U.T. 1: ACCESO A FICHEROS

1. TIPOS DE FICHEROS

Según su contenido distinguimos entre dos tipos de ficheros:

- **Ficheros de texto** cuando el contenido del fichero contenga exclusivamente caracteres de texto (podemos leerlo con un simple editor de texto)

Extensión	Tipo de fichero
.txt	Fichero de texto plano
.xml	Fichero XML
.json	Fichero en formato de notación de objetos de JavaScript JSON

.props	Fichero de propiedades
.conf	Fichero de configuración
.sql	Script SQL

- **Ficheros binarios** cuando no incluyan sólo texto. Pueden contener imágenes u otros contenidos multimedia, así como valores binarios, fundamentalmente numéricos.

Extensión	Tipo de fichero
.pdf	Fichero PDF
.jpg	Fichero de imagen
.odt, .doc	Fichero generado por un procesador de textos
.avi	Fichero de video
.ppt, .pptx	Fichero de PowerPoint

En el caso de ficheros binarios, podremos encontrarnos con extensiones como *.bin* o *.dat* para hacer referencia a ficheros que contienen información binaria en un formato no estándar. Serán simplemente ficheros que una determinada aplicación sea capaz de leer/escribir de una forma específica para dicha aplicación.

1.1. Propiedades del Sistema

Para el acceso a ficheros de configuración del sistema, conviene conocer el funcionamiento de las System Properties de las que dispone Java en su API. Éntre otras, podemos encontrarnos con algunas propiedades muy interesantes relacionadas con este tema:

- `"file.separator"` Obtiene el caracter, según el S.O., para la separación de las rutas (`/` ó `\`). También se puede utilizar la constante `File.separator`
- `"user.home"` Obtiene la ruta de la carpeta personal del usuario (que dependerá también del S.O.)
- `"user.dir"` Obtiene la ruta del directorio en el que se encuentra actualmente el usuario
- `"line.separator"` Obtiene el caracter que separa las líneas de un fichero de texto (difiere entre Windows/Linux)

En el caso de que se quiera acceder al valor de alguna de estas propiedades debe hacerse utilizando la llamada al método `System.getProperty(String)`

```
System.out.println("La carpeta de mi usuario es " + System.getProperty("user.home"));
```

2. FICHEROS DE TEXTO

En esta parte nos vamos a centrar en 3 tipos (aunque podrían ser más):

- **Ficheros de texto plano** que contendrán texto *libre* y donde podremos escribir sin respetar ningún tipo de formato concreto
- **Ficheros de configuración** que contendrán información de configuración para una aplicación. Tienen un formato específico y Java, además, proporciona una API para trabajar más cómodamente con ellos
- **Ficheros XML** que contienen información y acompañada de etiquetas que le dan significado. Tienen unas reglas y formato más o menos definido y Java también proporciona una API para trabajar con ellos

Otros tipos de ficheros de texto también muy extendidos podrían ser *.json* y *.csv* pero no serán estudiados en esta parte.

2.1. Ficheros de texto plano

Los ficheros de texto son aquellos que únicamente contienen texto, por lo que pueden ser editados directamente con cualquier editor de texto plano (Gedit, vim, . . .). Normalmente se almacenan con la extensión *.txt* pero también podríamos utilizar otras extensiones al nombre, como los scripts SQL (*.sql*), ficheros de código como Java (*.java*) u otros lenguajes, ficheros de configuración (*.ini*, *.props*, *.conf*, . . .), . . .

También se incluyen en la categoría de ficheros de texto los que además incluyen información adicional (siempre en forma de texto) que permiten interpretar los datos del fichero de una manera u otra, añadiendo más información al mismo. Estos formatos son HTML, XML, JSON, . . .

En cualquier caso, desde Java siempre se podrán leer/escribir de la misma manera, según veremos a continuación. También veremos como pueden leerse/escribirse utilizando librerías aquellos ficheros de texto plano que contienen formato, como hemos comentado anteriormente, haciendo de esa forma mucho más fácil el trabajo.

Escribir y leer en ficheros de texto plano

Veamos un ejemplo haciendo uso de alguna de las clases del paquete *java.nio*.

```
FileSystem sistemaFicheros = FileSystems.getDefault();
// también puedo obtener el objeto Path a partir de Paths.get(String ruta),
// pero a partir de la version 11 se aconseja el método estático Path.of(String ruta)
Path rutaFichero = sistemaFicheros.getPath("textos.txt");
String text = "Esto es una cadena de prueba\n";
// Escribimos con el método estático, de la clase Files, writeString
// Requiere 3 parámetros: el objeto Path, el String y las opciones
Files.writeString(rutaFichero, text/*.getBytes(StandardCharsets.UTF_8)*/,
StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);

text = "Esto es una segunda cadena de prueba\n";
Files.writeString(rutaFichero, text/*.getBytes(StandardCharsets.UTF_8)*/,StandardOpenOption.APPEND);

// Podemos leer todas las líneas en un ArrayList con el método estático readAllLines de la clase Files
// Requiere 2 parámetros: el objeto Path y la codificación, o uno sólo y la codificación será UTF_8
List<String> lines = Files.readAllLines(Paths.get("textos.txt"),StandardCharsets.UTF_8);
for (String line : lines)
 System.out.println(line);
```

El código anterior hace uso de las clases *FileSystem* y *FileSystems* para obtener el objeto *Path*, que viene a ser la equivalencia a *java.io.File*. A continuación crea el fichero y escribe en él, haciendo uso de *Files* con distintas opciones indicadas por la enumeración *StandardOpenOption*. Finalmente el programa muestra todo el contenido del fichero, haciendo uso igualmente de *Files* y *Paths*.

2.2. Ficheros de configuración

En la API de Java se incluyen librerías para trabajar con los ficheros de configuración de aplicaciones (por ejemplo, servicios de red). Puesto que todos siguen un mismo patrón (líneas de comentario comenzando por el caracter almohadilla y resto de líneas con sintaxis "directiva=valor"), es la librería la que se encarga de acceder al fichero a bajo nivel y el programador sólo tiene que indicar a qué propiedad quiere acceder, sin haber de añadir nada de código para leer o escribir el fichero tal y como hemos visto en el punto anterior.

Un ejemplo de fichero de configuración de esta librería de java sería el fichero que sigue:

```
# Fichero de configuracion
# Thu Nov 14 10:49:39 CET 2013

user=user1
password=pass1
```

```
server=miservidor
port=9999
```

Escribir ficheros de configuración

Así, si queremos generar, desde Java, un fichero de configuración como el anterior utilizamos la clase *java.util.Properties*, la cual hereda de *Hashtable*:

```
Properties configuracion = new Properties();
configuracion.setProperty("user", "user1");
configuracion.setProperty("password", "pass1");
configuracion.setProperty("server", "miservidor");
configuracion.setProperty("port", "9999");
try {
  configuracion.store(new FileOutputStream("configuracion.props"),
                                           "Fichero de configuracion");
} catch (FileNotFoundException fnfe ) {
  fnfe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Leer ficheros de configuración

A la hora de leerlo, en lugar de recorrer todo el fichero como suele ocurrir con los ficheros de texto, sólo habremos de cargarlo e indicar de qué propiedad queremos obtener su valor (`getProperty(String)`).

```
Properties configuracion = new Properties();
try {
  configuracion.load(new FileInputStream("configuracion.props"));
  usuario = configuracion.getProperty("user");
  password = configuracion.getProperty("password");
  servidor = configuracion.getProperty("server");
  puerto = Integer.valueOf(configuration.getProperty("port"));
} catch (FileNotFoundException fnfe ) {
  fnfe.printStackTrace();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Para ambos casos, escritura y lectura, hay que tener en cuenta que, al tratarse de ficheros de texto, toda la información se almacena como si de un `String` se tratara. Por tanto, todos aquellos tipos `Date`, `boolean` o incluso cualquier tipo numérico serán almacenados en formato texto. Así, habrá que tener en cuenta las siguientes consideraciones:

- Para fechas, deberán ser convertidas a texto cuando se quieran escribir y nuevamente reconvertidas a `Date` cuando se lea el fichero y queramos trabajar con ellas
- Para datos de tipo `boolean`, podemos usar el método `String.valueOf(boolean)` para pasarlos a `String` cuando queramos escribirlos. En caso de que queramos leer el fichero y pasar el valor a tipo `boolean` podremos usar el método `Boolean.parseBoolean(String)`
- Para los tipos numéricos (`integer`, `float`, `double`) es muy sencillo, ya que Java los convertirá a `String` cuando sea necesario al escribir el fichero. En el caso de que queramos leerlo y convertirlos a su tipo concreto, podremos usar los métodos `Integer.parseInt(String)`, `Float.parseFloat(String)` y `Double.parseDouble()`, según proceda.

3. FICHEROS BINARIOS

```
public class Producto implements Serializable {
  private static final long serialVersionUID = 1L;
  private String nombre;
  private float precio;

  public Producto(String nombre, float precio) {
    this.nombre = nombre;
    this.precio = precio;
  }

  public String getNombre() { return nombre; }
  public void setNombre(String nombre) { this.nombre = nombre; }

  public float getPrecio() { return precio ; }
  public void setPrecio(float precio) { this.precio = precio; }
}
```

3.1. Serialización de objetos

Serializar es el proceso por el cual un objeto en memoria pasa a transformarse en una estructura que pueda ser almacenada en un fichero (persistencia). Al proceso contrario le llamaremos *deserializar*.

Hay que tener en cuenta que durante el proceso de serialización, cada objeto se serializa a un fichero, por lo que si queremos almacenar todos los objetos de una aplicación, la idea más conveniente es tener todos éstos en alguna estructura compleja (y por comodidad dinámica) de forma que sea esta estructura la que serialicemos o deserialicemos para guardar o cargar los objetos de una aplicación. Estructuras como `ArrayList`, `Set` ó `HashMap` son clases muy utilizadas para trabajar de esta manera.

Serializar un objeto

```
. . .

try ( FileOutputStream fos = new FileOutputStream("archivo.dat");
      ObjectOutputStream oos = new ObjectOutputStream(fos); )
{
  // listaProductos sería un ArrayList, o similar, creado anteriormente
  oos.writeObject(listaProductos);
} catch (IOException ioe) {
  . . .
}
```

Deserializar un objeto

```
. . .
     List<Producto> listaProductos = null;
     try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("archivo.dat"))) {
         listaProductos = (ArrayList<Producto>) ois.readObject();
     } catch (FileNotFoundException fnfe) {
         fnfe.printStackTrace();
     } catch (ClassNotFoundException cnfe) {
         cnfe.printStackTrace();
     } catch (IOException ioe) {
         ioe.printStackTrace();
     }
. . .
```

4. FICHEROS DE ACCESO ALEATORIO

Con la clase *RandomAccessFile* podemos leer y escribir archivos de manera aleatoria, es decir, que nos podemos situar en una posición concreta, en lugar de tener que recorrerlo desde el principio. Con esta clase, podemos escribir y leer con cada tipo de variable como Integer, String, Double, etc.

Se escribe de forma binaria en el archivo, por lo que no será legible para nosotros al abrir el fichero. Es importante saber que cada tipo de variable ocupa un determinado número de bytes:

```
long: 8 bytes
int: 4 bytes
short: 2 bytes
byte: 1 byte
double: 8 bytes
float: 4 bytes
boolean: 1 byte
char: 2 bytes
String: 2 bytes por cada caracter.
```

Supongamos ahora la siguiente implementación para la clase Producto.

```
public class Producto {
    private int id;
    private String nombre;
    private double precio;
    private boolean descuento;
    private char tipo;
    public Producto(int id, String nombre, double precio, boolean descuento, char tipo)
    {
        this.id = id;
        this.nombre = nombre;
        this.precio = precio;
        this.descuento = descuento;
        this.tipo = tipo;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNombre() {
```

```
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public double getPrecio() {
        return precio;
    }
    public void setPrecio(double precio) {
        this.precio = precio;
    }
    public boolean isDescuento() {
        return descuento;
    }
    public void setDescuento(boolean descuento) {
        this.descuento = descuento;
    }
    public char getTipo() {
        return tipo;
    }
    public void setTipo(char tipo) {
        this.tipo = tipo;
    }
    @Override
    public String toString() {
        return "Producto{" + "id=" + id + ", nombre=" + nombre + ", precio=" + \
         precio + ", descuento=" + descuento + ", tipo=" + tipo + '}';
    }
}
```

Lo primero es crear los productos que vamos a insertar, vamos a guardarlos, por ejemplo, en un ArrayList.

```
ArrayList<Producto> productos = new ArrayList<>();
productos.add(new Producto(1, "Producto 1", 10.5, true, 'T'));
productos.add(new Producto(2, "Producto 2", 15.1, true, 'R'));
productos.add(new Producto(3, "Producto 3", 11.5, false, 'T'));
productos.add(new Producto(4, "Producto 4", 50, false, 'D'));
productos.add(new Producto(5, "Producto 5", 79.3, true, 'U'));
```

Vamos a crear el objeto RandomaccessFile.

```
try (RandomAccessFile raf = new RandomAccessFile("ejemplo_raf.dat", "rw")) {
} catch (FileNotFoundException ex) {
    System.out.println(ex.getMessage());
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```

Necesitamos dos parámetros, uno es la ruta del fichero y otro es el modo de apertura, entre los que tenemos:

r: solo lectura

rw: lectura y escritura. Este es el más usado

rws: como rw pero vacía el contenido del archivo y la fecha de modificación del archivo

rwd: como rw pero vacía el contenido del archivo; puede que la fecha de modificación no cambie hasta cerrar el archivo.

Es importante que a la hora de escribir lo hagamos en orden, para cada tipo de variable tenemos un método *writeXXX*.

```
try (RandomAccessFile raf = new RandomAccessFile("ejemplo_raf.dat", "rw")) {
    for (Producto p: productos) {
        raf.writeInt(p.getId());
        String nom = p.getNombre();
            // limito el nombre escrito a 10 caracteres máximo
        raf.writeUTF(nom.substring(0,Math.min(nom.length(),10)));
        raf.writeDouble(p.getPrecio());
        raf.writeBoolean(p.isDescuento());
        raf.writeChar(p.getTipo());
    }
} catch (FileNotFoundException ex) {
    System.out.println(ex.getMessage());
} catch (IOException ex) {
    System.out.println(ex.getMessage());
}
```

En este caso, cada registro sería de 37 bytes (4 (int) + 2 (longitud del string) + (10 * 2) (string) + 8 (double) + 1 (boolean) + 2 (char)).

También es importante ver como añado los String, uso la clase *StringBuffer* y lo limito al tamaño que yo desee, en este caso a 10 caracteres, si tiene menos caracteres, lo rellenará con caracteres nulos '\0' y si tiene más lo cortará. Como he comentado, es importante saber cuánto ocupa cada String para después leer; por eso todos deben tener un tamaño fijo.

Pues ahora vamos a leer el fichero con *RandomAccessFile*, lo primero es posicionarse donde queramos trabajar. Esto lo podemos hacer con el método *seek*, indicando los bytes, desde principio del fichero, donde debamos posicionarnos. Recuerda que debemos leer en el orden de como escribimos.

Por lo que si queremos posicionarnos en el segundo registro, tendremos que usar seek(37), ya que el primero es el byte 0, si quisiéramos el tercer registro sería seek(74, es decir, 37 x 2).

Veamos como se lee:

```
    // Utilizamos 37 pero lo correcto sería definir una constante con ese valor
raf.seek(37);
System.out.println(raf.readInt());
System.out.println(raf.readUTF(););
System.out.println(raf.readDouble());
System.out.println(raf.readBoolean());
System.out.println(raf.readChar());
```

El ejemplo completo:

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.util.ArrayList;
public class EjemploRandomAccessFile {
    public static void main(String[] args) {
        ArrayList<Producto> productos = new ArrayList<>();
        productos.add(new Producto(1, "Producto 1", 10.5, true, 'T'));
        productos.add(new Producto(2, "Producto 2", 15.1, true, 'R'));
        productos.add(new Producto(3, "Producto 3", 11.5, false, 'T'));
        productos.add(new Producto(4, "Producto 4", 50, false, 'D'));
        productos.add(new Producto(5, "Producto 5", 79.3, true, 'U'));
```

```
        try (RandomAccessFile raf = new RandomAccessFile("ejemplo_raf.dat", "rw")) {
            for (Producto p : productos) {
                    // en este caso, escribimos los registros secuencialmente, uno a continuación del otro
                raf.writeInt(p.getId());
                String nom = p.getNombre();
                // limito el nombre escrito a 10 caracteres máximo
                raf.writeUTF(nom.substring(0,Math.min(nom.length(),10)));
                raf.writeDouble(p.getPrecio());
                raf.writeBoolean(p.isDescuento());
                raf.writeChar(p.getTipo());
            }
            // Ahora vamos a leer sólo uno, el segundo. Nos posicionamos donde empieza ese segundo registro,
            // saltamos sólo uno: 37 bytes
            // Utilizamos 37 pero lo correcto sería definir una constante con ese valor
            raf.seek(37);
            System.out.println("El contenido del segundo registro es:");
            System.out.println(raf.readInt());
            System.out.println(raf.readUTF());
            System.out.println(raf.readDouble());
            System.out.println(raf.readBoolean());
            System.out.println(raf.readChar());
        } catch (FileNotFoundException ex) {
            System.out.println(ex.getMessage());
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
}
```

En este caso, hemos hecho una escritura secuencial pero en este tipo de ficheros normalmente nos posicionamos con el método *seek(int bytes)* según alguna fórmula que relacione algún valor de alguno de sus campos, por ejemplo el identificador numérico "id" con su posición de inicio, en bytes, en el fichero.

5. JAVA.NIO

El paquete *java.nio*, o NIO, se introdujo en un primer paso en la versión 1.4 del lenguaje, para ser ampliada con otros subpaquetes posteriormente en la 1.7. A diferencia de *java.io*, o IO convencional, en java NIO se introduce el flujo de datos orientado al uso de búferes y canales para las operaciones de E/S (Entrada/Salida), que proporcionan diversas ventajas y nuevas funcionalidades, así como un mejor rendimiento.

Las abstracciones centrales de las API de NIO son las siguientes:

Buffers, que son contenedores de datos, conjuntos de caracteres y sus decodificadores y codificadores asociados, que traducen entre bytes y caracteres Unicode.

Canales de varios tipos, que representan conexiones a entidades capaces de realizar operaciones de E/S.

Selectores, que junto con los canales seleccionables definen una E/S multiplexada y sin bloqueo. No los vamos a ver.

Otras novedades de NIO son que los canales son bidireccionales, por lo que se pueden utilizar tanto para leer como para escribir datos, así como que el flujo de datos de IO no permitía avanzar y retroceder en los datos. En este caso si era necesario avanzar y retroceder en los datos leídos, primero debíamos almacenarlos en un búfer. En el caso de NIO, de manera natural podemos acceder a los datos con desplazamientos en un sentido u otro.

Las clases del paquete java.io operan con técnica de bloqueo (*blocking approach*) para la gestión de las operaciones de entrada/salida (I/O), lo que significa que el hilo de ejecución que está realizando la operación de I/O queda bloqueado hasta que se cumple su finalización. De esta forma, sólo se puede realizar una operación de I/O por hilo de ejecución y el bloqueo puede resultar ineficiente en algunos escenarios.

En cambio, la API de NIO proporciona una alternativa de no-bloqueo (*non-blocking approach*) a través de los canales (channels) y los buffers (buffers), admite subprocesos múltiples para que los datos se puedan leer y escribir de forma asíncrona de tal manera que mientras se realizan esas operaciones de E/S no se bloquee el hilo actual, lo que lo hace más eficiente. Esto posibilita que múltiples operaciones de I/O

puedan ser gestionadas por un solo hilo de ejecución y reducir el tiempo de espera de una operación de I/O, especialmente en aplicaciones con mucha carga de I/O. El concepto de subprocesos múltiples se introduce con objetos *Selector* que permiten a un solo hilo de ejecución escuchar múltiples canales para eventos IO de forma asincrónica o sin bloqueo, realizando esas acciones sólo cuando alguna operación está disponible en uno de los canales monitorizados. Esta forma de operar es particularmente útil en aplicaciones de altas prestaciones, donde se necesita procesar un gran número de conexiones de I/O al mismo tiempo.

5.1. Buffers

Las clases de Buffer son la base sobre la que se construye Java NIO. De ellas, la clase *ByteBuffer* es la más utilizada por ser el tipo de byte más versátil. Por ejemplo, podemos usar bytes para componer otros tipos primitivos no booleanos como double, int, etc. Además, podemos usar bytes para transferir datos entre JVM y dispositivos de E/S externos.

Como vamos a ver, los búferes en Java NIO se pueden tratar como un contenedor de datos de tamaño fijo para escribir o leer datos de un canal, de forma que los búferes actúen como puntos finales de los canales. El uso de búferes permite que el paquete NIO sea más eficiente y rápido en comparación con el IO clásico.

Los parámetros primarios que definen el búfer NIO de Java se pueden definir como:

Capacidad - Cantidad máxima de datos / bytes que se pueden almacenar en el búfer. La capacidad de un búfer no se puede alterar. Una vez que el búfer está lleno, se debe borrar antes de escribir en él.

Límite - El límite tiene significado distinto según el modo de búfer; en el modo de escritura, el límite de búfer es igual a la capacidad, lo que significa que el máximo de datos que se pueden escribir en el búfer; mientras que en el modo de lectura, el límite de búfer toma el valor de la cantidad de datos que se pueden leer desde el búfer, es decir, los que se hayan almacenado.

Posición - Apunta a la ubicación actual del cursor en el búfer. Inicialmente establecido como 0 en el momento de la creación del búfer. En otras palabras, es el índice del siguiente elemento a leer o escribir, que se actualiza automáticamente al usar *get()* o *put()*.

Marca - Para situar un marcador de la posición en un búfer: cuando se llama al método *mark()*, se registra la posición actual y cuando se llama a *reset()* se restaura la posición marcada.

Tipos de búferes. Se pueden clasificar en las siguientes variantes en función de los tipos de datos que maneja el búfer:

Búferes básicos:

ByteBuffer

Búferes tipados (viewBuffers):

CharBuffer

DoubleBuffer

FloatBuffer

IntBuffer

LongBuffer

ShortBuffer

Los búferes también pueden ser clasificados según la memoria que los aloja:

Búferes directos: alojados directamente en memoria nativa, accesible directamente por el sistema operativo, usando el método *allocateDirect()*. Son más rápidos para operaciones con canales pero más costosos de crear o liberar su memoria.

```
ByteBuffer bf = ByteBuffer.allocateDirect(1024);
```

Búferes no directos: alojados en memoria de la máquina virtual, como cualquier objeto Java usual, usando *allocate()*. Son más rápidos para acceso desde dicha máquina virtual pero más lentos en operaciones de entrada-salida.

La recomendación es usar búferes directos sólo cuando necesitemos máximo rendimiento en I/O (por ejemplo, manipulación de ficheros grandes o en redes de alta velocidad). Evítalos para búferes pequeños o de uso efímero, ya que su creación/liberación es costosa.

Métodos importantes para trabajo con búfers (alguno de ellos se invocan desde un objeto canal y otros desde objeto búfer):

allocate(int capacity) - Este método se usa para asignar un nuevo búfer con capacidad como parámetro. El método de asignación arroja IllegalArgumentException en caso de que la capacidad pasada sea un entero negativo.

read() y *put()* - Son métodos para escritura en el búfer. El método de lectura se usa para leer datos del canal y escribirlos al búfer, mientras que el segundo es un método que se usa para escribir datos en el búfer desde otro búfer.

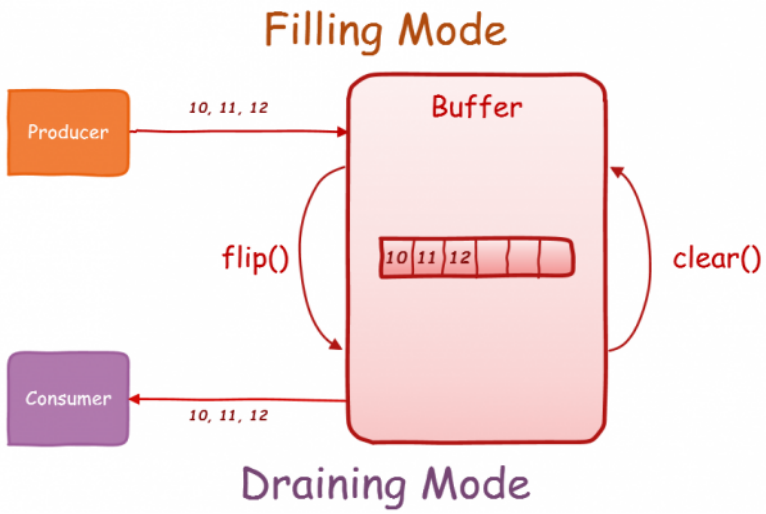
write() y *get()* - Son métodos para lectura en el búfer. El método de escritura se usa para escribir datos en un canal leyendo desde un búfer, mientras que *get* es un método para leer datos de otro búfer.

flip() - El método de inversión cambia el modo de búfer **de escritura a lectura**, también establece la posición de nuevo a 0 y establece el límite a donde estaba la posición en el momento de la escritura.

rewind() - El método de rebobinado se utiliza cuando es necesario volver a leer, ya que vuelve a poner la posición en cero y no altera el valor del límite.

clear() and *compact*() - Ambos métodos se utilizan para hacer que el búfer pase del modo **de lectura a escritura**. clear() hace que la posición sea cero y el límite sea igual a la capacidad, en este método los datos en el búfer no se borran, solo se reinicializan los marcadores. Por su parte, *compact()* se usa cuando quedan algunos datos no leídos y aún usamos el modo de escritura del búfer. El método copia todos los datos no leídos al principio del búfer y establece la posición justo después del último elemento no leído.

mark() y *reset()* - Como el nombre sugiere, este método se usa para marcar cualquier posición particular en un búfer, mientras que el segundo restablece la posición de regreso a la posición marcada.



ByteBuffer es clase abstracta, por lo que no podemos construir una nueva instancia directamente. Sin embargo, proporciona métodos estáticos para facilitar la creación de instancias. Hay dos formas de crear una instancia de ByteBuffer: mediante asignación (método *allocate*) o ajuste (método *wrap*, genera el

búfer a partir de un array dado). Para ser precisos, *ByteBuffer* tiene dos métodos de asignación: *allocate* y *allocateDirect*.

Con *allocate* obtenemos un búfer no directo, es decir, una instancia de búfer con una matriz de bytes subyacente.

```
ByteBuffer buffer = ByteBuffer.allocate(10);
```

En cambio, cuando usamos el método *allocateDirect*, genera un búfer directo.

```
ByteBuffer buffer = ByteBuffer.allocateDirect(10);
```

Para simplificar, vamos a ver sólo el trabajo con **búferes no directos**.

Por su parte, *wrap* permite que una instancia reutilice una matriz de bytes existente.

```
byte[] bytes = new byte[10];
ByteBuffer buffer = ByteBuffer.wrap(bytes);
```

Cualquier cambio realizado en los elementos del array de bytes se reflejará en la instancia del búfer y viceversa. En este caso también es un buffer no directo (guardado en la memoria de la máquina virtual).

El siguiente ejemplo muestra la implementación de los métodos vistos anteriormente.

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;

public class BufferDemo {
   public static void main (String [] args) {
      //crear buffer para 10 caracteres
      CharBuffer buffer = CharBuffer.allocate(10);
      String text = "bufferDemo";
      System.out.println("Texto de entrada: " + text);
      /* for (int i = 0; i < text.length(); i++) {
         char c = text.charAt(i);
         //escribir caracter en el buffer.
            buffer.put(c);
      } */
      buffer.put(text);     // más eficiente que el bucle anterior (todo de una, no caracter a caracter)
      int buffPos = buffer.position();
      System.out.println("Posición después de la escritura en el buffer: " + buffPos);
      buffer.flip();
      System.out.println("Leyendo contenido del buffer:");
      while (buffer.hasRemaining()) {
         System.out.println(buffer.get());
        // También podríamos haber leído todo de una utilizando el método toString()
      }
      // estableciendo la posición del buffer en 5.
      buffer.position(5);
      // estableciendo la marca según la posición
      buffer.mark();
      // intentando volver a cambiar la posición
      buffer.position(6);
      // utilizando el método reset para restaurar la posición marcada
      // reset() lanza InvalidMarkException si la marca no se puede establecer
      // o si la nueva posición es inferior que la posición marcada
      buffer.reset();
      System.out.println("Restaurada la posición del buffer: " + buffer.position());
   }
}
```

Métodos asXXXBuffer(): estos métodos de la clase *java.nio.ByteBuffer* se utilizan para crear una vista de un búfer de bytes como un búfer de otro tipo.

Por ejemplo, el método *asDoubleBuffer()* se utiliza para crear una vista del búfer de bytes como un búfer para datos de tipo *double* (de 8 en 8 bytes). El contenido del nuevo búfer comenzará desde la posición

actual de este búfer. Los cambios realizados en el contenido del búfer primero serán visibles en el nuevo búfer y viceversa; los valores de posición, límite y marca de los dos búferes serán independientes.

Podemos leer o escribir un solo byte desde/hacia los datos subyacentes del búfer en operaciones individuales. Estas operaciones incluyen:

```
public abstract byte get();
public abstract ByteBuffer put(byte b);
public abstract byte get(int index);
public abstract ByteBuffer put(int index, byte b);
```

Podemos notar dos versiones de los métodos get()/put() de los métodos anteriores: uno no tiene parámetros y el otro acepta un índice. Entonces, ¿cuál es la diferencia? La que no tiene índice es una operación relativa, que opera en el elemento de datos en la posición actual y luego incrementa la posición en 1. Sin embargo, la que tiene un índice es una operación completa, que opera en los elementos de datos en el índice y no cambiará la posición.

Por el contrario, las operaciones masivas pueden leer o escribir varios bytes desde/hacia los datos subyacentes del búfer. Estas operaciones incluyen:

```
public ByteBuffer get(byte[] dst);
public ByteBuffer get(byte[] dst, int offset, int length);
public ByteBuffer put(byte[] src);
public ByteBuffer put(byte[] src, int offset, int length);
```

Todos los métodos anteriores pertenecen a operaciones relativas. Es decir, leerán o escribirán desde/hasta la posición actual y cambiarán el valor de la posición, respectivamente.

ByteBuffer también proporciona otras vistas: *asCharBuffer()*, *asShortBuffer()*, *asIntBuffer()*, *asLongBuffer()*, *asFloatBuffer()* y *asDoubleBuffer()*.

5.2. Canales

Como el nombre sugiere, el canal se usa como medio de flujo de datos de un extremo a otro. Aquí el canal NIO actúa entre el búfer, en un extremo, y una entidad, como puede ser un fichero, en el otro extremo.

A diferencia de los flujos convencionales que se utilizan en Java, los canales son bidireccionales, es decir, pueden leer y escribir.

El canal Java NIO se implementa principalmente en las siguientes clases:

FileChannel - Para leer datos de un archivo. El objeto del canal de archivo solo se puede crear llamando al método getChannel() del objeto fichero; no podemos crear el canal directamente.

DatagramChannel - El canal de datagramas puede leer y escribir datos a través de la red con UDP (User Datagram Protocol), protocolo sin conexión.

SocketChannel - El canal SocketChannel puede leer y escribir datos en la red a través de TCP (Protocolo de control de transmisión), protocolo con conexión.

ServerSocketChannel - El ServerSocketChannel lee y escribe los datos a través de conexiones TCP como servidor, escuchando conexiones, como sucede, por ejemplo, con un servidor web. Para cada conexión entrante se crea un *SocketChannel*.

FileChannel permite leer y/o escribir datos en un archivo. Un objeto de este tipo se puede crear de 2 formas:

getChannel() - método en cualquier *FileInputStream*, *FileOutputStream* o *RandomAccessFile*.

open() - método estático en *FileChannel* que abre el canal.

El tipo de *FileChannel* depende de cómo se haya creado; es decir, si el objeto se crea llamando al método *getChannel* de *FileInputStream*, el canal de archivo se abre para lectura y lanzará *NonWritableChannelException* en caso de que intente escribir en él. Si lo creamos a partir de un *RandomAccessFile* el segundo parámetro del constructor determinará si podemos escribir o no.

El siguiente ejemplo lee de un archivo de texto de "temp.txt" e imprime el contenido en la consola.

```
import java.io.*;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
public class ChannelDemo {
   public static void main(String args[]) throws IOException {
      RandomAccessFile file = new RandomAccessFile("temp.txt","r");
      FileChannel fileChannel = file.getChannel();
      ByteBuffer byteBuffer = ByteBuffer.allocate(512);
      while (fileChannel.read(byteBuffer) > 0) {
         // flip the buffer to prepare for get operation
         byteBuffer.flip();
         while (byteBuffer.hasRemaining())
            System.out.print((char) byteBuffer.get());
      }
      file.close();
   }
}
```

Se deja al alumno corregir el error de ejecución que dará el anterior código cuando el fichero exceda los 512 bytes de longitud.

Veamos otro ejemplo, éste con escritura a fichero y posterior lectura. Observa los últimos caracteres mostrados y comprueba si es correcto.

```
import java.io.*;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;
import java.nio.file.*;
import java.util.*;

public class FileChannelDemo {
   public static void main(String args[]) throws IOException {
      // añade el contenido al fichero
      writeFileChannel(ByteBuffer.wrap("Welcome to DAM2".getBytes()));
      // lee el fichero
      readFileChannel();
   }

   public static void writeFileChannel(ByteBuffer byteBuffer)throws IOException {
      Set<StandardOpenOption> options = new HashSet<>();
      options.add(StandardOpenOption.CREATE);
      options.add(StandardOpenOption.APPEND);
      Path path = Paths.get("ejemplos/temp.txt");
      FileChannel fileChannel = FileChannel.open(path, options);
      fileChannel.write(byteBuffer);
      fileChannel.close();
   }

   public static void readFileChannel() throws IOException {
```

```
      RandomAccessFile randomAccessFile = new RandomAccessFile("ejemplos/temp.txt",
      "rw");
      FileChannel fileChannel = randomAccessFile.getChannel();
      ByteBuffer byteBuffer = ByteBuffer.allocate(512);
      Charset charset = Charset.forName("UTF-8");
      while (fileChannel.read(byteBuffer) > 0) {
         byteBuffer.rewind();
         System.out.print(charset.decode(byteBuffer));
         byteBuffer.flip();
      }
      fileChannel.close();
      randomAccessFile.close();
   }
}
```

5.3. Dispersión o reparto y recopilado

Como sabemos, Java NIO es una API más optimizada para operaciones de E/S de datos en comparación con la API de E/S convencional de Java. Una funcionalidad añadida por NIO es leer o escribir datos desde / hacia múltiples búferes al canal. Este procedimiento se denomina *Scatter and Gather*, en el que los datos se reparten (*scattering*) entre múltiples búferes de un solo canal en caso de leer datos, mientras que los datos se recopilan o reúnen (*gathering*) desde múltiples búferes a un solo canal en caso de escribir datos.

Para lograr esta lectura y escritura múltiples desde el canal, existen *ScatteringByteChannel* y *GatheringByteChannel* que Java NIO proporciona para leer y escribir los datos como se ilustra en el siguiente ejemplo.

En él leeremos datos de un solo canal en múltiples búferes. Para esto, se asignan múltiples búferes y se agregan a un array de búferes. Luego, esta matriz se pasa como parámetro al método *read()* de *ScatteringByteChannel* que luego escribe datos desde el canal en la misma secuencia en que los búferes están en el array. Una vez que un búfer está lleno, el canal se mueve para llenar el siguiente búfer.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.ScatteringByteChannel;

public class ScatterExample {
        private static String FILENAME = "nums.dat";
        public static void main(String[] args) {
                ByteBuffer bLen1 = ByteBuffer.allocate(1024);
                ByteBuffer bLen2 = ByteBuffer.allocate(1024);
                FileInputStream in;
                try {
                        in = new FileInputStream(FILENAME);
                        ScatteringByteChannel scatter = in.getChannel();
                        scatter.read(new ByteBuffer[] {bLen1, bLen2});
                        bLen1.position(0);
                        bLen2.position(0);
                        int valor1 = bLen1.asIntBuffer().get();
                        int valor2 = bLen2.asIntBuffer().get();
                        System.out.println("Scattering : Valor1 = " + valor1);
                        System.out.println("Scattering : Valor2 = " + valor2);
                }
                catch (FileNotFoundException exObj) {
```

```
                        exObj.printStackTrace();
                }
                catch (IOException ioObj) {
                        ioObj.printStackTrace();
                }
        }
}
```

Veamos ahora el recopilado.

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.GatheringByteChannel;

public class GatherExample {
        private static String FILENAME = "nums2.dat";
        public static void main(String[] args) {
                String stream1 = "Recopilando datos del primer búfer";
                String stream2 = "Recopilando datos del segundo búfer";
                ByteBuffer bstream1 = ByteBuffer.wrap(stream1.getBytes());
                ByteBuffer bstream2 = ByteBuffer.wrap(stream2.getBytes());
                // Los siguientes dos búferes guardarán los datos a escribir
                ByteBuffer bLen1 = ByteBuffer.allocate(1024);
                ByteBuffer bLen2 = ByteBuffer.allocate(1024);
                int len1 = stream1.length();
                int len2 = stream2.length();
                // Escribimos la longitud de los datos en los búfers
                bLen1.asIntBuffer().put(len1);
                bLen2.asIntBuffer().put(len2);
                System.out.println("Recopilando: " + len1);
                System.out.println("Recopilando : " + len2);
                System.out.println("Recopilando primer texto: " + stream1);
                System.out.println("Recopilando segundo texto:" + stream2);
                // Escribimos datos al fichero
                try {
                        FileOutputStream out = new FileOutputStream(FILENAME);
                        GatheringByteChannel gather = out.getChannel();
                        // Escribimos al canal, por este orden, los 2 búferes que contienen un entero
                        // cada uno y, a continuación, los 2 búferes con el texto
                        gather.write(new ByteBuffer[] {bLen1, bLen2, bstream1, bstream2});
                        System.out.println("Observa el tamaño final del fichero");
                        out.close();
                        gather.close();
                }
                catch (FileNotFoundException exObj) {
                        exObj.printStackTrace();
                }
                catch(IOException ioObj) {
                        ioObj.printStackTrace();
                }
        }
}
```

Por último, se puede concluir que el enfoque de dispersión / recopilación en Java NIO se presenta como un método optimizado y multitarea cuando se usa correctamente. Permite delegar en el sistema operativo el trabajo duro de separar los datos que lee en múltiples áreas o ensamblar fragmentos dispares de datos en un todo.

6. FICHEROS XML

Los ficheros XML permiten el intercambio de información entre aplicaciones utilizando ficheros de texto con etiquetas que dan significado a cada uno de los valores almacenados.

Para trabajar con datos en XML desde nuestras aplicaciones Java tenemos dos modelos o tipos de procesadores XML, son **DOM** (Document Object Model) y **SAX** (Sample API for Xml).

Básicamente la diferencia entre uno y otro es que el primero requiere tener en memoria cargado todo el documento para su procesamiento, mientras que el segundo no. Eso hace que el primero, DOM, que es el que vamos a utilizar, consuma más memoria RAM pero su procesamiento sea más rápido.

Por ejemplo, el siguiente fichero XML podría ser el resultado de volcar una base de datos sobre productos de una empresa, de forma que dicha información podría ahora leerse desde otra aplicación e incorporarla. Se puede ver cómo, aparte de los datos de dichos productos (sus nombres, precios, . . .), aparece otra información en forma de etiquetas (entre los caracteres < y >) que permite dar significado a cada dato almacenado en el fichero.

Obviamente, tal y como ocurría con los ficheros de configuración, toda la información se tiene que pasar a texto para poder crear el fichero y reconvertida a su tipo original cuando se cargue de nuevo.

```
<?xml version="1.0" encoding="UTF-8" standalone="no">
<xml>
<productos>
  <producto>
    <nombre>Cereales</nombre>
    <precio>3.45</precio>
  </producto>
  <producto>
    <nombre>Colacao</nombre>
    <precio>1.45</precio>
  </producto>
  <producto>
    <nombre>Agua mineral</nombre>
    <precio>1.00</precio>
  </producto>
</productos>
</xml>
```

La clase Java que definiría cada uno de los objetos que se representan por el fichero XML anterior, sería la siguiente:

```
public class Producto {
  private String nombre;
  private float precio;

  public Producto(String nombre, float precio) {
    this.nombre = nombre;
    this.precio = precio;
  }

  public String getNombre () { return nombre; }
  public void setNombre (String nombre) { this.nombre = nombre; }

  public float getPrecio () { return precio; }
  public void setPrecio (float precio) { this.precio = precio; }
}
```

6.1. Escribir ficheros XML

Para poder trabajar con DOM en Java utilizaremos los paquetes *org.w3c.dom*, *javax.xml.parsers* y *javax.xml.transform*.

Haremos uso de las clases o interfaces siguientes: *Document*, para representar el documento XML, *Element*, *Node*, *NodeList*, *Attr*, *Text*, etc.

El punto de partida será crear un objeto *DocumentBuilderFactory* y, a partir de éste, un *DocumentBuilder*.

Ahora crearemos el documento con su nodo raíz de nombre "productos" en plural. Lo hacemos a partir de la interfaz *DOMImplementation*.

El siguiente paso ya sería recorrer el fichero con los datos de los productos y, por cada uno, crear un nodo "producto" con 2 elementos hijos (nombre, precio). Cada nodo hijo tendrá su valor. Para crear cada elemento utilizaremos el método *createElement(String)* y para su valor utilizaremos *createTextNode(String)*. Cada elemento o valor añadido lo conectaremos con su ascendente en la jerarquía del documento mediante el método *appendChild(Element)* o *appendChild(Text)*.

Y el último paso será crear la fuente XML y el destino (fichero). Se crea un objeto *DOMSource* a partir del documento, se obtiene el destino o resultado de tipo *StreamResult* para el fichero "productos.xml", se obtiene un *TransformerFactory* y se realiza la transformación:

```
. . .
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
DOMImplementation dom = builder.getDOMImplementation();
Document documento = dom.createDocument(null, "xml", null);
Element raiz = documento.createElement("productos");
documento.getDocumentElement().appendChild(raiz);
Element nodoProducto = null , nodoDatos = null ;
Text texto = null;

for (Producto producto : listaProductos) {
  nodoProducto = documento.createElement("producto");
  raiz.appendChild(nodoProducto);
  nodoDatos = documento.createElement("nombre");
  nodoProducto.appendChild(nodoDatos);
  texto = documento.createTextNode(producto.getNombre());
  nodoDatos.appendChild(texto);
  nodoDatos = documento.createElement("precio");
  nodoProducto.appendChild(nodoDatos);
  texto = documento.createTextNode(producto.getPrecio());
  nodoDatos.appendChild(texto);
}
Source source = new DOMSource(documento);
Result resultado = new StreamResult(new File("productos.xml"));
Transformer transformer = TransformerFactory.newInstance().newTransformer();
transformer.setOutputProperty("indent","yes");
transformer.transform(source,resultado);
. . .
```

Puedes ver cómo, para ello, hemos de crear un objeto *DocumentBuilder* a partir de un *DocumentBuilderFactory*. Una vez lo tenemos, generamos un *DOMImplementation* y, sobre éste, un objeto *Document*. Con él creado, utilizamos los métodos *createElement*, *getDocumentElement*, *appendChild* y *createTextNode* para generar la estructura del XML.

Observa cómo los *createElement* y *createTextNode* son siempre sobre el objeto *Document*, mientras que los *appendChild* se realizan sobre el objeto que los contiene.

6.2. Leer ficheros XML

Si lo que queremos es leer un fichero XML y cargarlo como una colección de objetos *Producto*, el ejemplo siguiente muestra cómo acceder a la información de dicho fichero XML.

Creamos el *DocumentBuilderFactory* y el consiguiente *DocumentBuilder* para cargar el documento con el método *parse()*. A continuación, obtenemos la lista de nodos con el nombre "producto" y con un bucle

recorremos la lista. Para cada nodo obtenemos sus etiquetas y valores con los métodos *getElementsByTagName*, *getChildNodes()* y *getNodeValue()*.

```
     DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
     DocumentBuilder builder = factory.newDocumentBuilder();
     Document documento = builder.parse(new File("productos.xml"));

NodeList productos = documento.getElementsByTagName("producto");
for (int i = 0; i < productos.getLength(); i++) {
  Node producto = productos.item(i) ;
  Element elemento = ( Element ) producto ;
  System.out.println(elemento.getElementsByTagName("nombre").item(0)
                     .getChildNodes().item(0).getNodeValue());
  System.out.println(elemento.getElementsByTagName("precio").item(0)
                     .getChildNodes().item(0).getNodeValue());
}
. . .
```

Observa cómo para la lectura, comenzamos por crear el *DocumentBuilderFactory*, a continuación el *DocumentBuilder* y por último el *Document* mediante el método *parse*.

A continuación, obtenemos un *NodeList* con todas las etiquetas *<producto>* y, mediante un bucle, vamos desgranando uno a uno el contenido para mostrarlo en pantalla.

7. JAXB, CONVERSIÓN ENTRE XML Y OBJETOS

Java Architecture for XML Binding (JAXB) permite convertir objetos Java a XML (hablaremos en tal caso de serializar o *marshalling*), y a la inversa (deserializar o *unmarshalling*).

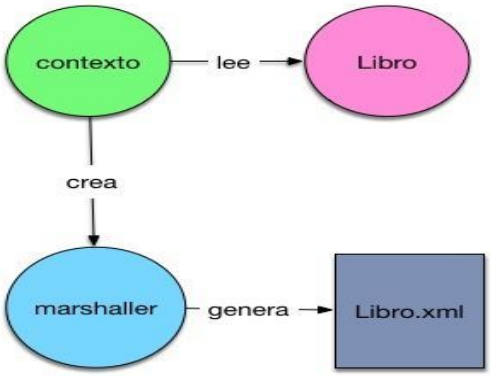
Vamos a ver una introducción a este estándar y sus anotaciones. Para ello utiliza la clase Producto, una clase Productos con una lista de productos y las anotaciones de JAXB:

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

@XmlRootElement(name = "productos")
public class Productos {
/* Si tuviéramos algún error "Class has two properties of the same name productos"
habríamos de poner la siguiente anotación no en el atributo,
sino en el getter correspondiente */
@XmlElement(name = "producto")
private List<Producto> productos;

public Productos() {
    productos = new ArrayList<Producto>();
}

public Productos(List<Producto> productos) {
    this.productos = productos;
}

...
}
```

Hemos añadido dos anotaciones principales: *@XmlRootElement* que especifica la clase raiz que vamos a convertir a XML, y *@XmlElement* que permite cambiar el nombre de los elementos cuando el fichero XML se construya. Es momento de generar el fichero XML .

```java
import javax.xml.bind.*;

public class Principal {
    public static void main(String[] args) {
        try {
            Productos productos = new Productos();
            ... // cargar productos a la lista
            JAXBContext contexto = JAXBContext.newInstance(
                    productos.getClass() );
                Marshaller marshaller = contexto.createMarshaller();
                marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                    Boolean.TRUE);
                marshaller.marshal(productos, System.out);
        } catch (PropertyException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (JAXBException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Disponemos de dos objetos: el contexto y el marshaller. El contexto se encarga de definir los objetos que vamos a utilizar y el marshaller en qué forma vamos a generar la estructura.



La misma operación se puede realizar al contrario , leer un fichero XML y rellenar un objeto Java.

```java
try {
    JAXBContext context = JAXBContext.newInstance(Productos.class );
    Unmarshaller unmarshaller = context.createUnmarshaller();
    Productos productos = (Productos)unmarshaller.unmarshal(new File("productos.xml") );
    System.out.println(productos);
} catch (JAXBException e) {
    e.printStackTrace();
}
```

Para ello previamente habrás de incluir en tu proyecto la/s dependencia/s requeridas para JAXB. Consulta éstas en Internet.