UNIDAD 1. Gestión de ficheros y serialización de objetos

Contenido

1. GESTIÓN DE FICHEROS Y DIRECTORIOS

Los ficheros o archivos y directorios del disco se representan de forma lógica en las aplicaciones Java como objetos de la clase File.

La clase File no se utiliza para transferir datos entre la aplicación y el disco, sino para obtener información sobre los ficheros y directorios de éste e incluso para la creación y eliminación de los mismos.

1.1. Creación de un objeto File

Existen diversas formas de crear un objeto File en función de cómo se indique la localización del fichero o directorio que va a representar. Por ejemplo, el constructor *Füe(String path)* permite construir un objeto File a partir de su dirección absoluta o relativa al directorio actual:

```
File f = new File ("datos.txt");

File f2 = new File ("misubdirectorio");
```

La creación del objeto no implica que exista el fichero o directorio indicado en la ruta. Si éste no existe, las anteriores instrucciones no provocarán ninguna excepción, aunque tampoco será creado de forma implícita.

Para crear físicamente el fichero o directorio indicado en el constructor de File habrá que recurrir a los siguientes métodos de la clase:

- boolean createNewFile(). Crea el fichero cuyo nombre ha sido especificado en el constructor. Devuelve *true* si se ha podido crear el fichero, mientras que el resultado será *false* si ya existía y por tanto no ha sido creado. Por ejemplo, si el fichero datos.txt del ejemplo anterior no existe, se podría ejecutar la siguiente instrucción para crearlo:

  ```
  f.createNewFile();
  ```

  La llamada a *createNewFile()* puede provocar una excepción lOException que habrá que capturar.

  Si un objeto File hace referencia a un fichero no existente y no es creado de forma explícita invocando a *createNewFile(),* la creación del mismo se llevará a cabo implícitamente cuando se vaya a utilizar el objeto File para construir un objeto Writer u OutputStream, a fin de realizar una operación de escritura sobre el fichero.

- boolean mkdir(). Crea el directorio cuyo nombre ha sido especificado en el constructor. Si el directorio no existía y se ha podido crear, el método devolverá *true,* de lo contrario el resultado será *false*.

  Por ejemplo, para crear físicamente el subdirectorio referenciado por el objeto File f2 sería:

  f2.mkdir();

La clase File también proporciona un constructor que permite crear un objeto File asociado a una carpeta:

File (File dir, String nombre_fichero)

En este caso, la carpeta especificada en *dir* debe existir, de lo contrario se producirá una excepción al intentar utilizar el fichero para realizar cualquier operación sobre el mismo.

El siguiente bloque de instrucciones crea una carpeta "carpetaDatos" y en su interior crea un fichero "info.txt":

```
File carpeta = new File("carpetaDatos");
carpeta.mkdir();
File fichero = new File(carpeta, "info.txt");
fichero.createNewFile(); //Si carpetaDatos no existiese provocaría IOException
```

1.2. Información sobre un fichero/directorio

Una vez creado el objeto File asociado al fichero o directorio, podemos obtener información del mismo aplicándole los siguientes métodos de la clase File:

- boolean canRead(). Indica si se puede o no leer el fichero.
- boolean canWrite(). Indica si se puede o no escribir en el fichero.
- boolean exists(). Indica si existe o no el fichero o directorio indicado en la ruta.
- boolean isFile(). Indica si el objeto File hace referencia o no a un fichero.
- boolean isDirectory(). Indica si el objeto File hace referencia o no a un directorio.
- String getName(). Devuelve el nombre del fichero sin el path.
- String getAbsolutePath(). Devuelve el path absoluto completo.

1.3. Eliminación y renombrado

Para eliminar físicamente un fichero o directorio la clase File proporciona el método *delete*() con el siguiente formato:

• boolean delete(). Elimina el fichero o directorio especificado por el objeto. Si el elemento ha podido ser eliminado el método devolverá *true,* si no devolverá *false*. Un directorio sólo podrá ser eliminado si está vacío; si no lo está, la llamada a *delete*() sobre el objeto File devolverá *false,* aunque no provocará ninguna excepción.

Si lo que queremos es renombrar un fichero o directorio existente, utilizaríamos el siguiente método:

• boolean renameTo(File nuevo). Renombra el fichero o directorio, asignándole el nombre del objeto File especificado. La llamada a este método devolverá *true* si se ha podido renombrar el elemento, mientras que el resultado será *false* si esto no ha sido posible. En el caso de un directorio, no es necesario que esté vacío para poderlo renombrar.

El siguiente ejemplo renombra un fichero existente:

```
File f=new File ("fichero.txt");
f.createNewFile(); //para poderlo renombrar
```

```
//debe existir

File f2=new File ("nuevo_nombre.txt );

f.renameTo (f2) ; //renombra "fichero.txt" a

//"nuevo_nombre.txt"
```

2. LECTURA DE UN FICHERO DE TEXTO

Muchas de las operaciones realizadas con ficheros en un programa tienen que ver con la lectura de ficheros de texto.

Para recuperar cadenas de caracteres de un fichero, el paquete java.io proporciona dos clases: FileReader y BufferedReader. A continuación, veremos cómo se utilizan estas clases, examinando los dos pasos que hay que seguir para la recuperación de cadenas de caracteres de un fichero.

2.1. Creación de un objeto FileReader

Para poder recuperar información de un fichero de texto, es necesario primeramente crear un objeto FileReader asociado al mismo. Un objeto FileReader representa un fichero de texto "abierto" para la lectura de datos. Este objeto es capaz de adaptar la información recuperada del fichero a las características de una aplicación Java, transformando los bytes almacenados en el mismo en caracteres unicode.

Se puede construir un objeto FileReader a partir de un objeto File existente o bien proporcionando directamente la ruta del fichero, a partir de los siguientes constructores:

FileReader (String path)

FileReader (File fichero)

La siguiente instrucción crearía un objeto FileReader que haga referencia al fichero datos.txt:

```
File f = new File ("datos.txt");

FileReader fr = new FileReader(f);
```

La clase FileReader proporciona el método r*ead()* para la lectura de la información almacenada en un fichero, resultando su uso bastante engorroso ya que los caracteres son recuperados como tipo byte, debiendo ser posteriormente convertidos a String. Es por ello que resulta más cómodo recurrir a la clase BufferedReader para realizar esta operación, utilizando el objeto FileReader como puente para crear un objeto de este tipo.

El siguiente fragmento de código muestra la lectura de un fichero carácter a carácter.

```
File f = new File("hola.txt");
int c;
FileReader fr = new FileReader(f);
while ((c = fr.read()) != -1){
    System.out.print((char)c);
}
```

2.2. Creación de un objeto BufferedReader

Una vez creado el objeto FileReader, el segundo paso consiste en crear un BufferedReader, para lo cual utilizaremos el constructor:

BufferedReader (Reader entrada)

Para leer de un fichero el procedimiento es el mismo que cuando se utiliza para leer cadenas de caracteres desde el teclado, sólo que en este caso el objeto Reader que hay que proporcionar al constructor será el objeto FileReader asociado al fichero:

```
FileReader fr = new FileReader(f);

BufferedReader bf=new BufferedReader(fr);
```

Una vez creado el objeto, puede utilizarse el método *readLine()* para leer líneas de texto del fichero de forma similar a como se leían del teclado. En el caso de un fichero, y dado que éste puede estar formado por más de una línea, será necesario utilizar un bucle *while* para recuperar todas las líneas de texto del mismo de forma secuencial.

El siguiente programa muestra por pantalla el contenido completo de un fichero llamado datos.txt:

```
File f = new File("hola.txt");
if (f.exists()) {
    FileReader fr = new FileReader(f);
    BufferedReader bf = new BufferedReader(fr);
    String linea;
    while ((linea = bf.readLine()) != null ){
        System.out.println(linea);
    }
}
```

Según se desprende del programa anterior, el método *readLine()* apunta a la siguiente línea de texto después de recuperar la línea actual. Cuando no existan más líneas para leer, la llamada a *readLine()* devolverá *null*.

Si lo que se quiere es leer carácter a carácter en vez de línea a línea, deberíamos utilizar el método *read()* en lugar de *readLine()*. El método *read()* devuelve un entero que representa el código unicode del carácter leído, siendo el resultado -1 si no hay más caracteres para leer.

2.3. Lectura con Files (programación funcional)

También se puede utilizar desde Java 8, Files.lines, para leer ficheros con expresiones lambda:

```
Files.lines(Path.of("hola.txt")).forEach(linea->
System.out.println(linea));
```

También podemos filtrar el fichero, o utilizar cualquier otra expresión lambda:

```
Files.lines(Path.of("hola.txt"))
        .filter(linea->linea.startsWith("h"))
        .forEach(linea-> System.out.println(linea));
```

Es importante destacar que Files.lines devuelve un Stream, vinculado a un fichero, por tanto debe ser cerrado para liberarlo. Para manejar esto de forma segura , se debe utilizar el bloque try-with-resources, que asegura que el recurso se cerrará automáticamente al terminar:

```
try (Stream<String> lineas = Files.lines(Path.of("hola.txt"))) {
    lineas.forEach(linea -> System.out.println(linea));
} catch (IOException e) {
    e.printStackTrace();
}
```

3. ESCRITURA EN FICHEROS DE TEXTO

La realización de esta operación también se lleva a cabo en dos pasos que implican la utilización de dos clases: FileWriter y PrintWriter.

3.1. Creación de un objeto FileWriter

Como primer paso para escribir en un fichero de texto, es necesario construir un objeto FileWriter que nos permita tener acceso al fichero en modo escritura. Para ello, podemos utilizar uno de los siguientes constructores:

- FileWriter (String path)
- FileWriter (File fichero)
- FileWriter (String path, boolean append)
- FileWriter (File fichero, boolean append)

Donde el parámetro *append* permite indicar si los datos que se van a escribir se añadirán a los ya existentes *(true)* o sobreescribirán a éstos *(false)*. Si se utiliza uno de los dos primeros constructores, los datos escritos en el fichero sustituirán a los existentes (equivalente a append=false).

La clase FileWriter proporciona un método *write()* que permite escribir en el fichero la cadena de caracteres pasada como parámetro, aunque también puede utilizarse la clase estándar de escritura PrintWriter para realizar esta operación.

3.2. Creación del objeto PrintWriter

La utilización de la clase PrintWriter posibilita que la escritura sobre un fichero se realice de la misma forma que la escritura en pantalla:

```
1° FileWriter fw = new FileWriter("datos.txt");

2° PrintWriter out = new PrintWriter(fw) ;
```

Desde la versión Java 5 también es posible crear un objeto PrintWriter a partir de un objeto File o incluso de la ruta del fichero, por lo que las dos instrucciones anteriores podrían reducirse a una sola:

```
PrintWriter out = new PrintWriter ("datos.txt");
```

Una vez creado el objeto PrintWriter, podemos utilizar los métodos print(), println() y printf() para escribir en el fichero. En el siguiente ejemplo usamos try-with-resources también para ahorrarnos tener que cerrar el fichero:

```
try (PrintWriter pw = new PrintWriter("hola.txt")) {
    pw.println("Escribiendo..");
    pw.flush();
    //pw.close();
}
```

Obsérvese la utilización de los métodos *flush()* y *close()* de PrintWriter después de la escritura en el fichero. La llamada al método *flush()* garantiza que todos los datos enviados a través del buffer de salida han sido escritos en el fichero, mientras que *close()* cierra la conexión con el fichero y libera los recursos utilizados por ésta.

3.3. Escritura con Files (programación funcional)

Igual que leíamos con Files, también se puede escribir utilizando programación funcional:

```
Files.write(Path.of("hola.txt"), "Hola".getBytes());
```

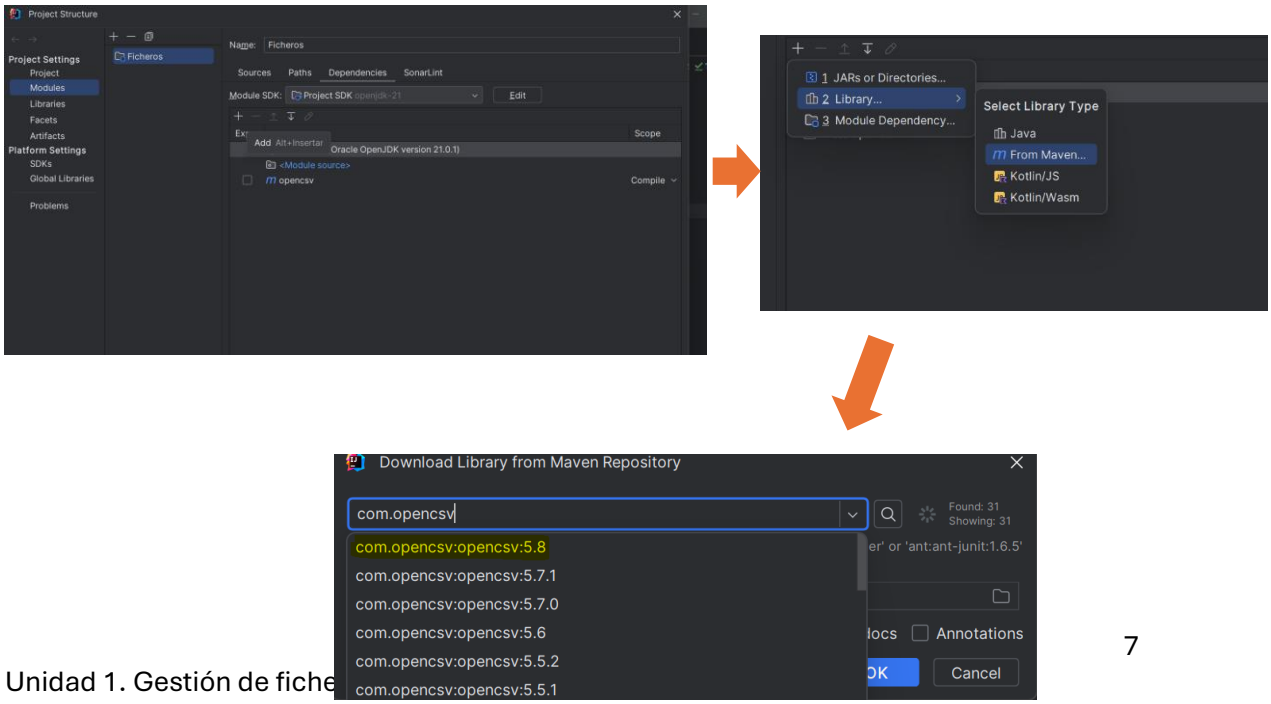
Si queremos que en lugar de reemplazar el contenido del fichero se añada, utilizamos StandardOpenOption.APPEND:

```
Files.write(Path.of("hola.txt"), "Holaa".getBytes(),
StandardOpenOption.APPEND);
```

4. OPENCSV: PARSEO DE FICHEROS CSV

4.1. Configuración de OpenCSV

Hay que añadir la dependencia de esta librería OpenCSV y la forma más fácil es mediante Maven. Se hace de esta manera. Abrimos la configuración de modules (open module settings) de Intellij y añadimos la librería Maven:



4.2. CSVReader

En la librería openCSV se identifican 2 clases principales:

- **CSVParser**. Es el parseador, por tanto, el encargado de procesar el fichero CSV para convertirlo en filas. La forma en que debe dividir el fichero es flexible y permite añadirle las configuraciones que necesitemos. Las principales son:
  - *withSeparator(String separator)*. Es la que permite configurar el parseador para especificarle cómo separar cada columna dentro de la fila. Por defecto es la coma (“,”), ya que es el formato CSV estándar.
  - *withIgnoreQuotations(boolean ignore)*. Si lo ponemos a true (que es la configuración por defecto) ignorará las comillas que pongamos en los valores de las columnas. Y si lo establecemos a falso lo considerará parte del valor.
- **CSVReader**. Permite leer línea a línea un fichero CSV. Permite añadirle algunas configuraciones, donde destacan:
  - *withCSVParser(CSVParser parser)*. Permite pasar un CSVParser para que ya lleve la configuración de parseado deseada.
  - *withSkipLines (int línea)*. Se puede especificar a partir de qué línea se quiere saltar la lectura. Si se pasa el parámetro 0, no se saltaría ninguna.

Si tenemos un fichero CSV con algunos datos separados por comas:

```
FICHERO CSV SUPER CHULO
23,"hola",2
11,"h",4
```

Con las clases comentadas anteriormente podríamos procesarlo columna a columna:

```
try (
        Reader reader = Files.newBufferedReader(Path.of("prueba.csv"));
        CSVReader csvReader = new CSVReaderBuilder(reader)
                .withSkipLines(1) // procesar a partir de segunda línea
                .withCSVParser(new CSVParserBuilder().build())
                .build()
        ) {
            String[] linea;
            while ((linea = csvReader.readNext()) != null) {
                for (String columna : linea) {
                    System.out.println("Columna: " + columna);
                }
            }
        }
```

4.3. CSVWriter

Dispone de 2 métodos principales:

- WriteNext(String[] linea). Permite escribir línea a línea.
- WriteNext(List<String[]>). Permite escribir una lista de líneas.

```
try (CSVWriter writer = new CSVWriter(new
FileWriter(Path.of("prueba2.csv").toString()))) {
    List<String[]> filas = new LinkedList<>();
    filas.add(new String[]{"1", "Ford Fiesta"});
    filas.add(new String[]{"2", "Ford Mondeo"});
    writer.writeAll(filas);
} catch (IOException e) {
   e.printStackTrace();
}
```

4.4. csvToBean: Parseado de un fichero CSV a una lista de objetos

La clase csvToBean permite mapear automáticamente las columnas del fichero con los atributos de una clase Java, permitiendo importar datos de forma masiva de forma rápida y limpia en forma de lista de objetos.

Para ello se utiliza la clase CsvToBeanBuilder, donde se especifica la clase a parsear. Finalmente, el método parse() realiza el parseado.

```
try (FileReader fileReader = new FileReader("Alumno.csv")) {
    // Se crea un csvToBean de clase Alumno
    CsvToBean<Alumno> csvToBean = new
CsvToBeanBuilder<Alumno>(fileReader)
            .withType(Alumno.class)
            .build();
    // Parsea el fichero CSV en una lista de alumnos
    List<Alumno> listaAlumnos = csvToBean.parse();
} catch (IOException e) {
    System.out.println("Error");
}
```

Para que se produzca el mapeo (el rellenado del objeto con los datos del fichero), **la clase** Alumno **debe incorporar el constructor vacío**. Además, **deben coincidir el nombre de los atributos con los nombres de columnas introducirlas en la primera línea del fichero CSV**:

```
nombre,edad,id
Juan,20, 1
Maria,22,2
Luis,18,3
```

Fichero Alumno.csv

Sin embargo, openCSV es insensible a mayúsculas y minúsculas, así que si pusiéramos EDAD o Edad seguiría haciendo el mapeo correctamente. Pero, si quisiéramos cambiar el nombre de la columna del fichero debemos utilizar las etiquetas etiquetas "*CsvBindByName*", especificando el nombre de columna deseado.

También podemos hacer que ignore un campo con la etiqueta @CsvIgnore. Eso implica que si el fichero CSV tiene el atributo edad y sus valores, no serán mapeados al objeto.

```
public class Alumno {
    @CsvBindByName(column= ”nombreAlumno”)
    private String nombre;
    @CsvBindByName(column=”edadAlumno”)
    private int edad;
    @CsvIgnore
    private int campoNoNecesario;

    public Alumno(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public int getEdad() {
        return edad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

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

4.5. Serialización de una colección de objetos a un fichero CSV

Una práctica habitual en la programación es querer realizar la exportación de una carga de objetos a un fichero externo para su posterior recuperación o para que otro usuario pueda realizar dicha carga masiva. De esta manera, el usuario se ahorra el trabajo de volver a introducir los elementos uno por uno vía Interfaz Gráfica, lo cual puede ser tedioso si son muchos registros.

La librería openCSV ofrece una serie de funciones para exportar/importar objetos en un fichero CSV.

El método build() de la clase StatefulBeanToCSVBuilder genérica permite a partir de un PrintWriter crear un objeto StatefulBeanToCSV vinculado a él, que permitirá parsear una lista de objetos a CSV. Aquí se muestra un ejemplo de uso:

```
List<Alumno> listaAlumnos = new ArrayList<>();
Collections.addAll(listaAlumnos,
        new Alumno("Javier", 29),
        new Alumno("Jose", 39),
        new Alumno("Ana", 50));
try (PrintWriter pw = new PrintWriter("Alumno.csv")) {
    StatefulBeanToCsv<Alumno> beanToCsv = new
StatefulBeanToCsvBuilder<Alumno>(
            pw).build();
    beanToCsv.write(listaAlumnos);
} catch (FileNotFoundException | CsvDataTypeMismatchException |
```

```
CsvRequiredFieldEmptyException e) {
    e.printStackTrace();
}
```

Teniendo en cuenta que tenemos una clase Alumno:

```
public class Alumno {
    private String nombre;
    private int edad;

    public Alumno(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
```

Se generará este fichero CSV:

```
"EDAD","NOMBRE"
"29","Javier"
"39","Jose"
"50","Ana"
```

5. SERIALIZACIÓN Y DESERIALIZACIÓN DE OBJETOS

La **serialización de objetos** en Java es un proceso mediante el cual un **objeto se convierte en una secuencia de bytes**, lo que permite su almacenamiento o transmisión. Este proceso es crucial para persistir el estado de un objeto o para enviarlo a través de la red a otra aplicación o máquina virtual Java. La **deserialización** es el proceso inverso, es decir, que **se reconstruye el objeto original a partir de esta secuencia de bytes**.

Una de las APIs más utilizadas para la serialización/deserialización de objetos es JAXB ("*Java Architecture for XML Binding*"). Esta API permite mapear objetos Java a elementos XML y viceversa.

5.1. Configuración de JAXB

Lo primero hay que hacer para poder usar la API JAXB es añadir 2 dependencias:

- javax.xml.bind:jaxb-api:2.4.0-b180830.0359
- org.glassfish.jaxb:jaxb-runtime:2.4.0-b180830.0438

5.2. Serialización de objetos con JAXB

Supongamos una clase que queremos serializar, en nuestro caso: clase Persona:

```
@XmlRootElement // Raíz del documento XML
public class Persona {
    private String nombre;
    private int edad;

    // Constructor sin argumentos necesario para JAXB
    public Persona() {}

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    @XmlElement // Elemento XML
    public String getNombre() {
        return nombre;
    }

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

    @XmlAttribute
    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}
```

Destacan 2 etiquetas principales de JAXB:

- **@XMLRootElement**. Se añade encima de la clase e indica a JAXB que se trata del elemento raíz del documento XML.
- **@XMLElement**. Se utiliza para indicar que un método (normalmente un getter) representa un elemento de XML.
- **@XMLAttribute**. Es muy parecido a la etiqueta @XMLElement, pero en lugar de representar un elemento XML, representaría un atributo del elemento raíz XML.

También hay que destacar que es **imprescindible** tener un **constructor por defecto** para que JAXB pueda crear la instancia de la clase durante la deserialización.

Una vez definida la clase con sus etiquetas JAXB, para serializar un objeto en XML utilizando JAXB se haría de esta manera:

```
try {
    Persona persona = new Persona("Juan", 30);
    JAXBContext context = JAXBContext.newInstance(Persona.class);
    Marshaller marshaller = context.createMarshaller();
    marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
    marshaller.marshal(persona, System.out);
} catch (JAXBException e) {
```

```
    e.printStackTrace();
}
//SALIDA:
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<persona edad="30">
    <nombre>Juan</nombre>
</persona>
```

La línea `marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);` no es imprescindible, ya que solo lo convierte en un formato legible con saltos de línea

Por simplicidad, existe otra manera de serializar con JAXB sin necesidad de introducir las etiquetas ni tener que introducir getter/setter. Usando la etiqueta @XmlAccessorType(XmlAccessType.FIELD). Veamos un ejemplo:

```
@XmlRootElement // Raíz del documento XML
@XmlAccessorType(XmlAccessType.FIELD) // Acceso directo a campos
public class Persona {

    @XmlAttribute // Mapea 'edad' como atributo XML
    private int edad;
    private String nombre;

    // Constructor sin argumentos necesario para JAXB
    public Persona() {}

}
```

Si hubiera un campo que no quisiéramos mapear, se añade la etiqueta @XMLTransiente:

```
@XmlTransient
private String password
```

Por tanto, la etiqueta **@XmlAccessType.FIELD mapea automáticamente todos los campos no estáticos y no transitorios** a elementos o atributos XML, sin necesidad de métodos getter o setter para serializar o deserializar los datos.

5.3. Deserialización de objetos con JAXB

Para deserializar un xml en un objeto Java con JAXB se haría de esta manera:

```
try {

   String xml = "<persona
edad=\"30\"><nombre>Juan</nombre></persona>";

    JAXBContext context = JAXBContext.newInstance(Persona.class);
    Unmarshaller unmarshaller = context.createUnmarshaller();
    Persona persona = (Persona) unmarshaller.unmarshal(new
StringReader(xml));

    System.out.println("Nombre: " + persona.getNombre());
    System.out.println("Edad: " + persona.getEdad());
} catch (JAXBException e) {
    e.printStackTrace();
}
```