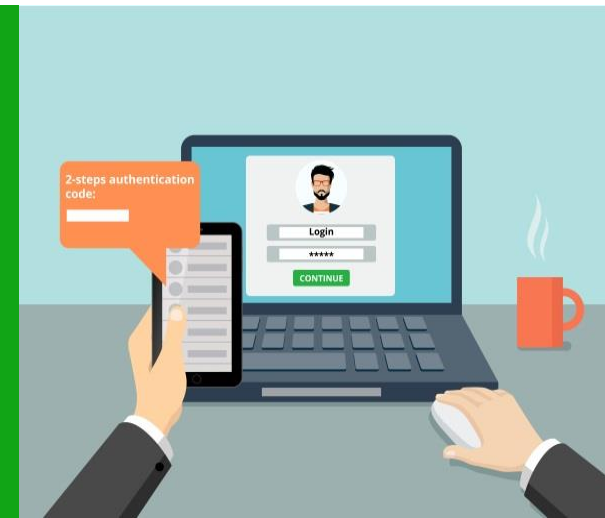


Módulo 6

Código: 486

Acceso a datos

Técnico Superior en Desarrollo de
Aplicaciones Multiplataforma



UNIDAD 1

Manejo de ficheros

Contenidos teóricos



1. Objetivos generales de la unidad
2. Competencias y contribución en la unidad
3. Contenidos conceptuales y procedimentales
4. Evaluación
5. Distribución de los contenidos
6. Sesiones

1. Objetivos generales de la unidad

Objetivos curriculares

1. Utilizar el API de Java para la gestión de ficheros y directorios.

2. Realizar las operaciones básicas sobre ficheros secuenciales y directos, utilizando el API de Java.


3. Procesar ficheros XML desde Java.

4. Trabajar con el formato de representación JSON.

5. Convertir ficheros XML e información representada en JSON, a otros formatos.

6. Gestión de excepciones.

2. Competencias y contribución en la unidad



e) Desarrollar aplicaciones multiplataforma con acceso a bases de datos utilizando lenguajes, librerías y herramientas adecuados a las especificaciones.

Utilizando clases específicas del API de Java, para realizar aplicaciones que gestionan la información almacenada en ficheros.
--

3. Contenidos

CONTENIDOS CONCEPTUALES		CONTENIDOS PROCEDIMENTALES
Gestión de ficheros	<ul style="list-style-type: none">• Introducción.• Clases Java para gestionar ficheros.	<ul style="list-style-type: none">• Introducción al concepto de fichero.• Ver las clases disponibles en el API de Java para gestionar ficheros y directorios.
Flujos o streams	<ul style="list-style-type: none">• Concepto de flujo o stream.• Tipos de flujos.• Clases Java para trabajar con los distintos tipos de flujos.	<ul style="list-style-type: none">• Ver el concepto de flujo o stream en Java.• Ver los diferentes tipos de flujos que se pueden presentar a la hora de trabajar con ficheros.• Conocer las clases que proporciona el API de Java para trabajar con los diferentes tipos de flujos.
Operativa con ficheros	<ul style="list-style-type: none">• Formas de acceso a un fichero.• Operaciones sobre ficheros.• Procesamiento de ficheros secuenciales.• Procesamiento de ficheros directos.• Gestión de excepciones	<ul style="list-style-type: none">• Identificar las formas de acceso secuencial y aleatoria sobre ficheros.• Trabajar con ficheros de caracteres y binarios, accediendo secuencialmente.• Trabajar con ficheros de caracteres y binarios, accediendo de forma directa.• Manejar las excepciones que pueden aparecer al procesar ficheros.

3. Contenidos

CONTENIDOS CONCEPTUALES		CONTENIDOS PROCEDIMENTALES
Ficheros XML	<ul style="list-style-type: none">• El lenguaje XML.• Procesamiento de ficheros XML.• Alternativa a XML: JSON• Conversión de XML y JSON a otros formatos.	<ul style="list-style-type: none">• Introducción al lenguaje XML.• Ver los dos modelos de parseo de documentos XML mas conocidos: DOM y SAX. Identificando las ventajas e inconvenientes de cada uno de ellos.• Trabajar con el formato de representación JSON como alternativa a los ficheros XML.• Convertir un documento XML y un JSON a otros formatos de representación de la información.

4. Evaluación

Se han utilizado clases para la gestión de ficheros y directorios.

Se han valorado las ventajas y los inconvenientes de las distintas formas de acceso.

Se han utilizado clases para recuperar información almacenada en un fichero XML.

Se han utilizado clases para almacenar información en un fichero XML.

Se han utilizado clases para convertir a otro formato información contenida en un fichero XML.

Se han previsto y gestionado las excepciones.

Se han probado y documentado las aplicaciones desarrolladas.

1. INTRODUCCIÓN

- Los programas necesitan comunicarse con su entorno, tanto para recoger datos e información que deben procesar como para devolver los resultados obtenidos.
- Esta fuente y destino de información puede ser un **fichero** o **archivo**.
- **Fichero**: conjunto de bits almacenados en un dispositivo como un disco duro, bajo un nombre único, con una extensión consistente en tres letras normalmente que determinan el tipo de fichero.
- Vamos a ver como podemos trabajar con los ficheros en lenguaje Java, realizando operaciones habituales como son crear un fichero, moverlo o copiarlo, leer y grabar información en él, etc.



https://www.psicologiauam.es/carmen.ximenez/TAgrupacion_4.html

2. CLASES JAVA PARA GESTIONAR FICHEROS

Clase File

- Representa un archivo o un directorio. Tiene los siguientes constructores:
`File(String nombre)`
`File(String directorio, String nombre)`
`File(File directorio, String nombre)`
- Ejemplos:
`File f1 = new File("c:\\windows\\notepad.exe"); // Un archivo`
`File f2 = new File("c:\\windows"); // Un directorio`
`File f3 = new File(f2, "notepad.exe"); // Es igual a f1`
- Como vemos para representar el carácter separador "\\" hay que escaparlo previamente con otro carácter "\\".
- Además, podemos utilizar rutas absolutas y rutas relativas. La ruta absoluta se indica desde la unidad de disco en la que se está trabajando y la relativa desde el directorio actual.

2. CLASES JAVA PARA GESTIONAR FICHEROS

Clase File

- Una vez construido un objeto **File** se pueden utilizar sus métodos para obtener información o para modificar el fichero o directorio al que representa.
- Si File representa un **archivo**, se pueden utilizar los siguientes métodos (entre otros):

Método	Función
<code>isFile()</code>	Devuelve true si el objeto File es un fichero
<code>length()</code>	Devuelve el tamaño del fichero en bytes
<code>canRead():</code>	Devuelve true si el fichero se puede leer
<code>canWrite():</code>	Devuelve true si el fichero se puede escribir
<code>delete()</code>	Borra el fichero
<code>lastModified()</code>	Fecha de la última modificación
<code>renameTo(File)</code>	Renombra el fichero

2. CLASES JAVA PARA GESTIONAR FICHEROS

Clase File

- Si File representa un **directorio**, se pueden utilizar los siguientes métodos (entre otros):

Método	Función
<code>isDirectory()</code>	Devuelve true si el objeto File es un directorio
<code>mkdir()</code>	Crea el directorio
<code>delete()</code>	Borrar el directorio
<code>String[]list()</code>	Lista el contenido del directorio

2. CLASES JAVA PARA GESTIONAR FICHEROS

Clase File

- Además disponemos de los siguientes métodos que devuelven el **path** del objeto File:

Método	Función
<code>getPath()</code>	Devuelve la ruta relativa
<code>getName()</code>	Devuelve el nombre del archivo
<code>getAbsolutePath()</code>	Devuelve la ruta absoluta
<code>getParent()</code>	Devuelve el directorio padre

2. CLASES JAVA PARA GESTIONAR FICHEROS

Clase File

- Ejemplo que muestra información de un fichero o directorio:

```
BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
File miF; // referencia al fichero o directorio
String nom; // nombre del fichero o directorio
try {
    do {
        System.out.println("Introduce [ruta]archivo o [ruta]directorio. ");
        System.out.print("Línea vacía para terminar: ");
        nom = teclado.readLine();
        if (nom.length() > 0) {
            miF = new File(nom);
            if (miF.exists()) {
                System.out.println(nom + " está en el disco. ");
                if (miF.isFile()) {
                    System.out.print("Es un fichero. ");
                    System.out.println("De tamaño: " + miF.length() + " bytes");
                } else if (miF.isDirectory()) {
                    System.out.println("Es un directorio. ");
                } else {
                    System.out.println("Se desconoce que es. ");
                }
            } else {
                System.out.println("No existe el objeto en el disco.");
            }
        }
    } while (nom.length() > 0);
} catch (IOException ex) {
    ex.printStackTrace();
}
```

2. CLASES JAVA PARA GESTIONAR FICHEROS

Clase File

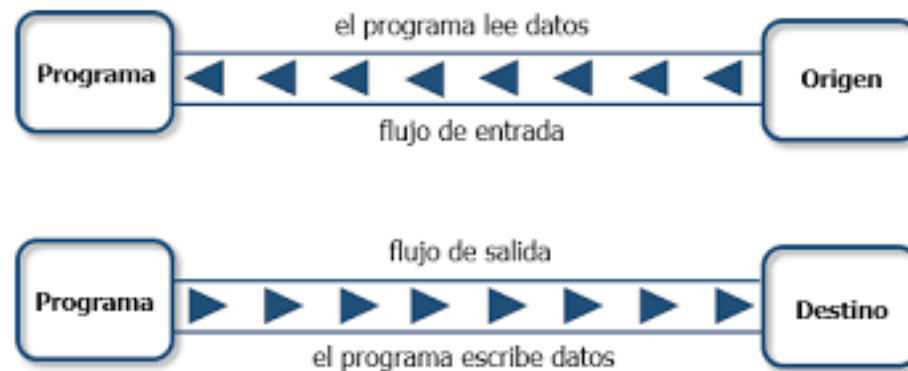
- Ejemplo que muestra un listado del directorio actual:

```
BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
File direct; // referencia al directorio
String dir[]; // array que almacena el nombre de los archivos y carpetas del directorio
try {
    System.out.print("Introduce [ruta]directorio: ");
    String nom = teclado.readLine();

    direct = new File(nom);
    if (direct.exists() && direct.isDirectory()) {
        // obtenemos la lista de elementos que contiene
        dir = direct.list();
        System.out.println("Elementos del directorio " + nom);
        for (String obj : dir) {
            System.out.println(obj);
        }
    } else {
        System.out.println("Directorio inexistente");
    }
} catch (IOException ex) {
    ex.printStackTrace();
}
```

3. CONCEPTO DE STREAM.

- La manera de representar estas entradas y salidas en Java es a base de **streams** (flujos de datos).
- **Stream**: objeto que actúa de intermediario entre el programa y la fuente o destino de los datos.
- Esta abstracción proporcionada por los flujos, hace que los programadores solo se tengan que preocupar de la forma de trabajar con estos objetos, sin importar el origen o destino concreto de los datos.
- Por ejemplo, cuando se quiere mostrar algo en pantalla se hace a través de un stream que conecta el monitor al programa. Se da a ese stream la orden de escribir algo y éste lo traslada a la pantalla. Este concepto es suficientemente general para representar la lectura/escritura de archivos, la comunicación a través de



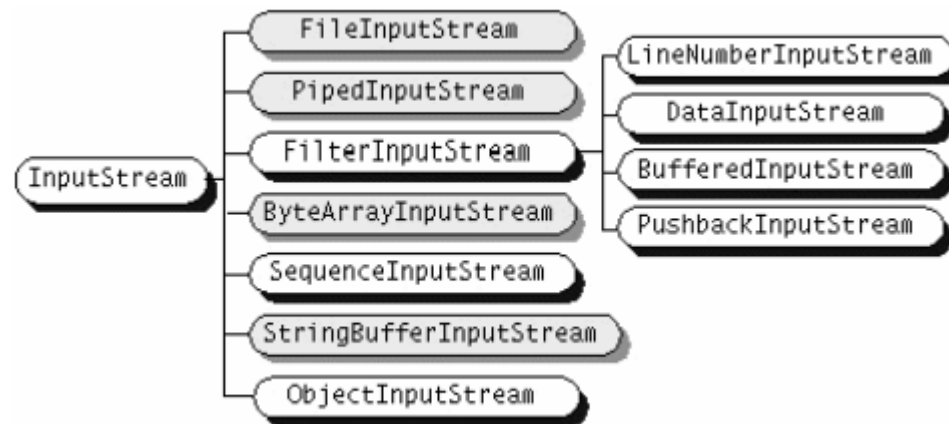
4. TIPOS DE STREAMS.

- Flujos de bytes (8 bits)
 - ✓ Operaciones de entrada y salida de bytes.
 - ✓ Se utiliza para leer/escribir datos binarios.
- Flujos de caracteres (16 bits)
 - ✓ Operaciones de entrada y salida de caracteres.
 - ✓ Se utiliza para leer/escribir cadenas de caracteres.
- A continuación, vamos a ver las clases que se utilizan en Java para trabajar con los 2 tipos de flujos, diferenciando si se van a realizar operaciones de lectura o escritura de información.

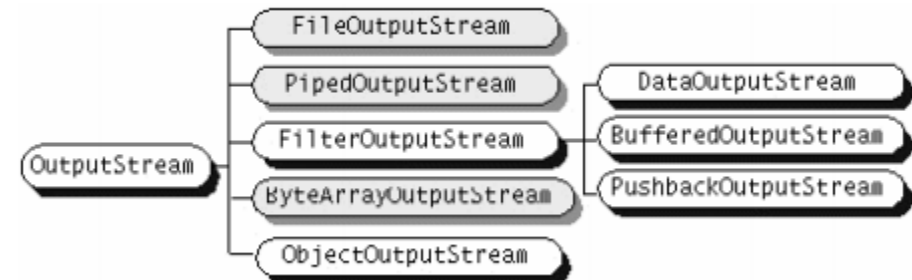
5. CLASES PARA TRABAJAR CON STREAMS.

- Flujos de bytes

Lectura

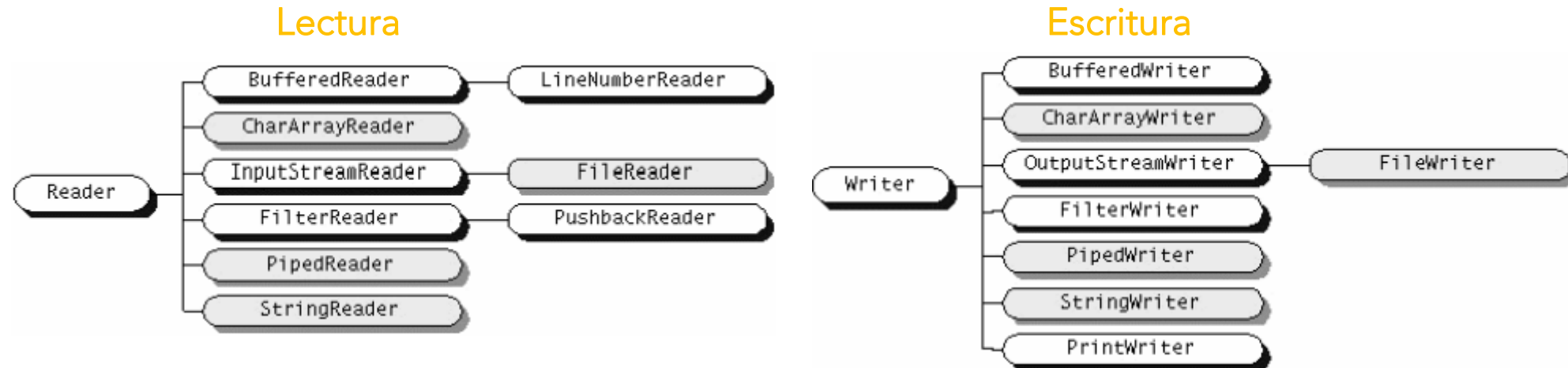


Escritura



5. CLASES PARA TRABAJAR CON STREAMS.

- Flujos de caracteres



5. CLASES PARA TRABAJAR CON STREAMS.

- En las imágenes anteriores las clases con fondo gris definen el dispositivo con que conecta el stream.
Las demás clases (fondo blanco) añaden características particulares a la forma de enviar/recibir datos del stream. La intención es que se combinen para obtener el comportamiento deseado.
- Ejemplo:

```
InputStreamReader entrada = new InputStreamReader(System.in);  
BufferedReader teclado = new BufferedReader (entrada);  
BufferedReader in = new BufferedReader(new FileReader("archivo.txt"));
```
- Las clases que nosotros vamos a utilizar para trabajar con ficheros son:
 - ✓ FileInputStream
 - ✓ FileOutputStream
 - ✓ FileReader
 - ✓ FileWriter
- Y siempre recomendable utilizar un buffer intermedio para aumentar el rendimiento de las operaciones.

6. FORMAS DE ACCESO A UN FICHERO.

a) Acceso secuencial

- Para acceder al registro i -ésimo es necesario pasar por los $(i-1)$ registros anteriores.
- La información es almacenada por orden de llegada.

b) Acceso directo o aleatorio

- Es posible acceder al registro i -ésimo sin necesidad de leer los registros anteriores.

En Java utilizaremos clases diferentes para acceder de forma secuencial o aleatoria a ficheros de caracteres y ficheros binarios.

Un **registro** es una agrupación heterogénea de datos. Cada uno de estos datos se llama *campo*.
Por ejemplo, puedo tener un registro con el siguiente formato:

nombre	apellidos	estado civil	edad
cadena	cadena	carácter	entero

Los campos pueden ser tipos simples (int, char, ...) o bien pueden ser otras estructuras de datos, incluso referencias a objetos.

7. OPERACIONES SOBRE FICHEROS.

- Las operaciones básicas a la hora de trabajar con ficheros son siempre las mismas, independientemente del tipo de fichero y de su forma de acceso.
- Estas operaciones son:
 1. Abrir fichero
 2. Procesar fichero para lectura o escritura.
 3. Cerrar fichero.
- El procesamiento del fichero consiste en las llamadas operaciones **CRUD** (*Create, Read, Update y Delete*), que también son las operaciones básicas cuando se trabaja con bases de datos:
 - ✓ **Create:** consiste en dar de alta (añadir) nuevos registros en el fichero.
 - ✓ **Read:** operación consistente en buscar un registro en el fichero.
 - ✓ **Update:** modificación de un registro del fichero.
 - ✓ **Delete:** consiste en dar de baja (eliminar) un registro del fichero.
- Lo que va a variar, va a ser la forma de realizar estas operaciones, en función del tipo de fichero en cuanto a su forma de acceso (ficheros secuenciales o ficheros directos).

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Escritura de texto.

- Se utiliza la clase `FileWriter`.
 - ✓ Los métodos `write` de esta clase permiten escribir caracteres en el fichero.
- **Ejemplo:** Programa que escribe frases introducidas por teclado en un archivo *frases.txt* ubicado en la carpeta *ficheros*.

```
BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
String s; // almacena la frase introducida por teclado

try {
    FileWriter fw = new FileWriter(".\\src\\ficheros\\frases.txt", true);
    BufferedWriter bw = new BufferedWriter(fw);

    //pedimos frases por teclado y las vamos escribiendo en el fichero
    System.out.print("Introduce una frase: ");
    s = teclado.readLine();
}
```

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Escritura de texto.

```
while (s.length() > 0) {  
    bw.write(s); //la almacenamos en el fichero  
    bw.newLine();  
    System.out.print("Introduce una frase: "); //pedimos nueva frase  
    s = teclado.readLine();  
}  
bw.close(); //cerramos el ficheros  
} catch (IOException e) {  
    System.out.println("Error de entrada/salida");  
    e.printStackTrace();  
}
```

- Al crear un objeto `FileWriter` se abre el fichero para escritura, de forma que si no existe se crea, y si ya existe se sobrescribe su contenido.
- En el constructor de `FileWriter` se puede indicar un segundo parámetro booleano, que con valor `true` abre el fichero para añadir más datos no sobrescribiendo su contenido.
- Se utiliza un objeto **`BufferedWriter`** para realizar la escritura más eficiente. Ésta se realiza con el método `write()` que puede lanzar excepciones de tipo **`IOException`**.

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Escritura de texto.

- También podemos utilizar la clase **PrintWriter** que dispone de los métodos `print()` y `println()`, homólogos a los de `System.out`, para escribir texto en un fichero. Este objeto se crea a partir de un objeto `FileWriter`.
- Así el ejemplo anterior

```
try {
    FileWriter fw = new FileWriter(".\\src\\ficheros\\frases.txt", true);
    PrintWriter pw = new PrintWriter(fw);

    //pedimos frases por teclado y las vamos escribiendo en el fichero
    System.out.print("Introduce una frase: ");
    s = teclado.readLine();
    while (s.length() > 0) {
        pw.println(s); //la almacenamos en el fichero
        System.out.print("Introduce una frase: "); //pedimos nueva frase
        s = teclado.readLine();
    }
    pw.close(); //cerramos el ficheros
} catch (IOException e) {
    System.out.println("Error de entrada/salida");
    e.printStackTrace();
}
```

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Lectura de texto.

- Se utiliza la clase `FileReader`.
 - ✓ Los métodos `read` de esta clase permiten leer caracteres del fichero.
- **Ejemplo de lectura:** Programa que lee las frases del archivo *frases.txt* ubicado en la carpeta *ficheros* y las muestra por consola.

```
File f = new File(".\\src\\ficheros\\frases.txt");
try {
    FileReader fr = new FileReader(f);
    BufferedReader br = new BufferedReader(fr);
    String s; //almacena cada linea leida del fichero
    s = br.readLine(); //lectura anticipada
```

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Lectura de texto.

- De la misma forma utilizamos un buffer (**BufferedReader**) para hacer más eficientes las lecturas, ya que utilizando este objeto podemos leer líneas completas con el método *readLine()*.
- Además, al crear un objeto **FileReader** se abre el fichero para lectura lanzando una excepción del tipo **FileNotFoundException** si el archivo no existe.

```
while (s != null) { //mientras haya cadenas por leer
    System.out.println(s); //la mostramos por pantalla
    s = br.readLine(); // leemos otra línea del fichero
}
br.close(); //cerramos el fichero
} catch (FileNotFoundException e) {
    System.out.println("Error al abrir el archivo");
} catch (IOException e) {
    System.out.println("Error de entrada/salida");
}
```

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Escritura de bytes.

- Se utiliza la clase `FileOutputStream`.
 - ✓ Sus métodos `write` se pueden utilizar para escribir bytes en el fichero, pero es más interesante utilizar la clase `DataOutputStream` que permite escribir datos primitivos de todo tipo, de forma independiente a la máquina.
- **Ejemplo:** Programa que almacena en un fichero binario llamado *aleatorios.dat* 100 números reales generados aleatoriamente.

```
File f = new File(".\\src\\ficheros\\aleatorios.dat");
Random r = new Random();

try {
    DataOutputStream dos = new DataOutputStream(
        new BufferedOutputStream(new FileOutputStream(f)));
```

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Escritura de bytes.

```
// Generamos 100 double aleatorios y les escribimos en el fichero
for (int i = 0; i < 100; i++) {
    dos.writeDouble((int) (10*r.nextDouble())); // Escritura del dato
}
dos.close(); // Cerramos el fichero
} catch (IOException e) {
    System.out.println("Error de escritura en el archivo");
    e.printStackTrace();
}
```

- Al crear un objeto `FileOutputStream` se abre el fichero para escritura, de forma que si no existe se crea, y si ya existe se sobrescribe su contenido.
- En el constructor de `FileOutputStream` se puede indicar un segundo parámetro booleano, que con valor `true` abre el fichero para añadir más datos no sobrescribiendo su contenido.
- Se utiliza un buffer intermedio para realizar la escritura de información más eficiente.

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Escritura de bytes.

- El objeto `DataOutputStream` se crea a partir del objeto `FileOutputStream`.
- Utiliza para escribir los números aleatorios el método `writeDouble`, porque los datos a escribir son reales. Existen métodos `write` para todos los tipos primitivos de java.
- Los métodos `write` pueden lanzar excepciones de tipo `IOException` que será necesario capturar.
- Son clases diseñadas para trabajar de manera conjunta. Una lee lo que la otra escribe como una secuencia de bytes.
- Estos ficheros binarios no son legibles directamente como ocurría con los ficheros de texto, que podíamos ver su contenido con un bloc de notas. De hecho, la información que grabamos en el fichero utilizando las clases anteriores, debe ser leída como una secuencia de bytes utilizando las clases homónimas de lectura, como veremos en el siguiente ejemplo.

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Lectura de bytes.

- Ejemplo: Programa que lee los números reales almacenados en el fichero binario *aleatorios.dat*.

```
boolean finArchivo = false; //controla el final del fichero
double d; // almacena el dato leído

File f = new File(".\\src\\Ficheros\\aleatorios.dat");

try {
    DataInputStream dis = new DataInputStream(
        new BufferedInputStream(new FileInputStream(f)));
    while (!finArchivo) { //mientras no sea el final del fichero
        d = dis.readDouble(); //leemos dato (double)
        System.out.println(d); //lo mostramos por pantalla
    }
    dis.close(); //cerramos el fichero
} catch (EOFException e) {
    finArchivo = true;
} catch (FileNotFoundException e) {
    System.out.println("Error. No se puede abrir un fichero inexistente");
} catch (IOException e) {
    System.out.println("Error de lectura en fichero");
}
```

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Lectura de bytes.

- Analizando el código anterior, observamos lo siguiente:
 - El objeto **DataInputStream** se crea a partir del objeto **FileInputStream**.
 - Este objeto utiliza el método *readDouble* para leer los números aleatorios porque se grabaron utilizando el método *writeDouble*.
 - Al crear un objeto **FileInputStream** se intenta abrir el fichero para lectura generando una excepción del tipo **FileNotFoundException** si el archivo no existe.
 - El bucle *while* utilizado para realizar la lectura completa del archivo, se ejecuta de forma indefinida hasta que se lance la excepción **EOFException** (*EOF – End Of File*).

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Operaciones de actualización

- **Altas**
 - ✓ Para añadir nuevos datos en un fichero secuencial, es necesario abrir el fichero en modo "append" utilizando los objetos adecuados según se trate de un fichero binario o de texto. De esta forma se conserva la información que tuviera y se graba la nueva al final del fichero, justo detrás de la información ya existente.
- **Bajas**
 - ✓ Para eliminar una información del fichero, será necesario utilizar un fichero auxiliar de apoyo.
 - ✓ El proceso será el siguiente:
 1. Mientras haya datos, leer registro del fichero original.
 2. Si el registro actual no se quiere dar de baja se escribe en el fichero auxiliar. Ir al punto 1.
 3. Sino, ir al punto 1.
 4. Eliminar el fichero original.
 5. Renombrar el fichero auxiliar con el nombre del fichero original.



Fin procesamiento
fichero

8. PROCESAMIENTO DE FICHEROS SECUENCIALES.

Operaciones de actualización

- Modificaciones

- ✓ Como ocurre en las bajas, también es necesario usar un fichero auxiliar.

- ✓ Proceso:



1. Mientras haya datos, leer registro del fichero original.

2. Si el registro actual no se quiere modificar se escribe en el fichero auxiliar. Ir al punto 1.

3. Sino, escribir registro modificado en el fichero auxiliar. Ir al punto 1.

→ Fin procesamiento

4. Eliminar el fichero original. **fichero**

5. Renombrar el fichero auxiliar con el nombre del fichero original.

9. SERIALIZACIÓN DE OBJETOS.

- La **serialización** de objetos consiste en transformar un objeto en una secuencia de bytes de tal manera que se represente el estado de dicho objeto.
- Una vez que tenemos serializado el objeto, esa secuencia de bits pueden ser posteriormente leídos para restaurar el objeto original.
- Para poder serializar un objeto, es necesario que su clase implemente la interfaz **java.io.Serializable**. Esta interfaz no declara ningún método miembro, se trata de una interfaz vacía.
- Veamos un ejemplo muy sencillo:

```
public class Persona implements Serializable {  
  
    private String nombre;  
    private int edad;  
  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
}
```

```
public String getNombre() { ...3 lines }  
  
public int getEdad() { ...3 lines }  
  
public void setNombre(String nombre) { ...3 lines }  
public void setEdad(int edad) { ...3 lines }  
  
@Override  
public String toString() { ...3 lines }
```

9. SERIALIZACIÓN DE OBJETOS.

- Simplemente con esto valdría, no tenemos que escribir nada más.
- Actuarían los métodos `defaultWriteObject` de la clase `ObjectOutputStream` y el homónimo, `defaultReadObject` de la clase `ObjectInputStream` para realizar la serialización y deserialización de los objetos `Persona` respectivamente.
- Veamos como:

```
public static void main(String[] args) {  
  
    FileOutputStream fos = null;  
    try {  
        fos = new FileOutputStream(".\\src\\ficheros\\personas.dat");  
        ObjectOutputStream oos = new ObjectOutputStream(fos);  
        oos.writeObject(new Persona("Juan", 25));  
        oos.writeObject(new Persona("Pepe", 50));  
        oos.writeObject(new Persona("María", 44));  
    } catch (FileNotFoundException ex) {  
        System.out.println("Fichero no encontrado.");  
    } catch (IOException ex) {  
        ex.printStackTrace();  
    } finally {  
        try {  
            fos.close();  
        } catch (IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

9. SERIALIZACIÓN DE OBJETOS.

- Como podemos ver en el código, tenemos que crear un flujo de salida creando un `FileOutputStream`, y después crear un `ObjectOutputStream` a partir de éste, que es el que procesará realmente los datos.
- A continuación, escribimos los objetos en el flujo de salida utilizando el método `writeObject` del objeto `ObjectOutputStream`, al que se le pasa como parámetro el objeto a escribir.
- Para leer los

```
public static void main(String[] args) {  
  
    FileInputStream fis = null;  
    try {  
        File f = new File(".\\src\\ficheros\\personas.dat");  
        if (f.exists()) {  
            fis = new FileInputStream(f);  
            ObjectInputStream ois = new ObjectInputStream(fis);  
            while (true) {  
                Persona p = (Persona) ois.readObject();  
                System.out.println(p.toString());  
            }  
        }  
    }  
}
```

9. SERIALIZACIÓN DE OBJETOS.

```
    } catch (EOFException ex) {  
        System.out.println("Fin fichero.");  
    } catch (ClassNotFoundException ex) {  
        System.out.println("Fichero no encontrado.");  
    } catch (FileNotFoundException ex) {  
        Logger.getLogger(LeerPersonas.class.getName()).log(Level.SEVERE, null, ex);  
    } catch (IOException ex) {  
        Logger.getLogger(LeerPersonas.class.getName()).log(Level.SEVERE, null, ex);  
    } finally {  
        try {  
            fis.close();  
        } catch (IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

- De forma análoga a la escritura, creamos un flujo de entrada creando un objeto **FileInputStream**, a continuación creamos un **ObjectInputStream** que vinculamos al primero, y utilizando el método **readObject** leemos todos los objetos **Persona** del flujo de entrada.

9. SERIALIZACIÓN DE OBJETOS.

- El proceso de serialización visto es suficiente para la mayoría de los casos, ahora bien, se puede personalizar para aquellos casos específicos.
- Para personalizar la serialización, es necesario “sobrecargar” los métodos **writeObject** y **readObject** de aquella clase cuyos objetos se quiera serializar. Utilizamos las comillas, porque realmente no es una sobrecarga, sino una nueva definición de los métodos.

- La definición de **writeObject** será:

```
private void writeObject (ObjectOutputStream s) throws IOException{  
    s.defaultWriteObject();  
    //código para escribir datos  
}
```

- El método **readObject** ha de leer todo lo que se ha escrito con **writeObject**, y además, puede realizar otras tareas necesarias para actualizar el estado del objeto:

```
private void readObject (ObjectInputStream s) throws IOException {  
    s.defaultReadObject();  
    //código para leer datos  
}
```

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

- La clase `RandomAccessFile` dispone de métodos para procesar archivos de acceso directo.
- Esta clase no es parte de la jerarquía `InputStream/OutputStream` ya que su comportamiento es totalmente distinto.
- Para crear un objeto `RandomAccessFile` podemos usar cualquiera de los siguientes constructores:
`new RandomAccessFile(File fichero, String modo)`
`new RandomAccessFile(String fichero, String modo)`
- El 2º argumento es el modo de apertura del fichero. Puede tener los siguientes valores:
 - "r" **Modo de sólo lectura.** Sólo se pueden leer registros, no se pueden modificar o grabar registros.
 - "rw" **Modo de lectura/escritura.** Permite hacer operaciones tanto de lectura como de escritura de registros.
- Si el fichero no existe y se abre en modo de sólo lectura ("r") se lanzará una `FileNotFoundException`, y si se abre en modo de lectura y escritura ("rw") se creará un fichero nuevo.
- Esta clase maneja un **puntero** que indica la posición actual en el fichero. Cuando el fichero se abre el puntero se coloca en el byte 0.

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

Métodos de posicionamiento

- Las operaciones de lectura y escritura, se realizan a partir de la posición en la que se encuentre el puntero del fichero.
- Una vez realizada la operación, el puntero queda situado justo después del último byte leído o escrito. Si por ejemplo, el puntero se encuentra en la posición n y se lee un campo `int` y un campo `double`, la posición del puntero después de la lectura será $n+12$.
- Algunos métodos fundamentales son: (todos pueden lanzar excepciones `IOException`)

- ✓ `long getFilePointer()`. Devuelve la posición actual del puntero (en bytes). Ese número de bytes representa el número de bytes desde el inicio del fichero hasta la posición actual.

Por ejemplo, la siguiente llamada al método `getFilePointer` devuelve 0 porque inmediatamente después de abrir el fichero el puntero está situado al inicio del fichero, es decir, en el byte 0:

```
RandomAccessFile raf=new RandomAccessFile("Libros.dat","rw");  
System.out.println("Posición del puntero: " + raf.getFilePointer());
```

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

Métodos de posicionamiento

- ✓ **void seek(long n).** Desplaza el puntero del fichero n bytes, tomando como origen el inicio del fichero. Es el método más utilizado, ya que permite colocarse en una posición concreta del archivo.
- ✓ **long length().** Devuelve el tamaño del fichero en bytes. Si queremos situar el puntero al final del fichero *Libros.dat* pondremos:

```
raf.seek(raf.length());
```

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

Lectura/Escritura

- Los métodos para leer y escribir en un fichero directo son los mismos que los de las clases `DataInputStream` y `DataOutputStream` vistos con los ficheros secuenciales:
 - ✓ `readBoolean`, `readByte`, `readChar`, `readInt`, `readDouble`, `readFloat`, `readUTF`, `readLine`.
Métodos de lectura. Leen un dato del tipo indicado. En el caso de `readUTF` lee una cadena en formato Unicode.
 - ✓ `writeBoolean`, `writeByte`, `writeBytes`, `writeChar`, `writeChars`, `writeInt`, `writeDouble`, `writeFloat`, `writeUTF`, `writeLine`. Métodos de escritura. Todos reciben como parámetro el dato a escribir.
- Como vemos `RandomAccessFile` soporta la lectura y escritura de todos los tipos de datos primitivos.
- Cada operación de lectura o escritura avanza el puntero del fichero tantas posiciones como el número de bytes leídos o escritos.
- Además, `RandomAccessFile` tiene su espacio de almacenamiento intermedio, así que no hay que añadirle un buffer adicional.
- Cuando ya no se necesite trabajar con el fichero, éste debe ser cerrado con el método `close()`.

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

- **Ejemplo:** Programa que crea el fichero directo *Personas.dat* con información relativa a personas cuyos datos personales son introducidos por teclado, de acuerdo al siguiente formato de registro:

Campo	Tipo (tamaño máximo)
Nombre	Cadena (20)
Apellidos	Cadena(40)
Edad	Entero
Estado civil	Carácter (S: soltero, C: casado, V: viudo)

- El programa a través de un menú permitirá realizar lo siguiente:
 - ✓ Grabar una persona en el fichero.
 - ✓ Mostrar un listado de todas las personas almacenadas en el fichero.
- La forma con la que se van a identificar las diferentes personas, es a través de su posición en el fichero.

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

- El formato de registro lo modelamos a través de la clase *Registro*:

```
public class Registro {  
  
    private String nombre;        //20 caracteres  
    private String apellidos;     //40 caracteres  
    private int edad;  
    private char estadoCivil;     //S:soltero C:casado V:viudo D:divorciado  
    public static int TAMR = 42 + 82 + 4 + 2;  
  
    public Registro(String nombre, String apellidos,  
                    int edad, char estadoCivil) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.edad = edad;  
        this.estadoCivil = estadoCivil;  
    }  
  
    public String getNombre() { ...3 lines }  
  
    public String getApellidos() { ...3 lines }  
  
    public int getEdad() { ...3 lines }  
  
    public char getEstadoCivil() { ...3 lines }  
  
    public void setNombre(String nombre) { ...3 lines }  
  
    public void setApellidos(String apellidos) { ...3 lines }  
  
    public void setEdad(int edad) { ...3 lines }  
  
    public void setEstadoCivil(char estadoCivil) { ...3 lines }  
}
```

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

- La clase que resuelve el problema sería la siguiente:
 - **IMPORTANTE:** vamos a utilizar el primer entero del fichero para almacenar el numero de registros que tiene. Con lo cual siempre que situemos el puntero del fichero tenemos que tener estos primeros 4 bytes

```
public class EjemploDirecto {  
  
    public static BufferedReader teclado;  
  
    public void mostrarMenu() {  
        System.out.println("1. Nueva persona");  
        System.out.println("2. Listar personas");  
        System.out.println("3. Salir");  
    }  
  
    public int pedirOpcion() throws IOException {  
        System.out.print("Opcion: ");  
        return Integer.parseInt(teclado.readLine());  
    }  
}
```

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

```
public void grabarPersona(RandomAccessFile raf, File f)
    throws IOException {

    int numReg; //contador de registros para situar el puntero
    if (!f.exists()) {
        numReg = 0;
        raf = new RandomAccessFile(f, "rw");
    } else {
        raf = new RandomAccessFile(f, "rw");
        numReg = raf.readInt();
    }

    System.out.println("Fichero con " + numReg + " registros");

    System.out.print("Nombre (ENTER para terminar): ");
    String nom = teclado.readLine();
```

```
while (!nom.equals("")) {
    numReg++;
    System.out.print("Apellidos: ");
    String apell = teclado.readLine();
    System.out.print("Edad: ");
    int edad = Integer.parseInt(teclado.readLine());
    System.out.print("Estado civil (S/C/V/D): ");
    char ec = teclado.readLine().toUpperCase().charAt(0);

    Registro r = new Registro(nom, apell, edad, ec);
    raf.seek(4 + (numReg - 1) * Registro.TAMR);
    raf.writeUTF(r.getNombre());
    raf.writeUTF(r.getApellidos());
    raf.writeInt(r.getEdad());
    raf.writeChar(r.getEstadoCivil());

    System.out.print("Nombre (ENTER para terminar): ");
    nom = teclado.readLine();
}

//actualizamos el numero de registros del fichero
raf.seek(0);
raf.writeInt(numReg);
raf.close();
```

10. PROCESAMIENTO DE FICHEROS DIRECTOS.

```
public void mostrarPersonas(RandomAccessFile raf, File f)
    throws IOException {

    raf = new RandomAccessFile(f, "r");

    System.out.println("\nPersonas almacenadas ");
    boolean EOF = false;
    int reg = 0; //contador de registros para situar el puntero

    try {
        while (!EOF) {
            raf.seek(4 + (reg * Registro.TAMR));
            String nombre = raf.readUTF();
            reg++;
            System.out.println("\nRegistro " + reg);
            System.out.println("Nombre: " + nombre);
            System.out.print("Apellidos: ");
            System.out.println(raf.readUTF());
            System.out.print("Edad: ");
            System.out.println(raf.readInt());
            System.out.print("Estado civil: ");
            System.out.println(raf.readChar());
        }
    } catch (EOFException e) {
        EOF = true;
    } finally {
        raf.close();
    }
}
```

```
public static void main(String[] args) {

    teclado = new BufferedReader(new InputStreamReader(System.in));
    File f = new File(".\\src\\Ficheros\\Personas.dat");
    RandomAccessFile raf = null;
    EjemploDirecto ej=new EjemploDirecto();


    try {
        ej.mostrarMenu();
        int opc;
        do {
            opc = ej.pedirOpcion();
            switch (opc) {
                case 1:
                    ej.grabarPersona(raf, f);
                    break;
                case 2:
                    ej.mostrarPersonas(raf, f);
                    break;
                case 3:
                    break;
                default:
                    System.out.println("Opcion incorrecta");
            }
        } while (opc != 3);

    } catch (java.io.FileNotFoundException e) {
        System.out.println("Error al abrir el fichero");
    } catch (java.io.IOException e) {
        e.printStackTrace();
    }
}
```


11. EL LENGUAJE XML.

- El lenguaje **XML** (*eXtended Markup Language*) ofrece la posibilidad de representar la información de forma neutra, independiente del lenguaje de programación y del sistema operativo empleado.
- Su utilidad en el desarrollo de aplicaciones software son por ejemplo los servicios web.
- Pero XML no ha nacido sólo para su aplicación en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo ...
- Tiene un papel muy importante ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.
- Desde un punto de vista a "bajo nivel", un documento XML no es otra cosa que un fichero de texto. Con lo cual, podemos utilizar las clases vistas anteriormente para acceder y manipular ficheros XML.
- Sin embargo, desde un punto de vista a "alto nivel", un documento XML no es un mero fichero de texto. Su uso en el desarrollo de aplicaciones hace necesarias herramientas específicas (librerías) para acceder y manipular este tipo de archivos de manera potente, flexible y eficiente.

```
<?xml version="1.0"?>
<quiz>
  <qanda seq="1">
    <question>
      Who was the forty-second
      president of the U.S.A.?
    </question>
    <answer>
      William Jefferson Clinton
    </answer>
  </qanda>
  <!-- Note: We need to add
  more questions later.-->
</quiz>
```



11. EL LENGUAJE XML.

- Las herramientas que leen el lenguaje XML y comprueban si el documento es válido sintácticamente se denominan **analizadores sintácticos** o **parsers**.
- Para XML existen un gran número de parsers, pero dos de los modelos más conocidos: son **DOM** y **SAX**.
- Estos parsers tienen implementaciones para la gran mayoría de lenguajes de programación: Java, .NET, etc.
- Una clasificación de las herramientas para el acceso a ficheros XML es la siguiente:
 - ✓ **Herramientas que validan los documentos XML.** Estas comprueban que el documento XML al que se quiere acceder está bien formado (well-formed) según la definición de XML y, además, que es válido con respecto a un esquema XML (XML-Schema).
Un ejemplo de este tipo de herramientas es *JAXB* (*Java Architecture for XML Binding*).
 - ✓ **Herramientas que no validan los documentos XML.** Estas 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. Ejemplos de este tipo de herramientas son *DOM* y *SAX*.

12. DOM.

- La tecnología **DOM** (*Document Object Model*) es una interfaz de programación que permite analizar y manipular dinámicamente y de manera global el contenido, el estilo y la estructura de un documento XML.
- Para trabajar con un documento XML primero se almacena en memoria en forma de árbol con nodos padre, nodos hijo y nodos finales que son aquellos que no tienen descendientes.
- En este modelo todas las estructuras de datos del documento XML se transforman en algún tipo de nodo, y luego esos nodos se organizan jerárquicamente en forma de árbol para representar la estructura descrita por el documento XML.
- Una vez creada en memoria esta estructura, los métodos de DOM permiten recorrer los diferentes nodos del árbol y analizar a qué tipo particular pertenecen. En función del tipo de nodo, la interfaz ofrece una serie de funcionalidades u otras para poder trabajar con la información que contienen.
- A continuación tenemos un documento XML para representar las puntuaciones de un juego:

12. DOM.

```
<?xml version="1.0" encoding="UTF-8"?>
<lista_puntuaciones>
  <puntuacion fecha="1288122023410">
    <nombre>Mi nombre</nombre>
    <puntos>45000</puntos>
  </puntuacion>
  <puntuacion fecha="1288122428132">
    <nombre>Otro nombre</nombre>
    <puntos>31000</puntos>
  </puntuacion>
</lista_puntuaciones>
```

Cada puntuación está definida por un atributo *fecha*, un texto que indica la fecha en la que ha obtenido la puntuación y por dos elementos hijo: *nombre* y *puntos*.

12. DOM.

- Vamos a ver como podemos desde Java abrir el documento XML y crear un árbol DOM para procesarlo.
- Con DOM hay que cargar el documento XML en memoria RAM y manipularlo directamente en memoria.
- Como DOM representa el documento como un árbol, podemos crear nuevos nodos, borrar y modificar los existentes. Entonces una vez dispongamos de la nueva versión, podremos almacenarlo en un fichero o mandarlo por Internet.
- Los programas Java que usan DOM necesitan estas interfaces (entre otras):
 - ✓ **Document**: objeto que representa a un documento XML.
 - ✓ **Element**: representa a cada elemento del documento XML.
 - ✓ **Node**: representa a cualquier nodo del documento XML.
 - ✓ **NodeList**: lista con todos los nodos hijo de un determinado nodo.
 - ✓ **Attr**: permite acceder a los atributos de un nodo.

12. DOM.

- Ejemplo: Almacenar nuevas puntuaciones en el fichero *puntuaciones.xml* anterior.

```
public class PuntuacionesDOM {  
  
    private static String fichero = "..\\src\\ficheros\\puntuaciones.xml";  
    private Document documento; //representa al arbol DOM  
    private boolean cargadoDocumento;  
  
    public PuntuacionesDOM() {  
        cargadoDocumento = false;  
    }  
  
    public void crearXML() {  
        try {  
            DocumentBuilderFactory factory  
                = DocumentBuilderFactory.newInstance();  
            DocumentBuilder builder = factory.newDocumentBuilder();  
            documento = builder.newDocument();  
            Element raiz = documento.createElement("lista_puntuaciones");  
            documento.appendChild(raiz);  
            cargadoDocumento = true;  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

- En `crearXML()` comenzamos construyendo objetos *DocumentBuilderFactory* y *DocumentBuilder* para poder crear una nueva instancia de documento.
- Creamos un nuevo elemento que es añadido en la raíz de documento.

12. DOM.

```
public void leerXML(InputStream entrada) throws Exception {  
    DocumentBuilderFactory fabrica  
        = DocumentBuilderFactory.newInstance();  
    DocumentBuilder constructor = fabrica.newDocumentBuilder();  
    documento = constructor.parse(entrada);  
    cargadoDocumento = true;  
}
```

- En `leerXML()` el proceso es similar, aunque ahora llamamos al método `parse()`, que se encargará de interpretar (parsear) el documento XML (File) y generar el DOM
- Ahora `documento` apunta al árbol DOM listo para ser recorrido.

12. DOM.

- El método `nuevo()` inserta dentro de *documento* una nueva etiqueta `<puntuacion>` con los atributos y etiquetas interiores necesarios.

```
public void nuevo(int puntos, String nombre, long fecha) {  
    Element puntuacion = documento.createElement("puntuacion");  
    puntuacion.setAttribute("fecha", String.valueOf(fecha));  
    Element e_nombre = documento.createElement("nombre");  
    Text texto = documento.createTextNode(nombre);  
    e_nombre.appendChild(texto);  
    puntuacion.appendChild(e_nombre);  
    Element e_puntos = documento.createElement("puntos");  
    texto = documento.createTextNode(String.valueOf(puntos));  
    e_puntos.appendChild(texto);  
    puntuacion.appendChild(e_puntos);  
    Element raiz = documento.getDocumentElement();  
    raiz.appendChild(puntuacion);  
}
```

- Comienza creando el elemento *puntuacion*, al que añade el atributo *fecha*. Luego crea el elemento *nombre* y le añade el texto correspondiente.
- Este elemento es añadido como hijo a *puntuacion*.
- El mismo proceso se repite para el elemento *puntos*.
- Para finalizar, el elemento *puntuacion* es añadido a la raíz del documento.

12. DOM.

- El método `nuevo()` inserta dentro de *documento* una nueva etiqueta `<puntuacion>` con los atributos y etiquetas interiores necesarios.

```
public void nuevo(int puntos, String nombre, long fecha) {  
    Element puntuacion = documento.createElement("puntuacion");  
    puntuacion.setAttribute("fecha", String.valueOf(fecha));  
    Element e_nombre = documento.createElement("nombre");  
    Text texto = documento.createTextNode(nombre);  
    e_nombre.appendChild(texto);  
    puntuacion.appendChild(e_nombre);  
    Element e_puntos = documento.createElement("puntos");  
    texto = documento.createTextNode(String.valueOf(puntos));  
    e_puntos.appendChild(texto);  
    puntuacion.appendChild(e_puntos);  
    Element raiz = documento.getDocumentElement();  
    raiz.appendChild(puntuacion);  
}
```

- Comienza creando el elemento *puntuacion*, al que añade el atributo *fecha*. Luego crea el elemento *nombre* y le añade el texto correspondiente.
- Este elemento es añadido como hijo a *puntuacion*.
- El mismo proceso se repite para el elemento *puntos*.
- Para finalizar, el elemento *puntuacion* es añadido a la raíz del documento.

12. DOM.

```
public ArrayList aArrayListString() {  
    ArrayList result = new ArrayList();  
    String nombre = "", puntos = "";  
    Element raiz = documento.getDocumentElement();  
    NodeList puntuaciones = raiz.getElementsByTagName("puntuacion");  
    for (int i = 0; i < puntuaciones.getLength(); i++) {  
        Node puntuacion = puntuaciones.item(i);  
        NodeList propiedades = puntuacion.getChildNodes();  
        for (int j = 0; j < propiedades.getLength(); j++) {  
            Node propiedad = propiedades.item(j);  
            String etiqueta = propiedad.getNodeName();  
            if (etiqueta.equals("nombre")) {  
                nombre = propiedad.getFirstChild().getNodeValue();  
            } else if (etiqueta.equals("puntos")) {  
                puntos = propiedad.getFirstChild().getNodeValue();  
            }  
        }  
        result.add(nombre + " " + puntos);  
    }  
    return result;  
}
```

- Este método devuelve un ArrayList de cadenas con las puntuaciones almacenadas.
- Para ello iremos recorriendo todo el documento comenzando por el elemento *raiz*.
- Obtenemos la lista de nodos puntuaciones para recorrerla.
- Para cada uno de estos nodos, obtenemos los nodos hijos. Recorremos esta segunda lista donde analizamos si el nombre del nodo es nombre o puntos.
- Antes de pasar al siguiente nodo puntuacion, añadimos lo obtenido al ArrayList.

12. DOM.

- Finalmente, si queremos escribir el documento XML (en memoria) al fichero en disco, utilizamos este método:

```
public void escribirXML(OutputStream salida) throws Exception {  
    TransformerFactory factory = TransformerFactory.newInstance();  
    Transformer transformador = factory.newTransformer();  
    transformador.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");  
    transformador.setOutputProperty(OutputKeys.INDENT, "yes");  
    DOMSource fuente = new DOMSource(documento);  
    StreamResult resultado = new StreamResult(salida);  
    transformador.transform(fuente, resultado);  
}
```

- Este método permite escribir el documento XML utilizando un objeto de la clase *javax.xml.transform.Transformer*.
- Tras configurarlo, se le indica como fuente *documento* y como resultado de la transformación el *OutputStream* pasado como parámetro.

13. SAX.

- SAX igual que DOM permite realizar un *parser* sobre un documento XML para así poder analizar su contenido.
- A diferencia de DOM, SAX no almacena los datos en memoria, por lo tanto, necesitaremos una estructura de datos donde guardar la información contenida en el XML.
- Para realizar el parseo se van generando una serie de eventos a medida que se va leyendo el documento secuencialmente, que provoca la llamada a los siguientes métodos:

startDocument() → encontrada etiqueta de inicio de documento

startElement() → encontrada etiqueta de inicio de elemento

startElement()

characters() → encontrados caracteres entre las etiquetas

endElement()

...

endElement() → encontrada etiqueta de fin de elemento

endDocument() → encontrada etiqueta de fin de documento

- Por ejemplo, al analizar el documento XML anterior, SAX generará los siguientes eventos:

12. SAX.

Comienza elemento: lista_puntuaciones
Comienza elemento: puntuacion, con atributo fecha="1288122023410"
Comienza elemento: nombre
Texto de nodo: Pepe
Finaliza elemento: nombre
Comienza elemento: puntos
Texto de nodo: 45000
Finaliza elemento: puntos
Finaliza elemento: puntuacion
Comienza elemento: puntuacion, con atributo fecha="1288122428132"
Comienza elemento: nombre
Texto de nodo: Yolanda
Finaliza elemento: nombre
Comienza elemento: puntos
Texto de nodo: 31000
Finaliza elemento: puntos
Finaliza elemento: puntuacion
Finaliza elemento: lista_puntuaciones

12. SAX.

- Entonces para analizar un documento mediante SAX, vamos a escribir métodos asociados a cada tipo de evento.
- Este proceso se realiza extendiendo la clase **DefaultHandler** que nos permite reescribir 5 métodos:
 - ✓ **startDocument()**: comienza el documento XML.
 - ✓ **endDocument()**: finaliza documento XML.
 - ✓ **startElement(String uri, String nombreLocal, String nombreCualif, Attributes atributos)**: comienza una nueva etiqueta, entonces se indican los parámetros
 - **uri**: espacio de nombres o vacío, si no se ha definido.
 - **nombreLocal**: nombre local de la etiqueta sin prefijo.
 - **nombreCualif**: nombre cualificado de la etiqueta con prefijo.
 - **atributos**: lista de atributos de la etiqueta.
 - ✓ **endElement(String uri, String nombreLocal, String nombreCualif)**: termina una etiqueta.
 - ✓ **characters(char ch[], int comienzo, int longitud)**: devuelve un array con los caracteres dentro de una etiqueta. Si queremos un String con estos caracteres podemos hacer:

```
String s = new String(ch, comienzo, longitud)
```

12. SAX.

- Ejemplo: Almacenar nuevas puntuaciones en el fichero *puntuaciones.xml*.

```
public class PuntuacionesSAX {  
  
    private static String fichero = ".\\src\\ficheros\\puntuaciones.xml";  
    private ListaPuntuaciones lista;  
    public static boolean cargadaLista;  
  
    public PuntuacionesSAX() {  
        lista = new ListaPuntuaciones();  
        cargadaLista = false;  
    }  
  
    public void guardarPuntuacion(int puntos, String nombre, long fecha) {  
        try {  
            if (!cargadaLista) { //si el programa se ejecuta la primera vez cargamos el documento XML  
                lista.leerXML(new FileInputStream(fichero));  
            }  
        } catch (FileNotFoundException e) {  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        lista.nuevo(puntos, nombre, fecha);  
        try {  
            lista.escribirXML(new FileOutputStream(fichero));  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

- ListaPuntuaciones* es donde guardaremos la información contenida en el fichero XML.
- Esta clase se define a continuación

```
public ArrayList<String> listaPuntuaciones() {  
    try {  
        if (!cargadaLista) {  
            lista.leerXML(new FileInputStream(fichero));  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return lista.aArrayListString();  
}
```

12. SAX.

```
public class ListaPuntuaciones {  
  
    public static List<Puntuacion> listaPuntuaciones;  
  
    public ListaPuntuaciones() {  
        listaPuntuaciones = new ArrayList<Puntuacion>();  
    }  
  
    public void setListaPuntuaciones(List<Puntuacion> listaPuntuaciones) {  
        this.listaPuntuaciones = listaPuntuaciones;  
    }  
  
    public void nuevo(int puntos, String nombre, long fecha) {  
        Puntuacion puntuacion = new Puntuacion(fecha, nombre, puntos);  
        listaPuntuaciones.add(puntuacion);  
    }  
  
    public ArrayList<String> aArrayListString() {  
        ArrayList<String> result = new ArrayList<String>();  
        for (Puntuacion puntuacion : listaPuntuaciones) {  
            result.add(puntuacion.getNombre() + " " + puntuacion.getPuntos());  
        }  
        return result;  
    }  
}
```

- El objetivo de esta clase es mantener una lista de objetos *Puntuacion*.
- Dispone de métodos para insertar un nuevo elemento y devolver un listado con todas las puntuaciones almacenadas.
- Lo verdaderamente interesante de esta clase es que permite la lectura de los datos del documento XML, para ello usaremos el métodos *leerXML()*.

12. SAX.

```
public void leerXML(InputStream entrada) throws Exception {  
    SAXParserFactory factory = SAXParserFactory.newInstance();  
    SAXParser parser = factory.newSAXParser();  
    XMLReader lector = parser.getXMLReader();  
    ManejadorXML manejadorXML = new ManejadorXML();  
    lector.setContentHandler(manejadorXML);  
    lector.parse(new InputSource(entrada));  
    PuntuacionesSAX.cargadaLista = true;  
}
```

- Para leer el documento XML comenzamos creando una instancia de la clase *SAXParserFactory*, lo que nos permite crear el parser XML de tipo *SAXParser*.
Luego creamos un lector, de la clase *XMLReader*, asociado a este parser.
Creamos la clase *ManejadorXML* y asociamos este manejador al *XMLReader*.
- Para finalizar le indicamos al *XMLReader* que entrada tiene para que realice el proceso de parser.
- La clase *ManejadorXML* va a depender del formato del fichero que queramos leer. La vemos a continuación:

12. SAX.

```
public class ManejadorXML extends DefaultHandler {  
  
    private StringBuilder cadena;  
    private Puntuacion puntuacion;  
  
    @Override  
    public void startDocument() throws SAXException {  
        ListaPuntuaciones.listaPuntuaciones = new ArrayList<Puntuacion>();  
        cadena = new StringBuilder();  
    }  
  
    @Override  
    public void startElement(String uri, String nombreLocal,  
                             String nombreCualif, Attributes atr) throws SAXException {  
        cadena.setLength(0);  
        if (nombreCualif.equals("puntuacion")) {  
            puntuacion = new Puntuacion();  
            puntuacion.setFecha(Long.parseLong(atr.getValue("fecha")));  
        }  
    }  
}
```

- Esta clase define un manejador que captura los cinco eventos generados en el proceso de parsing.
- En *startDocument()* nos limitamos a inicializar variables.
- En *startElement()* verificamos que hemos llegado a una etiqueta <puntuación>.

12. SAX.

```
@Override
public void characters(char ch[], int comienzo, int lon) {
    cadena.append(ch, comienzo, lon);
}

@Override
public void endElement(String uri, String nombreLocal,
    String nombreCualif) throws SAXException {
    if (nombreCualif.equals("puntos")) {
        puntuacion.setPuntos(Integer.parseInt(cadena.toString()));
    } else if (nombreCualif.equals("nombre")) {
        puntuacion.setNombre(cadena.toString());
    } else if (nombreCualif.equals("puntuacion")) {
        ListaPuntuaciones.listaPuntuaciones.add(puntuacion);
    }
}

@Override
public void endDocument() throws SAXException {
}
```

El método *characters()* se llama cuando aparece texto dentro de una etiqueta (<etiqueta> caracteres </etiqueta>).

El método *endElement()* resulta más complejo, dado que en función de que etiqueta esté acabando realizaremos una tarea diferente. Si se trata de </puntos> o de </nombre> utilizaremos el valor de la variable *cadena* para actualizar el valor correspondiente. Si se trata de </puntuacion> añadimos el objeto *puntuacion* a la lista.

12. SAX.

- Finalmente presentamos la clase Puntuacion que modela cada uno de las puntuaciones del fichero XML.

```
public class Puntuacion {  
  
    private long fecha;  
    private String nombre;  
    private int puntos;  
  
    public Puntuacion(long fecha, String nombre, int puntos) {  
        this.fecha = fecha;  
        this.nombre = nombre;  
        this.puntos = puntos;  
    }  
  
    public Puntuacion() { ...2 lines }  
  
    public long getFecha() { ...3 lines }  
  
    public String getNombre() { ...3 lines }  
  
    public int getPuntos() { ...3 lines }  
  
    public void setFecha(long fecha) { ...3 lines }  
  
    public void setNombre(String nombre) { ...3 lines }  
  
    public void setPuntos(int puntos) { ...3 lines }  
  
    @Override  
    public String toString() { ...3 lines }
```

NOTA: Incorpora la librería *apache-xml-xerces.jar* para que el ejemplo funcione.

13. JSON.

- Finalmente vamos a hablar de una de las notaciones para representar información más populares en los últimos años, y que está quitando sitio a XML.
- Se trata de JSON, cuyas siglas significan *JavaScript Object Notation*.
- Se trata de un formato para representar información similar a XML que presenta dos ventajas frente a este:
 - ✓ Es más compacto, al necesitar menos bytes para codificar la información.
 - ✓ El código necesario para realizar un parser es mucho menor.
- Estas ventajas hacen que cada vez sea más popular, especialmente en el intercambio de datos a través de la red.
- A continuación, vamos a comparar la codificación JSON de nuestro ejemplo de puntuaciones con el fichero XML:



13. JSON.

```
<lista_puntuaciones>
  <puntuacion fecha="1288122023410">
    <nombre>Pepe</nombre>
    <puntos>45000</puntos>
  </puntuacion>
  <puntuacion fecha="1288122428132">
    <nombre>Yoli</nombre>
    <puntos>31000</puntos>
  </puntuacion>
</lista_puntuaciones>
```

```
{
  "puntuaciones": [
    { "fecha": 1288122023410, "nombre": "Pepe", "puntos": 45000 },
    { "fecha": 1288122428132, "nombre": "Yoli", "puntos": 31000 }
  ]
}
```

- Vemos que se reduce casi en un 50% el número de caracteres empleados.
- Desde Java podemos utilizar una librería desarrollada por Google para procesar ficheros JSON. Esta librería es Gson.



13. JSON.

GSON

- GSON es una librería de código abierto creada por Google que permite serializar objetos Java para convertirlos en un String.
- Su uso más frecuente es convertir un objeto en su representación JSON y viceversa.
- La gran ventaja de esta librería es que puede ser usada sobre objetos de cualquier tipo de clases, incluso clases preexistentes que no hayamos creado nosotros.
- Esto es posible al no ser necesario introducir código en las clases para que sean serializadas.
- Empezaremos viendo como se convierte un objeto Java a JSON, para luego realizar el proceso contrario, convertir un JSON a objetos Java.

13. JSON.

Java → JSON (GSON)

- Vamos a convertir un objeto *Amigo* a JSON. Para ello crearemos un objeto Gson que se encargará de realizar la conversión.
- La clase *Amigo* debe tener todos los getter y setter:

```
public class Amigo {  
  
    private String nombre;  
    private Integer edad;  
    private String tf;  
    private String mail;  
  
    public Amigo(String nombre, Integer edad,  
                 String tf, String mail) {  
        this.nombre = nombre;  
        this.edad = edad;  
        this.tf = tf;  
        this.mail = mail;  
    }  
}
```

```
public String getNombre() { ...3 lines }  
  
public Integer getEdad() { ...3 lines }  
  
public String getTf() { ...3 lines }  
  
public String getMail() { ...3 lines }  
  
public void setNombre(String nombre) { ...3 lines }  
  
public void setEdad(Integer edad) { ...3 lines }  
  
public void setTf(String tf) { ...3 lines }  
  
public void setMail(String mail) { ...3 lines }
```


13. JSON.

Java → JSON (GSON)

- Ahora lo único que tenemos que hacer es llamar al método *toJson()* pasándolo el objeto Amigo:

```
String JSON=""; //Almacena amigos en formato JSON
Gson gson = new Gson();
JSON+=gson.toJson(new Amigo("Pepe",25,"600111222","pepe@gmail.com"));
JSON+=gson.toJson(new Amigo("Maria",20,"983333444","maria@gmail.com"));
System.out.println(JSON);
```

- El resultado será el siguiente:

```
{"nombre":"Pepe","edad":25,"tf":"600111222","mail":"pepe@gmail.com"}
{"nombre":"Maria","edad":20,"tf":"983333444","mail":"maria@gmail.com"}
```

13. JSON.

Java → JSON (GSON)

- Si por ejemplo queremos convertir a JSON una lista de amigos, haríamos lo siguiente:

```
public class Agenda {  
  
    private ArrayList<Amigo> contactos;  
  
    public ArrayList<Amigo> getContactos() {  
        return contactos;  
    }  
  
    public void setContactos(ArrayList<Amigo> contactos) {  
        this.contactos = contactos;  
    }  
  
}
```

```
Amigo amigol = new Amigo("amigol", 18, "900555666", "amigol@gmail.com");  
Amigo amigo2 = new Amigo("amigo2", 30, "800777888", "amigo2@gmail.com");  
Amigo amigo3 = new Amigo("amigo3", 22, "700333444", "amigo3@gmail.com");  
ArrayList<Amigo> amigos=new ArrayList<>();  
amigos.add(amigol);  
amigos.add(amigo2);  
amigos.add(amigo3);  
  
Agenda ag=new Agenda();  
ag.setContactos(amigos);  
  
JSON=gson.toJson(ag);  
System.out.println(JSON);
```

13. JSON.

Java → JSON (GSON)

- Ahora el resultado es el siguiente:

```
{"contactos":
```

```
  [ {"nombre":"amigo1","edad":18,"tf":"900555666","mail":"amigo1@gmail.com"},
```

```
    {"nombre":"amigo2","edad":30,"tf":"800777888","mail":"amigo2@gmail.com"},
```

```
    {"nombre":"amigo3","edad":22,"tf":"700333444","mail":"amigo3@gmail.com"}]
```

```
]
```

```
}
```

13. JSON.

Java ← JSON (GSON)

- El método usado para realizar la conversión es *fromJson* al que tenemos que pasar el String con el JSON y la clase del objeto destino.
- Por ejemplo vamos convertir en objeto *Agenda* la siguiente cadena:

```
"{'contactos':[{'nombre':'amigo4','edad':28,'tf':'900666777','mail':'amigo4@gmail.com'},  
{'nombre':'amigo5','edad':35,'tf':'800999999','mail':'amigo5@gmail.com'}]}'"
```

- Haremos lo siguiente:

```
String JSONtoAgenda = "{'contactos':[{'nombre':'amigo4','edad':28,'tf':'900666777','mail':'amigo4@gmail.com'},"  
+ "{'nombre':'amigo5','edad':35,'tf':'800999999','mail':'amigo5@gmail.com'}]}'";  
ag = gson.fromJson(JSONtoAgenda, Agenda.class);
```

- Como las comillas dobles tienen un significado especial en Java hay que reemplazarlas por comillas simples para evitar problemas de conversión.

13. JSON.

- Es muy habitual que tengamos que convertir un ArrayList a JSON o un JSON en ArrayList. El primer caso es sencillo de resolver usando el método *toJson()* al que le pasamos el ArrayList como argumento.
- Sin embargo, para el caso contrario usando el método *fromJson()* como hemos visto anteriormente hay pasar el tipo de objeto al que vamos a convertir, como segundo parámetro. Si ponemos ArrayList o ArrayList<Tipo> el compilador nos da un error.

Lo que tenemos que hacer es crear un objeto `java.lang.reflect.Type`, que nos permite representar cualquier tipo que el lenguaje soporte, en nuestro caso un ArrayList de un tipo en específico.

- Ejemplo:

```
//ArrayList de amigos
ArrayList<Amigo> clase=new ArrayList<>();
clase.add(new Amigo("Juan",18,null,"juan@hotmail.com"));
clase.add(new Amigo("Pedro",19,null,"pedro@hotmail.com"));
clase.add(new Amigo("Maria",20,null,"maria@hotmail.com"));

//convertimos a JSON el ArrayList
String amigosJSON=gson.toJson(clase);
System.out.println(amigosJSON); //mostramos el JSON

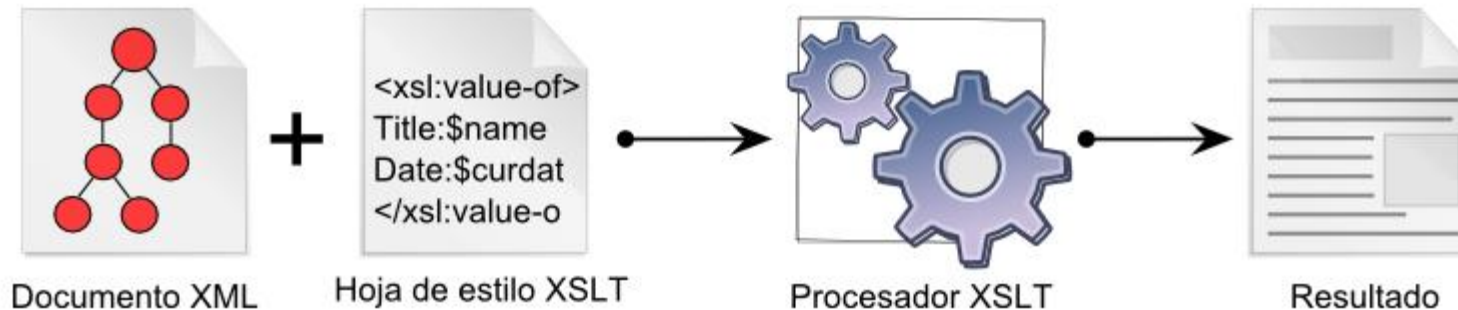
//proceso contrario: convertir el JSON a ArrayList
//creamos el tipo que represente el ArrayList de obj. Amigo
Type tipo = new TypeToken<ArrayList<Amigo>>(){}.getType();
ArrayList<Amigo> pandilla=gson.fromJson(amigosJSON,tipo);
System.out.println(pandilla); //mostramos el contenido del ArrayList
```

14. TRANSFORMACIÓN DE DOCUMENTOS XML

- XML se presenta como un estándar para transmitir datos a través de Internet. Ante la posibilidad de que distintas aplicaciones utilicen esquemas o DTD diferentes, es necesario un sistema que permita transformar los datos de un documento XML.
- **XSLT** (*eXtensible Stylesheet Language – Transformations*), describe un lenguaje basado en XML para transformar documentos XML a cualquier otro formato.
- Normalmente, utilizaremos XSLT para transformar documentos entre esquemas XML que permitan su procesamiento por distintos sistemas.
- También utilizaremos XSLT para transformar documentos XML en HTML, o cualquier otro formato que facilite su presentación en la pantalla o en impresora.
- No debemos confundir las transformaciones XSLT con la presentación de documentos XML con CSS. Con XSLT, generaremos un documento HTML a partir de un documento XML. Se tratará de dos documentos distintos.
- Con CSS, el navegador recibe un documento XML que formatea utilizando las reglas CSS para presentarlo en pantalla de forma que sea más fácilmente legible, pero es el mismo documento.

14. TRANSFORMACIÓN DE DOCUMENTOS XML

- XSLT es parte de la especificación XSL (*eXtensible Stylesheet Language*). En XSL se distingue entre:
 - ✓ XSL FO (*eXtensible Stylesheet Language Formatting Objects*)
 - ✓ XSLT (*eXtensible StyleSheet Language Transformations*)
- XSL FO cuenta con escaso soporte por parte de la industria debido a su complejidad. Su propósito es definir la forma en la que se debe presentar un documento XML en papel o en pantalla. En este sentido, XSL FO sería una especificación similar a CSS.
- Una hoja de estilo XSLT es un documento XML bien formado con extensión .xsl. Veamos un ejemplo de como utilizando Java, a partir de un documento XML y una hoja de estilo XSLT, generamos



14. TRANSFORMACIÓN DE DOCUMENTOS XML

ciudades.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ciudades>
  <ciudad>
    <nombre>Madrid</nombre>
    <habitantes>3500000</habitantes>
  </ciudad>
  <ciudad>
    <nombre>Valladolid</nombre>
    <habitantes>800000</habitantes>
  </ciudad>
  <ciudad>
    <nombre>Toledo</nombre>
    <habitantes>50000</habitantes>
  </ciudad>
</ciudades>
```

ciudades_plantilla.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <xsl:apply-templates />
  </xsl:template>
  <xsl:template match='ciudades'>
    <head>
      <title>LISTA DE CIUDADES</title>
    </head>
    <body>
      <h1>LISTA DE CIUDADES</h1>
      <table border='1'>
        <tr>
          <th>Nombre</th>
          <th>Poblacion</th>
        </tr>
        <xsl:apply-templates select="ciudad"/>
      </table>
    </body>
  </xsl:template>
  <xsl:template match='ciudad'>
    <tr>
      <xsl:apply-templates />
    </tr>
  </xsl:template>
  <xsl:template match='nombre|habitantes'>
    <td>
      <xsl:apply-templates />
    </td>
  </xsl:template>
</xsl:stylesheet>
```

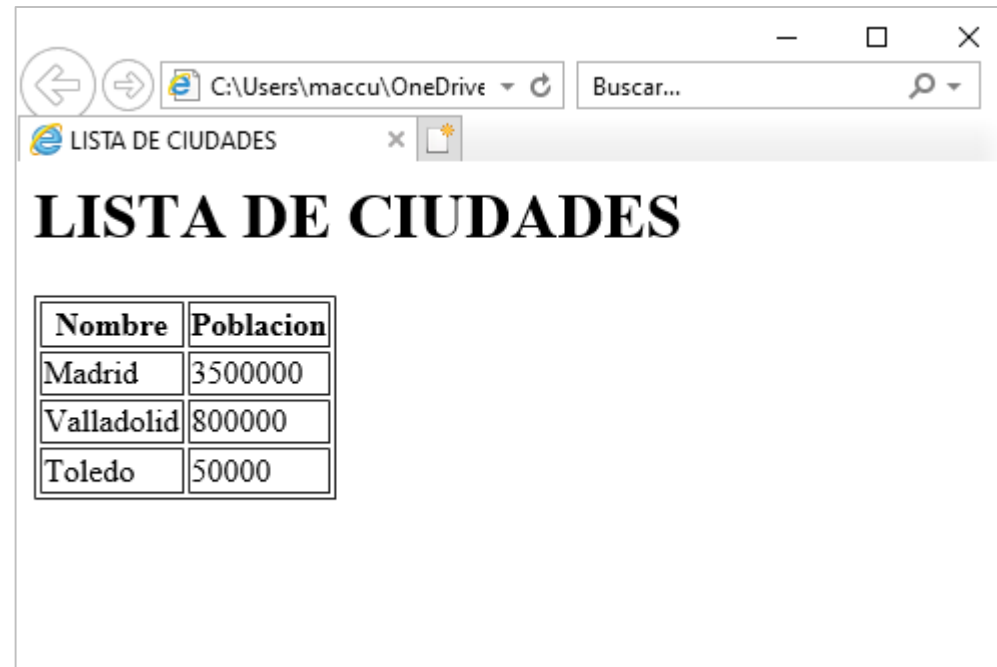

14. TRANSFORMACIÓN DE DOCUMENTOS XML

- El código Java necesario para realizar la transformación sería el siguiente:

```
OutputStream html = null;
try {
    TransformerFactory transformer = TransformerFactory.newInstance();
    Source xslDoc = new StreamSource("./src/ficheros/ciudades_plantilla.xsl");
    Source xmlDoc = new StreamSource("./src/ficheros/ciudades.xml");
    String salida = "./src/ficheros/resultado.html";
    html = new FileOutputStream(salida);
    Transformer transform = transformer.newTransformer(xslDoc);
    transform.transform(xmlDoc, new StreamResult(html));
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (TransformerConfigurationException ex) {
    Logger.getLogger(EjemploXSLT.class.getName()).log(Level.SEVERE, null, ex);
} catch (TransformerException ex) {
    Logger.getLogger(EjemploXSLT.class.getName()).log(Level.SEVERE, null, ex);
} finally {
    try {
        html.close();
    } catch (IOException ex) {
        Logger.getLogger(EjemploXSLT.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

14. TRANSFORMACIÓN DE DOCUMENTOS XML

- Observando el código, utilizamos un objeto `Transformer` y su método `newTransformer` al que le pasamos el archivo XSL que vamos a utilizar para transformar el archivo XML.
- Posteriormente la transformación se consigue con el método `transform`, que nos da como resultado el archivo HTML. Si lo visualizamos en el explorador obtenemos lo siguiente:



15. EXCEPCIONES EN JAVA

- **Excepción:** situación anómala que aparece cuando ejecutamos un programa.
- Algunas de estas situaciones “anómalas” pueden ser: invocar a un método desde un objeto *null*, dividir por cero, abrir un fichero inexistente, etc.
- Nosotros como programadores tenemos la responsabilidad de gestionar las excepciones de nuestros programas de una forma adecuada, es lo que se conoce con el nombre de **manejo de excepciones**.
- Es importante entender que las excepciones no corrigen los errores de programación, lo que hacen es:
 - ✓ *Informar* de la situación.
 - ✓ *Actuar* en consecuencia, según nuestro criterio.
- Las excepciones tiene dos características principales:
 - a) **Enriquecen** los métodos: ya que con ellas no solo los métodos van a poder devolver un resultado, sino que también vana poder “lanzar” una excepción.
 - b) **Permiten alterar el flujo** normal de los programas: el código cliente que llame a un método **que lanza excepciones**, debe de estar preparado para las 2 posibilidades de retorno del método (mencionadas en el punto anterior).

15. EXCEPCIONES EN JAVA

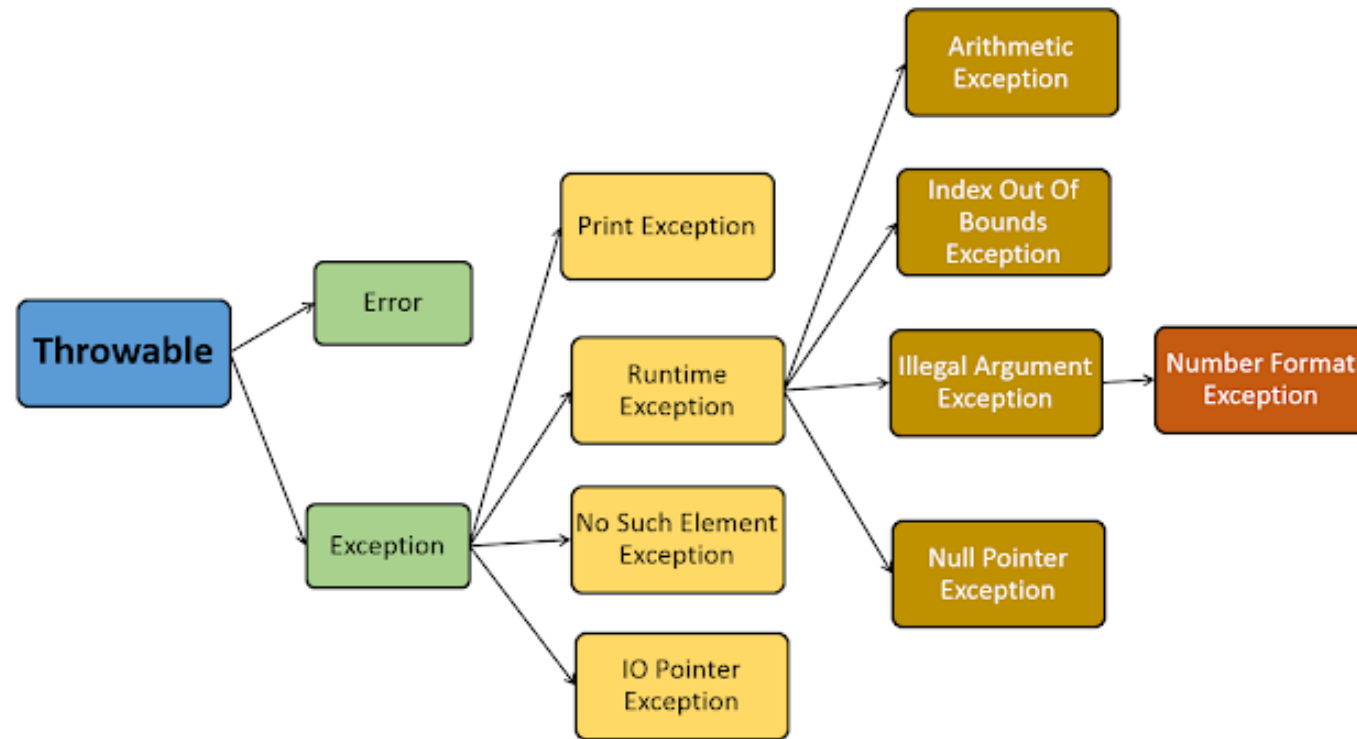
- Un ejemplo típico de excepción es el siguiente:

```
int a = 5;  
int b = 0;  
double c = a / b;  
System.out.println("Resultado=" + c);
```

 Exception in thread "main" java.lang.ArithmeticException: / by zero

- Cuando se intenta realizar la división, como el denominador valor 0, en ese momento se “lanza” la excepción del tipo indicado.
- Es en ese momento cuando se crea un objeto de tipo **ArithmeticException**, que si no es “tratado por nadie”, llega hasta la JVM y muestra por consola el mensaje de error.
- Vamos a ver como podemos manejar esta excepción para evitar precisamente esto.
- En primer lugar tenemos que saber que todas las excepciones en Java heredan de la clase **Exception**, que a su vez hereda de **Throwable**.
- Como podemos observar en la siguiente imagen, sólo hay 2 subclases directas de **Throwable**, una es la clase **Error**, que representa errores poco comunes por lo que no se suelen tratar, y la comentada clase **Exception**, que representa situaciones razonales de capturar en una aplicación.

15. EXCEPCIONES EN JAVA



15. EXCEPCIONES EN JAVA

- En Java las excepciones se manejan a través de los bloques **try-catch**:

```
try {  
    // Código que podría generar excepciones  
} catch(Tipo1 id1) {  
    // Manejo de excepciones de Tipo1  
} catch(Tipo2 id2) {  
    // Manejo de excepciones de Tipo2  
  
...  
} finally {  
    // Código que siempre se va a ejecutar  
}
```
- Dentro del **try** debemos situar el código en el que puede surgir la excepción.
- En el **catch**, dentro de los paréntesis, como si fuera la cabecera de un método, debemos situar el tipo de excepción que queremos capturar seguido de un identificador de la misma. Y dentro del bloque, debemos tratar la excepción por ejemplo mostrando simplemente un mensaje que indique la excepción generada.
- Finalmente en el **finally**, situamos el código que siempre se va a ejecutar, tanto si ocurre excepción como si no. Por ejemplo, se suele utilizar para cerrar conexiones con ficheros o bases de datos, vaciar búferes, etc.

15. EXCEPCIONES EN JAVA

- Si queremos obtener una mayor información sobre la excepción producida, podemos utilizar los siguientes métodos :
 - ✓ **String getMessage()**: devuelve el mensaje descriptivo de la excepción.
 - ✓ **String toString()**: devuelve la descripción de la excepción. Suele indicar la clase de excepción y el texto de getMessage().
 - ✓ **void printStackTrace()**: muestra la información de la pila de llamadas. El resultado es el mismo mensaje que muestra la máquina virtual de Java cuando no se controla la excepción.
- Modificamos el ejemplo anterior manejando la excepción producida:

```
try {  
    int a = 5;  
    int b = 0;  
    double c = a / b;  
    System.out.println("Resultado=" + c);  
} catch (java.lang.ArithmeticException ex) {  
    System.out.println(ex.getMessage());  
    System.out.println(ex.toString());  
    ex.printStackTrace();  
} finally {  
    System.out.println("Fin del ejemplo");  
}
```

15. EXCEPCIONES EN JAVA

- Podemos nosotros mismos lanzar excepciones utilizando la sentencia **throw**:
throw new tipoExcepción(mensaje);
- De esta manera estamos lanzando una excepción del tipo indicado para que "alguien" la maneje. El mensaje es opcional.
- Cuando un método lanza excepciones con **throw**, o cuando Java obliga en código a manejar determinadas excepciones, es obligatorio que el método indique que "propaga" esas excepciones con la sentencia **throws** en la cabecera del método seguida del tipo o tipos de excepción que puede lanzar.
- Ejemplo:

```
public static FileReader abrirFichero(String fichero)
    throws FileNotFoundException {
    FileReader fr = new FileReader(fichero);
    return fr;
}
```

```
public static double dividir(int a, int b) throws Exception {
    if (b == 0) {
        throw new Exception();
    }
    double c = a / b;
    return c;
}
```


15. EXCEPCIONES EN JAVA

- Entonces el código cliente que use este método “sabe” que propaga las excepciones indicadas, con lo cuál tiene dos opciones:
 - a) Manejarla mediante un bloque try-catch.
 - b) Propagarla nuevamente al método superior (llamante). Esto es muy común cuando se desarrollan librerías de clases para que sean utilizadas por otros programadores).

```
public static void main(String[] args) {  
    try {  
        abrirFichero("./src/ficheros/frases.txt");  
        System.out.println(dividir(3,5));  
    } catch (FileNotFoundException ex) {  
        ex.printStackTrace();  
    } catch (Exception ex) {  
        ex.printStackTrace();  
    }  
}
```