePointer() == file.length()) break;
56             } //fin del for que recorre el fichero
57
58             Source source = new DOMSource(document);
59             Result result = new StreamResult(new java.io.File("EmpleadosAmpliado.xml"));
60             Transformer transformer = TransformerFactory.newInstance().newTransformer();
61             transformer.transform(source, result);
62         }
63         catch (Exception e) {System.out.println("Error: "+e);}
64
65         file.close(); //cerrar el fichero
66     } //fin de main
67 }
```

```

68 //Inserción de los datos del empleado
69 static void CrearElemento(String datoEmple, String valor, Element raiz, Document document) {
70     Element elem = document.createElement(datoEmple);
71     Text text = document.createTextNode(valor); //damos valor
72     raiz.appendChild(elem); //pegamos el elemento hijo a la raíz
73     elem.appendChild(text); //pegamos el valor
74 }
75 } //fin de la clase

```

Si probamos la clase, podemos comprobar con el editor XML de Eclipse que se ha generado el fichero con 7 empleados:

EmpleadosAmpliado.xml	
Node	Content
xml	version="1.0" encoding="UTF-8" standalone="no"
Empleados	
empleado	
id	1
apellido	FERNANDEZ
dep	10
salario	1000.45
empleado	
id	2
apellido	GIL
dep	20
salario	2400.6
empleado	
id	3
apellido	LOPEZ
dep	10
salario	3000.0
empleado	
id	4
apellido	RAMOS
dep	10
salario	1500.56
empleado	
id	5
apellido	SEVILLA
dep	30
salario	2200.0
empleado	
id	6
apellido	CASILLA
dep	30
salario	1435.87
empleado	
id	7
apellido	REY
dep	20
salario	2000.0

También podemos visualizarlo con cualquier navegador web:

Este fichero XML no parece tener ninguna información de estilo asociada. Se muestra debajo el árbol del documento.

```
<Empleados>
  <empleado>
    <id>1</id>
    <apellido>FERNANDEZ</apellido>
    <dep>10</dep>
    <salario>1000.45</salario>
  </empleado>
  <empleado>
    <id>2</id>
    <apellido>GIL</apellido>
    <dep>20</dep>
    <salario>2400.6</salario>
  </empleado>
  <empleado>
    <id>3</id>
    <apellido>LOPEZ</apellido>
    <dep>10</dep>
    <salario>3000.0</salario>
  </empleado>
  <empleado>
    <id>4</id>
    <apellido>RAMOS</apellido>
    <dep>10</dep>
    <salario>1500.56</salario>
  </empleado>
  <empleado>
    <id>5</id>
    <apellido>SEVILLA</apellido>
    <dep>30</dep>
    <salario>2200.0</salario>
  </empleado>
  <empleado>
    <id>6</id>
    <apellido>CASILLA</apellido>
    <dep>30</dep>
    <salario>1435.87</salario>
  </empleado>
  <empleado>
    <id>7</id>
    <apellido>REY</apellido>
    <dep>20</dep>
    <salario>2000.0</salario>
  </empleado>
</Empleados>
```

Serialización de objetos a XML (marshalling)

- Al proceso de volcar o serializar un objeto a un fichero XML se le da el nombre de **marshalling**, al proceso contrario **unmarshalling**.

Librerías para conversión de documentos XML a otros formatos

sgfgsdgsdfdsdf.