Módulo 6

Acceso a datos

UF1. PERSISTENCIA EN FICHEROS

En esta Unidad Formativa se estudiará la persistencia en ficheros o la manera de preservar la información de un objeto de manera permanente, así como su recuperación para que pueda ser nuevamente utilizado.

1. Gestión de archivos

Objetivos:

- Emplear clases para la gestión de archivos y directorios.
- Acceder de varias formas y evaluar sus pros y sus contras.
- Emplear distintas operaciones básicas para acceder a archivos de acceso secuencial y aleatorio.
- Emplear clases para almacenamiento y recuperación de la información contenida en un archivo XML.
- Emplear clases para cambiar a otro formato la información almacenada en un archivo XML.
- Administrar excepciones.
- Serializar objetos Java a representaciones XML.

Contenidos:

- Clases asociadas a las operaciones de gestión de ficheros y directorios: creación, borrado, copia, movimiento, entre otros.
- Formas de acceso a un archivo.
- Clases para gestión de flujos de datos desde/hacia archivos.
- Trabajo con archivos XML: analizadores sintácticos (*parser*) y vinculación (*binding*).
- Excepciones: detección y tratamiento.

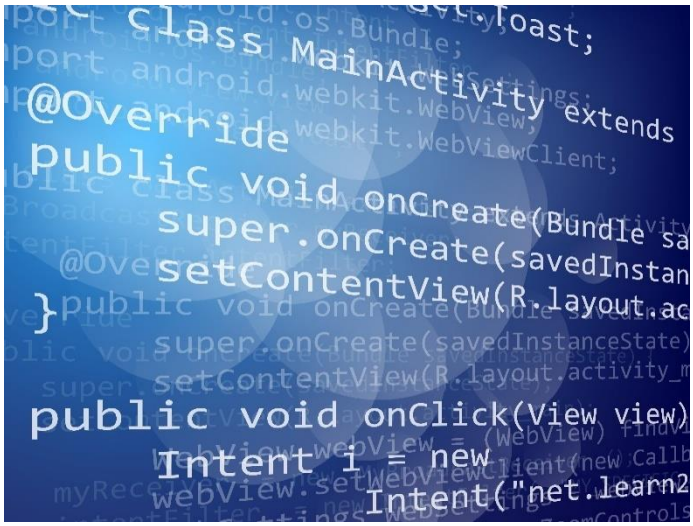
Un **archivo** es un grupo de bits que se almacenan en un dispositivo como, por ejemplo, un disco duro. Son identificados con un nombre y la descripción de la carpeta o directorio que lo contiene.

Uno de los pros de utilizar archivos es que los datos que se almacenan en dicho archivo se mantienen en el dispositivo, aunque se apague el ordenador, es decir, **no son volátiles**. Los archivos poseen un nombre y se sitúan en directorios o carpetas. El nombre en dicho directorio debe ser único, es decir, **no es posible que existan dos ficheros con el mismo nombre en una misma carpeta**.

Cada archivo tiene un tipo de **extensión** que suele ser de tres letras y que permite conocer qué tipo de fichero es como, por ejemplo, PDF, JPG, entre otros. Se muestran a continuación:

Un archivo está formado por un conjunto de líneas o registros y cada registro, a su vez, está formado por un conjunto de campos relacionados. La forma en la que se agrupan los datos en el archivo depende de la persona que lo cree.

En este tema se aprenderá a utilizar los archivos con el lenguaje **Java**.



1.1. Clases asociadas a las operaciones de gestión de ficheros y directorios: creación, borrado, copia, o movimiento, entre otros

Para manejar la **entrada/salida en Java** existe un paquete llamado ***java.io*** que contiene las clases necesarias, por lo que es imprescindible importar dicho paquete al comenzar a trabajar con archivos.

En primer lugar, la **clase File** permitirá obtener información acerca de los archivos, como es el caso de su nombre, sus distintos atributos o los directorios. Se puede mostrar el nombre de un archivo en concreto, un listado de varios archivos contenidos en una misma carpeta para crear otra nueva, o una ruta de directorios completa si no existiera.

Se pueden usar estos **tres tipos de constructores para la creación de un objeto File**:

- **File (String carpetaOArchivo)**:
  - En Windows: new File (“C:\\carpeta\\archivo.txt”).
  - En Linux: new (File(“/carpeta/archivo.txt”).

- **File (String carpeta, String nombreArchivo)**:
  - new File (“carpeta”, “archivo.txt”).

- **File (File carpeta, String archivo)**:
  - new File (new File (“carpeta”), “archivo.txt”).

Para definir la ruta absoluta en **Linux** se utiliza el prefijo **"/"**. En Microsoft **Windows** el prefijo utilizado para definir una ruta es una letra de unidad seguida de **":"** y, si es una ruta absoluta, va seguida de **"\\"**.

A continuación, se exponen algunos de los **métodos** de la clase File con los cuales podremos eliminar un fichero, conocer la ruta, saber si es un archivo válido o no, conocer el tamaño, y otros métodos bastante útiles para realizar operaciones con ficheros.

MÉTODO	DESCRIPCIÓN
boolean canRead()	Devuelve ***true*** si se puede leer el fichero.
boolean canWrite()	Devuelve ***true*** si se puede escribir en el fichero.
boolean createNewFile()	Crea el fichero asociado al objeto ***File***. Devuelve ***true*** si se ha podido crear. Para poder crearlo el fichero no debe existir. Lanza una excepción del tipo ***IOException.***
boolean delete()	Elimina el fichero o directorio. Si es un directorio debe estar vacío. Devuelve ***true*** si se ha podido eliminar.
boolean exists()	Devuelve ***true*** si el fichero o directorio existe.
String getName()	Devuelve el nombre del fichero o directorio.
String getAbsolutePath()	Devuelve la ruta absoluta asociada al objeto ***File.***

String getCanonicalPath()	Devuelve la ruta única absoluta asociada al objeto ***File***. Puede haber varias rutas absolutas asociadas a un ***File,*** pero solo una única ruta canónica. Lanza una excepción del tipo ***IOException.***
String getPath()	Devuelve la ruta con la que se creó el objeto ***File***. Puede ser relativa o no.
String getParent()	Devuelve un ***String*** conteniendo el directorio padre del ***File***. Devuelve ***null*** si no tiene directorio padre.
File getParent()	Devuelve un objeto ***File*** conteniendo el directorio padre del ***File***. Devuelve ***null*** si no tiene directorio padre.
boolean isAbsolute()	Devuelve ***true*** si es una ruta absoluta.
boolean isDirectory()	Devuelve ***true*** si es un directorio válido.
boolean isFile()	Devuelve ***true*** si es un fichero válido.
long lastModified()	Devuelve un valor en milisegundos que representa la última vez que se ha modificado (medido desde las 00:00:00 GMT, del 1 de enero de 1970). Devuelve 0 si el fichero no existe o ha ocurrido un error.
long length()	Devuelve el tamaño en bytes del fichero. Devuelve 0 si no existe. Devuelve un valor indeterminado si es un directorio.
String[] list()	Devuelve un array de ***String*** con el nombre de los archivos y directorios que contiene el directorio indicado en el objeto ***File***. Si no es un directorio devuelve ***null***. Si el directorio está vacío devuelve un array vacío.

String[] list(FilenameFilter filtro)	Similar al anterior. Devuelve un array de ***String*** con el nombre de los archivos y directorios que contiene el directorio indicado en el objeto ***File*** que cumplen con el filtro indicado.
boolean mkdir()	Crea el directorio. Devuelve ***true*** si se ha podido crear.
boolean mkdirs()	Crea el directorio incluyendo los directorios no existentes especificados en la ruta ***padre*** del directorio a crear. Devuelve ***true*** si se ha creado el directorio y los directorios no existentes de la ruta padre.
boolean renameTo(File dest)	Cambia el nombre del fichero por el indicado en el parámetro ***dest***. Devuelve ***true*** si se ha realizado el cambio.

Es posible consultar todos los **métodos en la API de Java**:

http://docs.oracle.com/javase/8/docs/api/java/io/File.html

EJEMPLO CÓDIGO:

```
public static File createFile() throws FileNotFoundException {
	File fichero = new File ("TestFolder", "test.txt");
	System.out.println(fichero.getName());
	System.out.println(fichero.getAbsolutePath());
	return fichero;

}
```

1.2. Formas de acceso a un archivo

Para acceder a la información de un archivo existen dos **maneras** diferentes: acceso secuencial y acceso directo o aleatorio.

- En el **acceso secuencial,** para acceder a un dato concreto del archivo, es necesario leer todo lo anterior, por lo que los datos o registros se leen y escriben de forma ordenada. Insertar algún dato entre los que ya están escritos no es posible, ya que se van escribiendo a partir del último dato escrito.

  Para el acceso secuencial en Java, la forma más común es en binario o a caracteres. En binario se utilizan las clases ***FileInputStream*** y ***FileOutputStream*** y para el acceso a caracteres las clases ***FileReader*** y ***FileWriter***.

EJEMPLO CÓDIGO ACCESO SECUENCIAL:

```
//Acceso secuencial
FileWriter fileW = new FileWriter(file);
fileW.write("Esto es un ejemplo");
fileW.flush();
fileW.close();
```

- En el **acceso directo o aleatorio**, al contrario que en el secuencial, se puede acceder a un dato o a un registro que se encuentre en cualquier posición del archivo sin necesidad de hacer una lectura completa del archivo hasta llegar a dicha posición. Los datos se almacenan en registros con un tamaño conocido, por lo que es posible moverse de un registro a otro para cambiarlos o leerlos.

  Para el acceso aleatorio se utiliza la clase **RandomAccessFile**.

EJEMPLO CÓDIGO ACCESO ALEATORIO:

```
//Acceso aleatorio
RandomAccessFile fichero = new RandomAccessFile(file, "rw");
fichero.writeBytes("Hola");

System.out.println(fichero.length());
```

Existen distintas operaciones que se llevan a cabo sin importar la forma de acceder a un archivo. Estas operaciones son: creación, apertura, cierre, lectura de datos y escritura de datos en el archivo y se explican a continuación:

- La **creación** del archivo se realiza asignando un nombre para poder acceder a él (como se explicó en un apartado anterior) y se realiza una única vez.
- Para poder manejar el archivo, previamente se tiene que utilizar algún programa que permita la **apertura** del mismo para poder manejarlo.
- Cuando no se vaya a utilizar el archivo será necesario **cerrarlo** y esta será la última instrucción del programa.
- La **lectura de datos** del archivo se realizará a través de alguna variable del programa y consistirá en trasladar información desde el archivo a la memoria principal para proceder a la lectura.
- En la **escritura** el proceso es a la inversa, se realiza desde la memoria al archivo.

Una vez abierto el archivo se suelen realizar las siguientes operaciones: altas, bajas, modificaciones y consultas.

- Las **altas** consisten en incluir un nuevo registro al archivo.
- En las **bajas** se elimina un registro ya existente en el archivo. Dicha eliminación puede ser lógica o física. En la lógica se cambia el valor de un campo del registro y en la física se elimina físicamente el registro del archivo.
- Las **modificaciones** consisten en cambiar parte de un registro. Se necesita localizar el registro que es necesario cambiar para, una vez localizado, proceder a reescribirlo.
- Las **consultas** consisten en la búsqueda de un registro en concreto.

Operaciones sobre archivos secuenciales

Para **realizar registros en archivos secuenciales** es preciso añadirlos de manera ordenada, es decir, cada uno seguido del último insertado. Para **añadir nuevos registros** se realizan a partir del último dato añadido en el final del archivo.

La **consulta** se realiza desde el comienzo del archivo, continuando la lectura después, hasta localizar el registro buscado. Para buscar, por ejemplo, el registro 35, será necesario buscar los 34 registros anteriores hasta llegar a él.

Las **altas en el archivo secuencial** se realizan tras el último registro añadido en el final del fichero.

Para **dar de baja un registro** hay que leer todos los registros anteriores al que se quiere dar de baja y pasarlos a un archivo auxiliar. Una vez reescritos se borra el archivo inicial y se renombra el nuevo archivo con el nombre del anterior.

Para **realizar modificaciones,** se debe localizar el registro que hay que cambiar, hacer dicha modificación y reescribir el archivo inicial en otro archivo auxiliar que contenga el nuevo registro. Es igual que el proceso para llevar a cabo bajas.

Los **archivos secuenciales** se suelen utilizar en distintas aplicaciones de procesos por lotes, como, por ejemplo, la copia de datos. Son **fáciles de usar** y una de las ventajas que tienen es el **rápido acceso a los siguientes registros** cuando se hace de forma secuencial. Los inconvenientes principales son que **no se puede acceder a un registro en concreto** y que, para actualizar el archivo, hay que crear uno nuevo con toda la información.

Operaciones sobre archivos aleatorios

En las operaciones con archivos aleatorios hay que tener en cuenta que para acceder a un registro se debe localizar primero la dirección o posición en la que se encuentra.

Recuerda que los archivos de acceso aleatorio utilizan rutas relativas en lugar de rutas absolutas.

Cuando se utilizan este tipo de archivos, para localizar un registro se tendrá en cuenta el tamaño del registro y la clave (identificador único de un registro).

Ejemplo: Hay un archivo de películas con tres campos: código, nombre y género. El campo código se utiliza como clave dando el valor 1 para el primer registro, 2 para el segundo, y así sucesivamente. Para localizar una película habrá que usar el código X accediendo a la posición con tamaño *(X-1).

En ocasiones, **una posición podría estar ocupada por otro registro**, por lo que sería necesario buscar alguna posición libre en el archivo, o usar una zona de excedentes en el mismo archivo para ubicar dichos registros.

Las **consultas** se realizan consultando la clave del registro y aplicando la función de conversión, obteniendo de esta manera la dirección y leyendo el registro situado en esa posición.

En las **altas**, para añadir un nuevo registro hay que conocer su clave, aplicar la función de conversión para obtener así la dirección y escribir el registro en dicha posición. Si estuviera ocupada por otro registro, se incluiría en la zona de excedentes.

Para las **bajas** se lleva a cabo la acción de forma lógica, utilizando para ello un *switch* en el que el valor 1 sea para cuando dicho registro exista y el valor 0 para dar de baja ese registro. Físicamente no se borra el registro, solo se cambia el valor de su campo clave a 0.

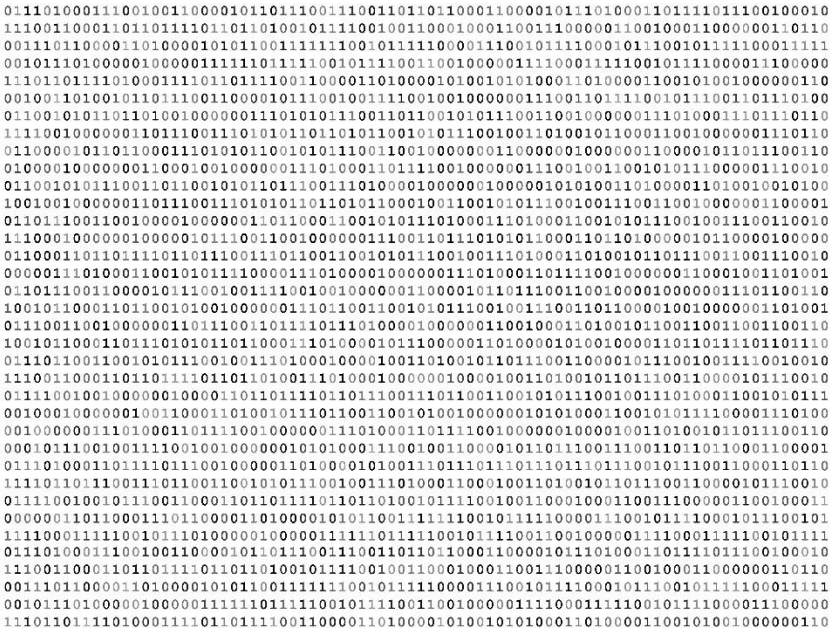
Para **modificar** el registro hay que localizarlo, conocer su clave para aplicar la función de conversión obteniendo así la dirección, cambiar la información y volver a escribir en esa posición.

Una de las **ventajas** de estos tipos de archivos es poder acceder de forma rápida a la lectura y escritura de un registro. Algunos **inconvenientes** pueden ser que hay que establecer relaciones entre la posición del registro y su contenido, y que, en ocasiones, se desaprovecha el espacio ya que se quedan huecos libres entre registros.

1.3. Clases para gestión de flujos de datos desde/hacia archivos

En Java se pueden usar dos tipos de archivos, de texto o binarios y para acceder a ellos es posible hacerlo de forma secuencial o aleatoria.

Estos **archivos de texto** están compuestos por caracteres legibles, mientras que los **archivos de tipo binario** se pueden almacenar con cualquier tipo de dato (*int*, *float*, *boolean*, *etc*.).



Archivos de texto

En los archivos de texto generados por un editor se suele utilizar de forma genérica un tipo de formato como **ASCII, UNICODE o UTF8** que sirven para almacenar los caracteres.

Al trabajar con estos formatos se utilizan las clases **FileReader** para leer estos caracteres y **FileWriter** para escribirlos. Para trabajar con estos archivos y poder leerlos o escribir con ellos, se hará siempre dentro de la excepción **try-catch**.

Si se utiliza la clase **FileReader,** se puede generar la excepción ***FileNotFoundException*** (nombre del fichero no existe o no es válido), y si se usa la clase **FileWriter,** la excepción ***IOException*** (disco lleno o protegido contra escritura).

Los **métodos** de la clase **FileReader** para realizar la **lectura** son:

MÉTODO	DESCRIPCIÓN
FileReader.abort()	Interrumpe la operación de lectura. A su regreso ***readyState*** será ***DONE.***
FileReader.readAsArrayBuffer()	Comienza la lectura del contenido del objeto ***Blob*** especificado. Una vez terminada, el atributo ***result*** contiene un ***ArrayBuffer*** representando los datos del fichero.
FileReader.readAsBinaryString()	Comienza la lectura del contenido del objeto ***Blob.*** Una vez terminada, el atributo ***result*** contiene los datos binarios en bruto del archivo como una cadena.
FileReader.readAsDataURL()	Comienza la lectura del contenido del objeto ***Blob.*** Una vez terminada, el atributo ***result*** contiene un data: ***URL*** que representa los datos del fichero.
FileReader.readAsText()	Comienza la lectura del contenido del objeto **Blob.** Una vez terminada, el atributo ***result*** contiene el contenido del fichero como una cadena de texto.

Para **leer un archivo en Java** se siguen los siguientes pasos:

1. Se crea o abre un fichero mediante la clase **File**.
2. Se abre un flujo de entrada con la clase **FileReader**.
3. Se realizan las operaciones de escritura o lectura y al finalizar se cierra con el método **close()**.

Un **ejemplo** de lectura de un archivo en Java:

EJEMPLO CÓDIGO :

```
public static void leerFichero(String fichero) throws FileNotFoundException, IOException {
        String linea;
        FileReader fileR = new FileReader(fichero);
        BufferedReader buffer = new BufferedReader(fileR);
        while((linea = buffer.readLine())!=null) {
            System.out.println(linea);
        }
        buffer.close();
    }
```

La clase ***IOException*** también puede ser lanzada por estos métodos. Al escribir todos los caracteres uno a uno se obtiene un ***String*** que se convierte en un *array* de caracteres. Para no tener que escribirlos uno a uno también se permite usar ***fic.write(cad)***.

- Es importante saber que, si se escriben caracteres sobre un archivo ya existente, toda la información almacenada en él se borrará, por lo que para usar la clase **FileWriter** se debe colocar al final del método, en el segundo parámetro, el valor true:

CÓDIGO :

```
FileWriter arc = new FileWriter(archivo, true);
```

- Si se quiere usar la clase **FileReader** para leer líneas completas no será posible, ya que no tiene métodos capaces para ello, por lo que se debe utilizar la clase **BufferedReader** que dispone del método ***readLine()***.
  Este método es capaz de leer una línea del archivo y devolverla, o devolver *null* en el caso de que no haya nada que leer o si se llega al final del archivo.

CÓDIGO :

```
BufferedReader archivo = new BufferedReader (new FileReader(Nombrearchivo));
```

- La clase **BufferedWriter** es derivada de la clase **Writer** y lo que realiza es una escritura eficiente de caracteres añadiéndole un *buffer*. Para construir un **BufferedWriter** será necesaria la clase **FileWriter**:

CÓDIGO :

```
BufferedWriter archivo = new BufferedWriter = (new FileWriter(nombreArchivo));
```

- De **Writer** también deriva la clase **PrintWriter** que posee los métodos ***print(String)*** y ***println(String)*** (iguales a los *System.out*) que sirven para escribir en un archivo. Estos métodos reciben un *String* y lo escriben en un archivo. En el caso del segundo método, además realiza un salto de línea. Para usar la clase **PrintWriter** se necesita la clase **FileWriter**:

CÓDIGO :

```
PrintWriter archivo = new PrintWriter(new FileWriter (NombreArchivo));
```

Existen varios métodos básicos para realizar las escrituras con el método **FileWriter**:

https://docs.oracle.com/javase/7/docs/api/java/io/FileWriter.html

Archivos binarios

Al contrario que con los archivos de texto, en aquellos casos en los que se quiere almacenar secuencias de dígitos binarios que son ilegibles, se llevarán a cabo a través de archivos binarios.

Las **dos clases** que se pueden usar para realizar el trabajo con archivos binarios en **Java** son:

1. ***FileInputStream*** (para entrada).
2. ***FileOutputStream*** (para salida).

Estas dos clases trabajan con flujos de bytes y crean un enlace entre el flujo de bytes y el archivo.

Los métodos proporcionados para la clase ***FileInputStream*** son iguales que los de la clase ***FileReader*****.** Devuelven el número de bytes leídos o devuelven -1 si se llega al final del archivo:

MODIFICADOR y TIPO	MÉTODO	DESCRIPCIÓN
int	available()	Devuelve una estimación del número de bytes restantes que pueden ser leídos (u omitidos) de este flujo de entrada sin bloquear por la siguiente invocación de un método para este flujo de entrada.
void	close()	Cierra este flujo de entrada de archivos y libera todos los recursos del sistema asociados con el flujo.
protected void	finalize()	Asegura que el método de cierre de este flujo de entrada de archivo se llama cuando no hay más referencias a él.

FileChannel	getChannel()	Devuelve el objeto *FileChannel* único asociado con este flujo de entrada de archivos.
FileDescriptor	getFD()	Devuelve el objeto *FileDescriptor* que representa la conexión al archivo real en el sistema de archivos utilizados por este *FileInputStream.*
int	read()	Lee un byte de datos de este flujo de entrada.
int	read(byte[]b)	Lee hasta *b.length* bytes de datos de este flujo de entrada en una matriz de bytes.
int	read(byte[]b, int off, int len)	Lee hasta *len* bytes de datos de este flujo de entrada en una matriz de bytes.
int	skip(long n)	Salta y descarta *n* bytes de datos del flujo de entrada.

Los métodos de la clase ***FileOutputStream*** para escritura son:

MODIFICADOR y TIPO	MÉTODO	DESCRIPCIÓN
void	close()	Cierra este flujo de salida de archivos y libera todos los recursos del sistema asociados con el flujo.
protected void	finalize()	Asegura que el método de cierre de este flujo de salida de archivo se llama cuando no hay más referencias a él.

FileChannel	getChannel()	Devuelve el objeto *FileChannel* único asociado con este flujo de salida de archivos.
FileDescriptor	getFD()	Devuelve el descriptor de archivo asociado con esta secuencia.
void	write(byte[] b)	Escribe *b.length* bytes de la matriz de bytes especificada en este flujo de salida de archivos.
void	write(byte[] b, int off, int len)	Escribe *len* bytes de la matriz de bytes especificada empezando en el desplazamiento fuera de este flujo de salida del archivo.
void	write(int b)	Escribe el byte especificado en este flujo de salida.

Para escribir bytes en el final del archivo se utiliza ***FileOutputStream***, situando en el segundo parámetro el valor *true*.

CÓDIGO :

```
FileOutputStream fileout = new FileOutputStream(fichero,true);
```

Si lo que se pretende es escribir datos de tipo primitivo: *int*, *float*, *long*, etc., se deben usar las clases ***DataInputStream*** y ***DataOutputStream*** que definen varios métodos *readXXX* y *writeXXX*.

Dichos métodos son variantes de los métodos ***read()*** y ***write()*** de la clase base, capaz de leer y escribir datos de tipo primitivo.

Para abrir un objeto con **DataInputStream** o **DataOutputStream** se utilizan los mismos métodos que para **FileInputStream** y **FileOutputStream,** respectivamente.

Se pueden ver **métodos** relacionados con estas clases en :

https://docs.oracle.com/javase/8/docs/api/java/io/DataInputStream.html

https://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html

Objetos en ficheros binarios

Como ya se ha visto anteriormente, es posible guardar datos de tipo primitivo en forma binaria, pero si existe un objeto con varios atributos, al almacenarlo en un archivo, sería fundamental guardar cada objeto por separado volviéndose así pesado si se cuenta con mucha cantidad de objetos.

Para solucionarlo, Java permite guardar objetos en archivos binarios a través de la **interfaz Serializable**, en la que se disponen de varios métodos para leer y guardar archivos binarios. Algunos de los más importantes son:

- **Object readObject()**: utilizado para leer objetos de **ObjectInputStream**. También puede lanzar las excepciones ***IOException*** y ***ClassNotFoundException***.

- **Void writeObject(Object obj)**: utilizado para escribir en el objeto indicado en el *ObjectOutputStream*. También puede lanzar la excepción ***IOException***.

En Java, a través de la interfaz *Serializable*, es posible coger cualquier objeto y convertirlo en una secuencia de bits y después ser regenerado al objeto original.

Para leer y escribir objetos serializables a un *stream* es posible utilizar las clases **ObjetctInputStream** y **ObjectOutputStream** respectivamente.

Problema que puede surgir con los archivos de objetos

Cuando se crea un archivo de objetos se inserta información en una cabecera inicial y después se añaden los objetos. Si se utiliza el archivo para añadir más registros, se realiza una nueva cabecera y se añaden los objetos a partir de esta. Existe un problema cuando se lee el archivo y se encuentra con la segunda cabecera y aparece la excepción ***StreamCorruptedException.*** En este caso no permitirá leer más objetos.

La cabecera se crea al poner *new ObjectOutputStream(archivo)*. Lo que se debe hacer para que no se añadan esas cabeceras es **redefinir la clase *ObjectOutputStream* estableciendo una nueva clase que la herede (extends)**.

Dentro de esa clase se redefine el método ***writeStreamHeader***() que es el encargado de escribir las cabeceras.

De este modo, el método se anula no haciendo nada, de manera que si el archivo ya está creado se denominará a ese método de la clase redefinida.

La **clase redefinida** sería, por **ejemplo**:

CÓDIGO :

```
Public MiObjectOutputStream(OutputStream out) throws IOException
{    super(out);    }
protected MiObjectOutputStream()
                    Throws IOException, SecurityException
{      super(out)    }
//Redefinición del método de escribir la cabecera para que no haga nada
protected void writeStreamHeader() throws IOException {}
```

Archivos de acceso aleatorio

Todo lo visto anteriormente está basado en archivos de forma secuencial en los que se comienza desde el inicio del archivo y se continúa leyendo una línea a continuación de la otra.

Para los archivos de acceso aleatorio (comentados anteriormente), Java dispone de la clase **RandomAccessFile** que permite acceder al archivo binario de forma aleatoria a través de sus métodos. No pertenece a la jerarquía **InputStream**/**OutputStream** ya que permite avanzar y retroceder en el archivo.

Existen **dos constructores** para la creación del archivo de acceso aleatorio y ambos pueden lanzar la excepción ***FileNotFoundException***:

- **RandomAccessFile (String nombrearchivo, String modoAcceso)**: se escribe el nombre del archivo y se incluye también el *path*.
- **RandomAccessFile (File objetoFile, String modoAcceso)**: se asocia un objeto File a un archivo.

El argumento ***modoAcceso*** tiene dos **valores**:

- ***r*** (abre el fichero en modo lectura. El archivo debe existir. Operación de escritura lanzará ***IOException***).
- ***rw*** (abre el archivo en modo lectura y escritura. Si no existe el archivo se crea).

Una vez se abre el archivo se pueden usar los métodos readXXX y writeXXX de las clases **DataInputStream** y **DataOutputStream**.

En la clase **RandomAccessFile** existe un puntero que indicará la posición en el archivo. Cuando se crea el archivo se encontrará en 0 en el principio del mismo y cuando se denomina a los métodos **read()** y **write()** ajustarán el puntero según la cantidad de bytes leídos o escritos.

1.4. Trabajo con archivos XML: analizadores sintácticos (*parser*) y vinculación (*binding*)

XML (*eXtensible Markup Language* o Lenguaje de Etiquetado Extensible) es un metalenguaje o, lo que es lo mismo, un lenguaje para definir lenguajes de marcado.

La información dentro de los documentos se realiza de forma **jerarquizada y estructurada** definiendo los contenidos dentro del mismo. Los archivos XML son archivos de texto en lenguaje XML donde se organiza la información de forma secuencial y jerarquizada.

Son características de este lenguaje las marcas que estructuran el documento, como son el símbolo menor que, (***<)*** y mayor que, (***>)*** que delimitan las marcas para estructurarlo. Cada una de ellas tendrá un nombre y podrá tener 0 o más atributos.

Los **archivos XML** se pueden utilizar para distintas **funciones**:

- Proporcionar datos en una base de datos.
- Almacenar copias de esas bases de datos.
- Escribir archivos de configuración de programas.
- Efectuar comandos en servidores remotos en el protocolo SOAP.

Para la **lectura** de este tipo de documentos XML se utilizan **procesadores de XML o *parser***, los cuales permiten acceder a su contenido y estructura. Son independientes del lenguaje utilizado y algunos de los más empleados son: **DOM** (Modelo de Objetos

de Documento) o **SAX** (API Simple para XML). Ambos tienen la misma función, pero la ejecutan de diferente manera. Algunas de sus **características** son:

DOM	SAX
Almacena la estructura del documento en memoria en forma de árbol y se recorre los diferentes nodos analizando a qué tipo particular pertenecen. Origen en el W3C.	Se lee el fichero de forma secuencial produciendo una secuencia de eventos, (inicio y final de un documento o etiqueta) en función del resultado de la lectura.
Más consumo de memoria.	Menos consumo de memoria.
Permite ver visión global del documento.	No permite ver de forma global el documento.

Acceso a archivos XML con DOM

Este procesador de XML en Java necesita de paquetes que incluyan las clases necesarias para trabajar con él, por lo que será necesario el paquete **org.w3c.dom** (contenido en el JSDK) y el paquete **javax.xml.parsers**.

Ambos contienen dos clases fundamentales como son: **DocumentBuilderFactory** y **DocumentBuilder**.

Desde el mismo procesador DOM no se define ningún método para generar un archivo XML a partir de un árbol DOM, por lo que se usará el paquete ***javax.xml.transform*** que permitirá especificar una fuente y un resultado como pueden ser archivos, flujos de datos, o nodos DOM.

Cuando se utilice DOM en Java son necesarias algunas interfaces como: **Document** (crea nuevos nodos), **Element** (expone propiedades y métodos con los cuales se pueden manipular elementos del documento y sus atributos), **Node** (representa cualquier nodo), **Nodelist** (lista con nodos hijos de un nodo), **Attr** (accede a atributos de un nodo), **Text** (datos carácter de un elemento), **CharacterData** (proporciona atributos o métodos para manipular datos de caracteres) o **DocumentType** (información contenida en la etiqueta **<!DOCTYPE>).**

PAQUETES NECESARIOS AL CREAR UN ARCHIVO XML CON DOM:

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

Acceso a archivos XML con SAX

El segundo procesador XML que se puede utilizar es **SAX** (API Simple para XML) el cual es una herramienta bastante útil, ya que posee varias clases e interfaces, además de consumir menos memoria al no cargar todo el documento. Por el contrario, es más complejo de programar en él. API escrito totalmente en Java y está incluida en el JRE, el cual permite crear nuestro propio parser de XML.

Al leer un archivo XML se producen eventos que ocasionan la llamada a métodos. Los eventos no son más que encontrar la etiqueta de inicio y fin del documento (**startDocument()** y **endDocument()**), de un elemento (**startElement()** y **endElement()**), los caracteres (**characters()**), etc.

CLASES NECESARIAS PARA IMPORTAR AL CREAR UN ARCHIVO XML EN SAX:

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

Se crea un objeto **XMLReader** que puede producir una excepción ***SAXException*** que es necesario capturar.

CÓDIGO:

```
XMLReader procesadorXML = XMLReaderFactory.createXMLReader();
```

Después, se deben indicar los tipos de objetos que tienen los métodos: **ContentHandler** (recibe notificaciones de los eventos), **DTDHandler** (recoge eventos relacionados con la DTD), **ErrorHandler** (métodos de tratamiento de errores), **EntityResolver** (se llama siempre que haga referencia a una unidad), **DefaultHandler** (implementación por defecto para todos sus métodos).

El tratamiento de la información se realiza a través de objetos como pueden ser:

setContentHadler(), **setDTDHandler()**, **setEntityResolver()** y **setErrorHandler()**

A continuación, se define el fichero que se quiere tratar a través de un objeto ***InputSource***. Para finalizar se procesa el documento a través del método ***parse()*** al que se le pasa un objeto **InputSource.**

Serialización de objetos a XML

Para serializar objetos a XML y viceversa se debe utilizar la librería **XStream**. Para poder darle uso hay que **descargar los JAR desde la web:**

http://x-streeam.github.io/download.html

Además, es necesario descargar el archivo ***xstream-distribution-1.4.8-bin.zip*** que habrá que descomprimir y buscar ***xstream-1.4.8.jar y kxml2-2.3.0.jar***. Para añadir estos archivos a un proyecto se hace de la siguiente forma; si se supone que se encuentran en el directorio ***C:\ejer1\xstream***, el **CLASSPATH** quedaría:

SET CLASSPATH = .;C:\ejer1\xstream\kmxl2-2.3.0.jar;C:\ejer1\xstream\xstream-1.4.8.jar

Para empezar a utilizar la clase **XStream** hay que crear una instancia de esa clase:

CÓDIGO:

```
XStream instancia = new XStream();
```

Estas etiquetas XML habitualmente se suelen corresponder con los nombres de los atributos con sus clases, pero es posible cambiarlas usando el método **alias (String alias, Class clase)**.

Existe otro método para poner un alias al nombre de campo, **alias (String alias, Class clase, String nombrecampo)**.

Para finalizar, se genera el archivo a través del método **toXML (Object objeto, OutputStream out)**:

CÓDIGO:

```
instancia.toXML(listaper, new FileOutputStream("archivo.xml"));
```

Para leer el archivo XML se deben usar los métodos **alias()** y **addImplicitCollection()** ya que se utilizaron también para la escritura. En el caso de deserializar el objeto a partir del archivo, se utiliza el método **fromXML (inputStream input)** que devuelve un tipo **Object**:

CÓDIGO:

```
Clase nombreclase = (Clase) instancia.fromXML (new FileInputStream("archivo.xml"));
```

Conversión de archivos XML a otro formato

Para expresar hojas de estilo en este lenguaje XML se usan unas recomendaciones a través de **XSL (*Extensible Stylesheet Language*)** que describen un proceso de presentación a través de un conjunto de elementos en XML. Contienen tanto reglas de construcción como reglas de estilo.

Si se quiere aplicar la hoja de estilos XSL al fichero XML hay que obtener un objeto **Transformer** creando la instancia **TransformerFactory** y aplicando el método **newTransformer(Source source)**.

CÓDIGO:

```
Transformer transformer = TransformerFactory.newInstance().
newTransformer(archivoestilo);
```

Para conseguir la transformación es necesario llamar al método ***transform(Source fuenteXml, Result resultado)*** y pasar los datos (archivo XML) y el *stream* de salida (archivo HTML).

CÓDIGO:

```
Transformer.transform(archivoXML, result);
```

1.5. Excepciones: detección y tratamiento

Una **excepción** es un evento que detiene el flujo normal de secuencias y retorna la información contenida a través del gestor de excepciones y detiene la aplicación.

Cuando existe un error dentro de un método Java, este crea un objeto **Exception** y lo maneja fuera. Está diseñado para cuando no es capaz de manejar información y lanza un error.

Las excepciones en Java son objetos que derivan de la clase **Exception** y que, a su vez, derivan de la clase ***Throwable***.

Capturar excepciones

Para capturar excepciones se utiliza el bloque llamado **try-catch**. En el primer bloque *try* se incluye el código que podrá generar una excepción y seguidamente se crean tantos bloques *catch* como se deseen, indicando el tipo de excepción que podrá generar en cada uno de ellos. Se puede finalizar, si es preciso, con un bloque **finally** que se ejecutará al terminar el bloque *try* o *catch*.

Al crear los bloques *catch* se puede generar una excepción general con la clase ***Exception***, pero será necesario ponerla al final para que las demás excepciones no queden invalidadas.

Algunos **métodos** de la clase ***Throwable*** son:

MÉTODO	DESCRIPCIÓN
getMessage()	Se usa para obtener un mensaje de error asociado con una excepción.
printStackTrace()	Se utiliza para imprimir el registro del *stack* donde se ha iniciado la excepción.
toString()	Se utiliza para mostrar el nombre de una excepción junto con el mensaje que devuelve ***getMessage***().
Void printStackTrace (PrintStream) o (PrintWriter)	Visualiza el objeto y la traza de pila de llamadas lanzada.

Asimismo, se puede incluir dentro de una sentencia *try* otra sentencia *try*. Si esa sentencia no tuviera bloque *catch* se buscaría el bloque *catch* más externo.

Especificar excepciones

Si existe algún método que decide no gestionar una excepción a través del bloque *try-catch*, habrá que especificar a través de la palabra clave ***throws*** la lista de excepciones que se pueden dar.

En la **API de Java** se pueden ver los tipos de **excepciones** que se pueden producir:

http://docs.oracle.com/javase/8/docs/api/

UF2. PERSISTENCIA EN BDR-BDOR-BDOO

En esta Unidad Formativa se muestra una visión global sobre cómo las aplicaciones orientadas a objetos usan los Sistemas Gestores de Bases de Datos (SGBD) para conseguir la persistencia de sus instancias.

1. Gestión de conectores

En este primer apartado de la unidad se dan a conocer los distintos conectores que existen para enlazar un código con las bases de datos, el desarrollo de aplicaciones para el acceso a base de datos (BD), la ejecución de procedimientos y la creación de informes con los datos almacenados.

Contenido:

- El desfase objeto-relacional.
- Protocolos de acceso a bases de datos. Conectores.
- Ejecución de sentencias de descripción de datos.
- Ejecución de sentencias de modificación de datos.
- Ejecución de consultas.

1.1. El desfase objeto-relacional

En la actualidad, las bases de datos relacionales están dejando paso a las bases de datos orientadas a objetos, ya que poseen unos tratamientos de datos más complejos y solucionan los problemas de aplicaciones más sofisticadas como son las de diseño de ingeniería o experimentos científicos, entre otras.

Dentro de estas aplicaciones de la **programación orientada a objetos (POO)** los elementos importantes son los **objetos**.

Las bases de datos orientadas a objetos son más complejas a la hora de programar y guardar los datos, dando lugar a más código. Esto es debido a que existe una diferencia de esquemas entre los elementos que se almacenan (objetos) y las características del lugar en el que se almacenan (tablas).

El concepto de desfase objeto-relacional es la **diferencia existente entre las bases de datos relacionales y las bases de datos orientadas a objetos**, ya que la forma de construir y de guardar los datos en una y en otra se realiza de forma distinta.

Este desfase objeto-relacional surge porque el modelo de base de datos relacional trabaja con relaciones y conjuntos, mientras que el modelo de base de datos orientada a objetos trabaja, como bien indica su nombre, con objetos y con las relaciones que existen entre ellos. El conjunto de las dificultades encontradas al usar un programa escrito en programación orientada a objetos, junto con una base de datos relacional, es lo que provoca que se produzca este desfase objeto-relacional.

Ambos tipos de bases de datos (relacional y orientada a objetos) pueden compaginarse realizando un mapeo de las estructuras para que puedan extraerse y almacenarse los datos.

1.2. Protocolos de acceso a bases de datos. Conectores

Los protocolos de acceso a bases de datos facilitan el acceso a la base de datos ocultando los detalles específicos de cada una. De esta forma, el programador solo debe tener en cuenta los aspectos que intervienen en su aplicación.

Dos de los protocolos más extendidos para conectar las bases de datos SQL son **ODBC** y **JDBC**.

- **ODBC** (*Open Database Connectivity)*: Es una especificación de Microsoft que tiene una interfaz en C y que permite el acceso desde cualquier aplicación.
- **JDBC** (*Java Database Connectivity): Ofrece conectividad de base de datos con aplicaciones Java* y define una API para conectarse principalmente a bases de datos de tipo relacional.

Al conjunto de las clases que nos permiten esta conexión entre nuestra aplicación y la base de datos se le denomina conector.

Existen algunos orígenes de datos que no son bases de datos, como pueden ser los **ficheros planos y los correos electrónicos**. Para ellos existe una norma llamada **OLE-DB** (***Object Linking and Embedding for Databases***). OLE-DB Es una API de Microsoft escrita en lenguaje C++ para orígenes de datos que no provienen de bases de datos. Proporciona estructuras para la conexión con los orígenes de datos, ejecución de comandos y la devolución de resultados.

En la norma OLE-DB se pueden averiguar los orígenes de datos para saber las interfaces que soportan, mientras que, por el contrario, en la ODBC los comandos siempre se encuentran en SQL. La norma OLE-DB puede estar en cualquier lenguaje.

La API de **Microsof ADO (*Active Data Objects*)** ofrece una interfaz simple a la hora de utilizarla y con funcionalidad OLE-DB.

Acceso a datos mediante ODBC

Como se ha explicado anteriormente, **ODBC** es una **norma que permite conectarse a una base de datos desde cualquier aplicación**, sin tener en cuenta qué tipo de gestor de BD se está utilizando.

Cada sistema de BD que sea compatible con ODBC facilita una biblioteca que se debe enlazar con el programa cliente. Este programa se comunica con el servidor para realizar la acción solicitada y así poder obtener resultados.

Pasos para usar **ODBC**:

1. **Configurar la interfaz ODBC**. En el programa se asignará un entorno SQL a través de la función **SQLAllocHandle()**. Con ello, se pretende que, con este manejador, la base de datos traduzca la información de la consulta en comandos que logre entender. Existen varios **tipos de manejadores**:

    - **SQLHENV**: define el entorno para acceder a los datos.
    - **SQLHDBC**: identifica la conexión y el estado.
    - **SQLHSTMT**: declaración SQL y cualquier conjunto de resultados asociados.
    - **SQLHDESC:** agrupación de metadatos que describen una sentencia SQL.

2. Proceder a **abrir la conexión** usando **SQLDriverConnect()** o **SQLConnect()**.

3. Una vez conectados con el programa se podrán **enviar datos SQL** a la BD utilizando **SQLExecDirect()**.

4. Cuando se **finaliza la sesión**, se desconectará de la base de datos y liberará la conexión junto con los manejadores SQL.

ODBC es una norma de compleja utilización ya que realiza una mezcla entre características simples y avanzadas y usa opciones muy elaboradas para hacer una consulta simple. Algunas de sus principales **funciones** son:

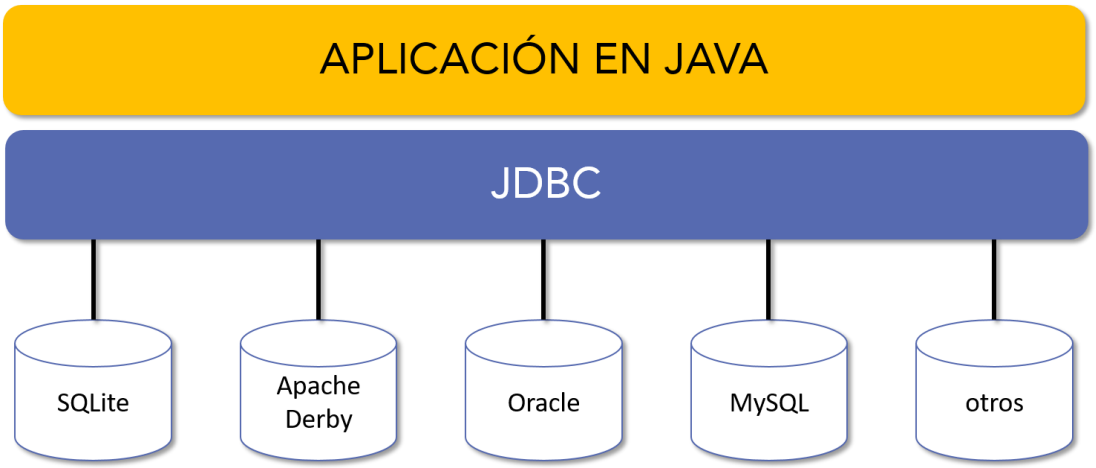
- **SQLAllocHandle, SQLDriverConnect**: establecen conexión con la BD.
- **SQLAllocStmt, SQLExecDirect**: para realizar sentencias SQL.
- **SQLFetch, SQLGetData, SQLNumResultCols**, **SQLRowCount**: se obtendrán datos de consultas SQL.
- **SQLDisconnect, SQLFreeHandle**: con estas funciones se pueden realizar operaciones para limpiar memoria y desconectar la BD.

Para acceder al programa mediante ODBC se debe hacer mediante una base de datos a través de un usuario y contraseña.

Acceso a datos mediante JDBC

JDBC no es solo una librería con la que poder acceder a los datos, sino que también **define una arquitectura estándar en la que los fabricantes pueden crear sus drivers para que las aplicaciones Java puedan acceder a los datos**. JDBC tiene una intefaz que es distinta para cada BD. Es lo que se denomina ***driver*** (conector).

La función de este driver es que los métodos de las clases JDBC se puedan corresponder con el API de la BD.



A través de **JDBC** se pueden realizar las siguientes **tareas**:

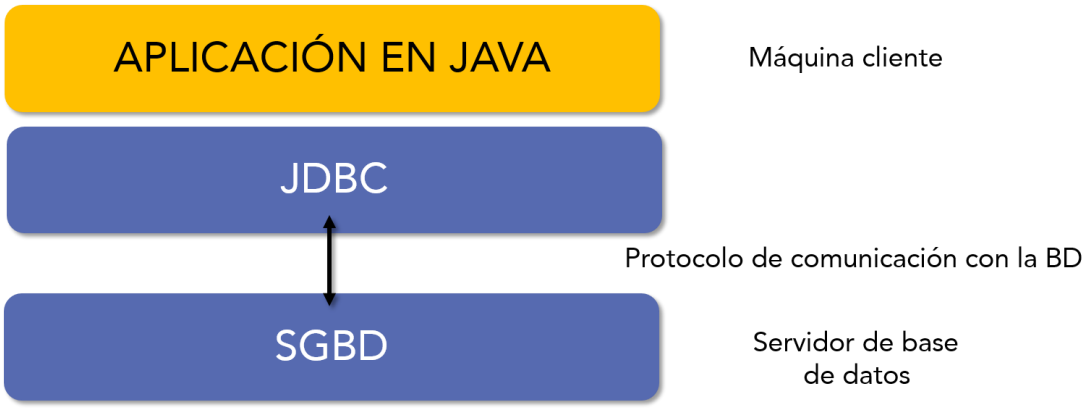
- Conectar con una base de datos.
- Realizar consultas e instrucciones para actualizar la BD.
- Recuperar y procesar los resultados de la BD.

Dos modelos de acceso a datos

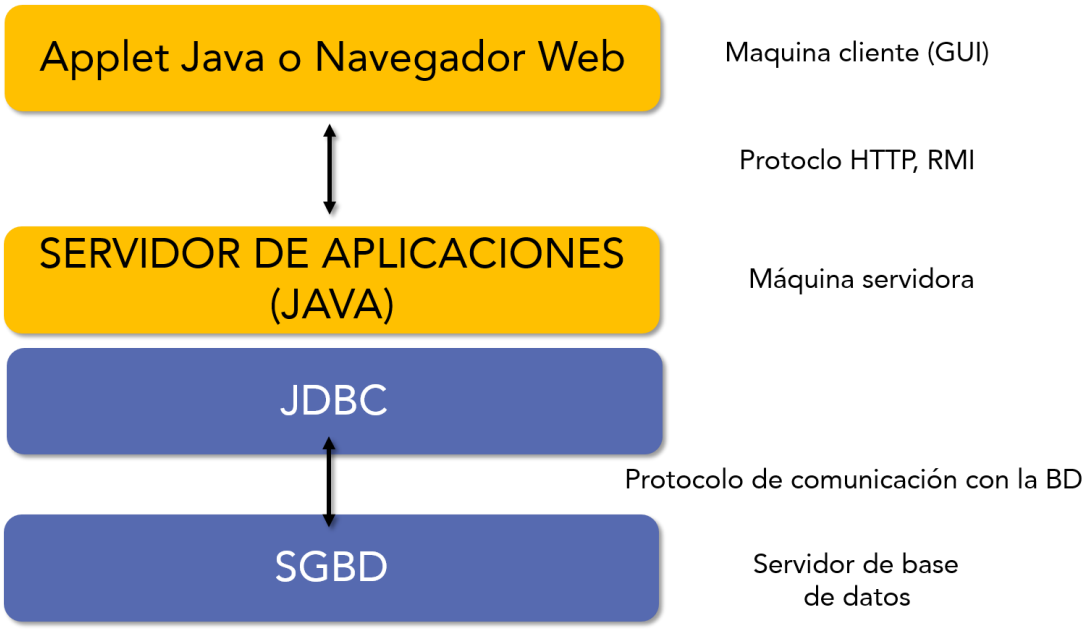
La API **JDBC** proporciona dos modelos con los que se puede acceder a las bases de datos:

1. **Modelo de dos capas**: se caracteriza porque la aplicación Java y la base de datos se comunican entre sí directamente. La aplicación se encarga de enviar las consultas SQL a la base de datos que las procesa y envía un resultado al programa.

   Para ello es necesario un driver JDBC que resida en el mismo sitio que la aplicación y las solicitudes se realizan a través de la red cliente-servidor.



2. **Modelo de tres capas**: a diferencia del modelo de dos capas, posee una capa intermedia la cual es la encargada de recibir las consultas SQL y de recibir los resultados. El cliente y el driver pueden estar en distintas máquinas.



Es posible definir un **servidor de aplicaciones** como la aplicación en **J2EE (*Java 2 Platform Enterprise Edition*)** que sirve para desarrollar, construir y desplegar aplicaciones empresariales multicapa basadas en la Web.

Se pueden utilizar diferentes aplicaciones como: *BEA WebLogic, IBM WebSphere, Oracle IAS o Borland AppServer.*

Tipos de drivers

Para la norma **JDBC** existen **cuatro** tipos de drivers:

Nombre	Descripción	Función	Requisitos
1. JDBC-ODBC Bridge	Facilita el acceso a una BD JDBC por medio de un driver ODBC.	Transforma las llamadas al API de JDBC en llamadas ODBC.	Obliga a tener instalado y configurado ODBC en la máquina cliente.
2. Native	Es un tipo de controlador que posee una parte escrita en Java y otra en lenguaje nativo de la BD.	Traduce llamadas de la API de JDBC en llamadas del motor de base de datos.	Escribe en la máquina cliente código binario propio del cliente de BD y del SO.
3. Network	Controlador de Java que utiliza entorno de red para comunicarse con la BD.	Traduce primero las llamadas al API de JDBC en llamadas al protocolo de red para luego ser traducida de nuevo al protocolo de la BD.	No exige la instalación en el cliente.
4. Thin	Controlador de Java puro con protocolo nativo.	Traduce las llamadas al API de JDBC en llamadas del protocolo de red utilizando el motor de BD.	No exige la instalación en el cliente.

Para acceder a las bases de datos, las opciones más adecuadas son la tercera y cuarta, ya que esta última es la más utilizada. Las dos primeras opciones serán usadas en caso de ser necesario, por si las anteriores no hacen el efecto esperado.

Cómo funciona JDBC

Para trabajar con JDBC es necesario la utilización de interfaces que permitan trabajar con las bases de datos y a raíz de ellas se dará origen a las clases que se usen.

Estarán especificadas en el paquete ***java.sql.*** Las más importantes son:

CLASE E INTERFAZ	DESCRIPCIÓN
Driver	Permite conectarse a una base de datos: cada gestor de base de datos requiere un driver distinto.
Drivermanager	Permite gestionar todos los drivers instalados en el sistema.
DriverPropertyInfo	Proporciona diversa información acerca de un driver.
Connection	Representa una conexión con una base de datos. Una aplicación puede tener más de una conexión.
DatabaseMetadata	Proporciona información acerca de una base de datos: tablas que contiene, etc.
Statement	Permite ejecutar sentencias SQL sin parámetros.
PreparedStatement	Permite ejecutar sentencias SQL con parámetros de entrada.
CallableStatement	Permite ejecutar sentencias SQL con parámetros de entrada y salida, como llamadas a procedimientos almacenados.
ResultSet	Contiene las filas resultantes de ejecutar una orden SELECT.
ResultSetMetadata	Permite obtener información sobre un ResultSet.

Los siguientes **pasos** se desarrollan siempre que se utiliza una **aplicación con JDBC** (ejemplo con MySql):

1) **Importar clases.**

EJEMPLO CÓDIGO:

```
import java.sql.*;
```

2) Cargar driver JDBC: Se realiza con el método forName() de la clase Class, al que se le debe pasar un String con el nombre del driver.

EJEMPLO CÓDIGO:

```
Class.forName("com.mysql.jdbc.Driver");
```

3) Identificar el origen de datos y crear objeto Connection: Para establecer la conexión se debe arrancar el servidor MySQL y usar la clase DriverManager a través del método *getConnection(URL_BBDD, USER, PASS)***.** Dentro de este método existen varios parámetros en el que en el primero se indica la url y tiene el siguiente formato al establecer conexión con MySQL:

jdbc.mysql://nombre_servidor:puerto/nombre_BD

Donde:

- jdbc.mysql, indica la utilización de un driver JDBC para MySQL.
- nombre_servidor, aquí se indica el nombre o IP del servidor al que conectarse. Si estuviera en la misma máquina se puede poner localhost.
- puerto, es el puerto indicado para las BD de MySQL, que por defecto será 3306.
- nombre_BD, se indica el nombre de la BD a la que conectarse.

El segundo parámetro indicado será el nombre del usuario con el que se accede a la base de datos. El tercero y último será la clave del usuario indicado anteriormente.

EJEMPLO CÓDIGO:

```
Connection connect = DriverManager.getConnection(BBDD, USER, PASS);
```

4) **Crear objeto Statement y la consulta:** A través del objeto Connection obtenido anteriormente, se ha de obtener un objeto Statement utilizando el método createStatement(). Además, necesitaremos crear la consulta a ejecutar.

EJEMPLO CÓDIGO:

```
Statement sentencia = connect.createStatement();
String consultaSql = "SELECT * FROM archivos";
```

5) **Ejecutar consultas con el objeto Statement y recuperar datos del ResultSet:** La sentencia que se obtiene tendrá el método *executeQuery()* y servirá para realizar la consulta pasándole un *String con la consulta SQL*. El resultado de la sentencia lo devolverá a través de un *ResultSet*.

EJEMPLO CÓDIGO:

```
ResultSet result = sentencia.executeQuery(consultaSql);
```

6) **Procesar los resultados:** En este momento podremos usar el método next() del ResultSet obtenido para avanzar con el puntero al siguiente registro de la lista y poder así procesar los resultados. Para poder recorrer la lista completa, será necesario utilizar la función while para que ejecute el método next().

EJEMPLO CÓDIGO:

```
while (resultado.next()) {
        System.out.printf("%d, %s, %s %n",
                resultado.getInt(1),
                resultado.getString(2),
                resultado.getString(3) );
}
```

Con los métodos getInt() y getString() indicados en el ejemplo anterior se pueden obtener los valores de dichos campos indicando el valor de la posición en la tabla o indicando el nombre de la columna que se desee.

7) **Liberar los recursos:**

a) **Liberar el objeto ResultSet:**

EJEMPLO CÓDIGO:

```
result.close()  //Cerramos ResultSet.
```

b) **Liberar el objeto Statement:**

EJEMPLO CÓDIGO:

```
sentencia.close() //Cerramos Statement.
```

c) **Liberar el objeto Connection:**

EJEMPLO CÓDIGO:

```
connect.close() //Cerramos conexión.
```

Acceso a datos mediante el puente JDBC-ODBC

Existen algunos productos, aunque escasos, que no poseen un controlador JDBC, pero para los que sí existen en ODBC. Para ellos se utiliza el puente ***JDBC-ODBC Bridge*****.** Se trata de un controlador que implementa las operaciones en JDBC y las transforma en operaciones ODBC. Está implementado en Java y se instala con el JDK como el paquete ***sun.jdbc.odbc***.

Para realizar el acceso a la base de datos a través del puente JDBC-ODBC es necesario crear un origen de datos o DNS. Si se trata de Windows, a través de la ruta **Panel de Control -> Herramientas administrativas -> Orígenes de datos (ODBC)**.

Es necesario pulsar en *Agregar* y se debe seleccionar el driver ***MySQL ODBC 5.3 ANSI Driver*** y pulsar en ***Finalizar***.

Cuando se muestre la siguiente pantalla será necesario indicar el nombre del DNS en la celda ***Data Source Name***.

Además, hay que rellenar los siguientes campos: ***TCP/IP Server, Port, User, Password*** y ***Database***. Una vez rellenados se puede pulsar en el botón *Test* para verificar la conexión y, una vez completado todo, se pulsa en el botón OK.

Se abrirá la ventana *Administrador de Orígenes de datos ODBC* y se pulsa en *Aceptar* para finalizar.

Si por cualquier motivo a la hora de comprobar el driver ODBC, no estuviera instalado, habrá que descargarlo accediendo a la web:

http://www.mysql.com/products/connector/

Se **descarga el driver** de la sección:

MySQL Connectors (Connectors/ODBC 5.3.6).resultado.getString(3));

A continuación, hay que cargar el driver cambiando dos líneas en el programa:

1. **Cargar el driver:**

CÓDIGO :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

2. **Establecer la conexión con la BD:**

CÓDIGO :

```
Connection conexion = DriverManager.getConnection("jdbc:odbc:Mysql-odbc");
```

Cuando se ejecute el programa desde Linux se deberán instalar los paquetes ***unixodbc***, ***unixodbc-dev*** y ***libmyodbc,*** además de crear el origen de datos como se hizo anteriormente. El código sería idéntico.

1.3. Ejecución de sentencias de descripción de datos

Cuando se trabaja con una base de datos, es posible que se desconozca la estructura que tiene, es decir, qué tablas y datos tiene. Para conocer esta información sobre la base de datos se puede usar una de las interfaces anteriormente nombradas: **DatabaseMetaData**

La **interfaz DatabaseMetaData** es capaz de ofrecer información sobre las bases de datos a través de métodos. Algunos devolverán un ***ResultSet***.

https://docs.oracle.com/javase/8/docs/api/java/sql/DatabaseMetaData.html

Se detalla a continuación algunos de los métodos más importantes de ***DatabaseMetaData*:**

- **getTables(*String catalogo, String esquema, String patronDeTabla, String tipos[]*)**: devuelve un objeto ***ResultSet*** capaz de proporcionar información sobre tablas y vistas de la base de datos. Posee cuatro **parámetros**:
  - ***catalogo***. Obtiene las tablas del catálogo indicado.
  - ***esquema***. Obtiene las tablas del esquema indicado.
  - ***patronDeTabla***. Se indica el nombre de las tablas que se quieren obtener del método.

    tipos: Array de String. Indica qué tipo de objetos se desean obtener. *Tipos válidos: TABLE, VIEW, SYSTEM TABLE, GLOBAL TEMPORARY, LOCAL TEMPORARY, ALIAS y SYNONYM*.

- **getColumns(String *catalogo*, String *esquema*, String *patronDeTabla*, String *patronDeColumna*)**: devuelve un objeto ***ResultSet*** que posee información sobre las columnas de una o varias tablas. Para poner un patrón del nombre de la tabla o columna se puede utilizar el guion bajo o porcentaje, como se muestra en el siguiente ejemplo:

CÓDIGO :

```
getColumns(null, "ejem", "archivos", "c%")
```

De esta manera se obtienen todas las columnas que comiencen por la letra c en la tabla archivos y en el esquema ejem. El valor null obtendrá información de todas las tablas y columnas del esquema actual.

- **getPrimaryKeys**(**String catalogo, String esquema, String tabla)**: devuelve una lista de las columnas que forman la clave primaria de la tabla especificada.
- **getExportedKeys*(String catalogo, String esquema, String tabla*)**: devuelve una lista de todas las claves ajenas que utilizan la clave primaria especificada.

- **getProcedures(*String catalogo, String esquema, String procedure*)**: devuelve procedimientos almacenados.

1.4. Ejecución de sentencias de modificación de datos

Como se vio anteriormente en el paso número 4 del uso de una aplicación con JDBC, se ha de obtener un **Statement** a través del método ***createStatement()*** de un objeto ***Connection*** válido:

EJEMPLO CÓDIGO:

```
Statement sentencia = conexion.createStatement();
```

Cuando se crea un objeto ***Statement*** se está ideando un espacio de trabajo donde se pueden realizar consultas SQL, ejecutarlas y recibir los resultados.

Se pueden utilizar los siguientes **métodos**:

- **ResultSet executeQuery (String)**: utilizado para sentencias SQL capaces de recuperar datos de un objeto ResulSet, como, por ejemplo, sentencias SELECT.

- **int executeUpdate (String)**: utilizado para sentencias que no son capaces de devolver un ResultSet como pueden ser: sentencias de manipulación de datos o DML (INSERT, UPDATE y DELETE), o sentencias de definición de datos o DDL

(CREATE, DROP y ALTER). Estas devuelven un número entero indicando el número de filas que se han visto afectadas y, en el caso de que sean sentencias de definición de datos, devuelven el valor 0.

- **boolean execute (String)**: utilizado para ejecutar cualquier sentencia SQL de las anteriores, tanto las que devuelven un ResultSet como las que no. En el que caso de que lo devuelva, lo hará con el valor *true* y habrá que denominar al método **getResultSet()** para recuperar las filas. En el caso de que no lo devuelva, lo hará con *false* y se usará **getUpdateCount()** para tomar el valor devuelto.

En el paso número 5 del uso de una aplicación con JDBC obtenemos un objeto **ResultSet** de la consulta ejecutada. Usando este ResultSet se puede acceder al valor de cualquier columna de la fila en la que se esté trabajando (paso número 6).

Existen varios **métodos** que se puede utilizar **sobre el ResultSet** para desplazar el puntero:

MÉTODO	FUNCIÓN
boolean next ()	Mueve el puntero del objeto ResultSet una fila hacia adelante a partir de la posición actual. Devuelve True o False.
boolean first ()	Mueve el puntero del objeto ResultSet al primer registro de la lista.
boolean last ()	Mueve el puntero del objeto ResultSet al último registro de la lista.
boolean previous ()	Mueve el puntero del objeto ResultSet al registro anterior de la lista.

void beforeFirst ()	Mueve el puntero del objeto ResultSet justo antes del primer registro.
int getRow ()	Devuelve el número de registro actual.

A continuación, para acceder al detalle de cada resultado, podemos usar los siguientes métodos:

MÉTODO	TIPO DE JAVA DEVUELTO
getString (int numerodecolumna) **getString (String nombredecolumna)**	String
getBoolean (int numerodecolumna) **getBoolean (String nombredecolumna)**	Boolean
getByte (int numerodecolumna) **getByte (String nombredecolumna)**	Byte
getShort (int numerodecolumna) **getShort (String nombredecolumna)**	Short
getInt (int numerodecolumna)	Int

getInt (String nombredecolumna)	
getLong (int numerodecolumna) **getLong (String nombredecolumna)**	Long
getFloat (int numerodecolumna) **getFloat (String nombredecolumna)**	Float
getDouble (int numerodecolumna) **getDouble (String nombredecolumna)**	Double
getBytes (int numerodecolumna) **getBytes (String nombredecolumna)**	Byte[]
getDate (int numerodecolumna) **getDate (String nombredecolumna)**	Date
getTime (int numerodecolumna) **getTime (String nombredecolumna)**	Time
getTimestamp (int numerodecolumna) **getTimestamp (String nombredecolumna)**	TimeStamp

1.5. Ejecución de consultas

Hasta ahora, todos los ejemplos se han centrado en el método createStatement() con el se obtenía el objeto **Statement**, a partir del cual continuamos trabajando y ejecutando las consultas o modificaciones (executeQuery o executeUpdate).

Sin embargo, se dispone de una variante de esta sentencia Statement, denominada **PreparedStatement**. Estas sentencias permiten que los parámetros de entradas se establezcan de forma dinámica, ganando eficiencia.

Anteriormente se han creado sentencias SQL en las que se van concatenando los datos para construir las sentencias completas. Sin embargo, usar la interfaz ***PreparedStatement*** permitirá construir cadenas de caracteres SQL con ***placeholder***, que representarán los datos asignados más tarde mediante el símbolo **(?).**

Con cada *placeholder* existe un índice en el que el 1 se corresponde con el primer lugar en la cadena, el 2 con el segundo, y así sucesivamente.

Solo se pueden utilizar para representar datos en la cadena SQL pero no para representar columnas o tablas. Será necesario establecer los datos cuando haya que ejecutar un ***PreparedStatement*** para que, al ejecutar la BD, se asignen variables de unión y se ejecute la orden SQL.

Para ejecutar varias veces usando diferentes valores, es necesario usar objetos ***PreparedStatement***, sin embargo, en los *Statement* la sentencia se debe suministrar cuando se ejecuta.

Los métodos de ***PreparedStatement*** son idénticos a los de ***Statement*** *(executeQuery(), executeUpdate() y execute()),* con la única diferencia de que los primeros no necesitan que se envíe la cadena de caracteres con la orden SQL, ya que esa función la hace el método ***prepareStatement(String).***

EJEMPLO CÓDIGO:

```
PrepareStatement ps = connection.prepareStatement(consultaSql);
```

Cuando se deseen asignar valores a cada uno de los *placeholder*, se utilizan los **métodos setXXX():**

EJEMPLO CÓDIGO:

```
public abstract void setXXX(int indice, tipoJava valor) throws SQLException
```

Los **métodos setXXX**() son:

MÉTODO	TIPO SQL
void setString (int índice, String valor)	VARCHAR
void setBoolean (int índice, boolean valor)	BIT
void setByte (int índice, byte valor)	TINYINT
void setShort (int índice, short valor)	SMALLINT
void setInt (int índice, int valor)	INTEGER
void setLong (int índice, long valor)	BIGINT
void setFloat (int índice, float valor)	FLOAT
void setDouble (int índice, doublevalor)	DOUBLE

Void setBytes (int índice, byte [ ] valor)	VARBINARY
Void setTime (int índice, time valor)	TIME
Void setDate (int índice, date valor)	DATE

Para asignar un valor NULL se utiliza el **método setNull(),** cuya sintaxis sería:

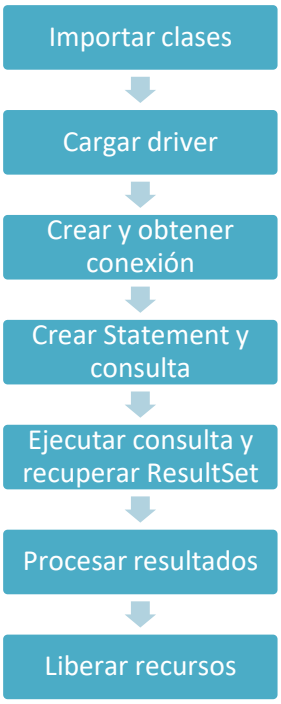
CÓDIGO:

```
void setNull (int índice, int tipoSQL)
```

En el que ***tipoSQL*** es una constante definida en la librería ***java.sql.Types***.

Ejemplo final

Con el siguiente diagrama se pueden recordar los pasos para acceder a una base de datos (MySQL en este caso) utilizando JDBC. Posteriormente se presenta un ejemplo práctico en código Java siguiendo los pasos previamente establecidos:



EJEMPLO CÓDIGO:

```
// Cargar el driver (mysql)
Class.forName("com.mysql.jdbc.Driver");

// Crear objeto Connection y establecer conexion con la base de
datos
Connection connect =
DriverManager.getConnection("jdbc:mysql://localhost/prueba",
"userPrueba", "pwdPrueba");

//Crear objeto Statement
Statement sentencia = connect.createStatement();

//Crear la consulta
String consultaSql = "SELECT * FROM usuarios";

//Ejecutar consulta obteniendo los resultados en el ResultSet
ResultSet result = sentencia.executeQuery(consultaSql);

//Procesar los resultados recorriendo cada fila
while (result.next()) {


      System.out.println(result.getInt(1) + " - " +
result.getString(2));
}

//Liberar los recursos
result.close(); //Cerrar ResultSet
sentencia.close(); //Cerrar Statement
connect.close(); // Cerrar Connection
```

2. Herramientas de mapeo objeto-relacional (ORM)

Este segundo punto de la unidad se centra en el mapeo de una base de datos, en cómo instalar y configurar una herramienta ORM, además de en cómo interpretar y definir los archivos de mapeo.

Además, es necesario saber aplicar mecanismos de persistencia que puedan desarrollar distintas aplicaciones para poder insertar, modificar y recuperar objetos persistentes.

Por último, se realizarán consultas con el lenguaje de la herramienta ORM y se gestionarán transacciones.

El **contenido** se divide en:

- Concepto de mapeo objeto-relacional.
- Características de las herramientas ORM. Herramientas ORM más utilizadas.
- Instalación de una herramienta ORM.
- Estructura de un fichero de mapeo. Elementos y propiedades.
- Clases persistentes.
- Sesiones y estados de un objeto.
- Carga, almacenamiento y modificación de objetos.
- Consultas SQL.

2.1. Concepto de mapeo objeto-relacional

El **mapeo objeto-relacional (ORM)** permite convertir datos que se encuentran en un sistema de bases de datos orientado a objetos, a otro sistema gestor de bases de datos de tipo relacional.

Para ello, se crea una BDOO virtual sobre la relacional, para así poder utilizar las características de la orientación a objetos.

Existen paquetes que realizan el mapeo, pero los programadores prefieren usar sus propias herramientas ORM.

2.2. Características de las herramientas ORM. Herramientas ORM más utilizadas

Las herramientas ORM permiten crear una **capa de acceso a los datos** de una manera sencilla, creando una clase por cada tabla de la BD y, de esta manera, permiten mapear una a una.

El lenguaje de consultas es independiente de la BD, por lo que se puede utilizar en unas bases de datos u otras.

Ventajas e inconvenientes del uso de estas herramientas:

Ventajas	Inconvenientes
Reducción del tiempo de desarrollo	Aplicaciones lentas
Abstracción de la BD	Aprendizaje de nuevo lenguaje ORM
Reutilización	Librerías más amplias
Producción de mejor código	Sistema más complejo
Independencia de la BD	
Propio lenguaje para consultas	
Portabilidad y escalabilidad de programas de software	

Existen varias **herramientas ORM**, entre ellas:

- Para proyectos PHP: *Doctrine*, *Propel* o *ADOdb Active Record.*
- Para mapeo objeto-relacional: *LINQ* de Microsoft.
- Para tecnología Java: *Hibernate.*
- Para tecnología .NET: *Nhibernate.*
- Otras: *QuickDB*, *iPersist*, *Java Data Objects*, *Oracle Toplink.*

Hibernate es una de las herramientas más populares.

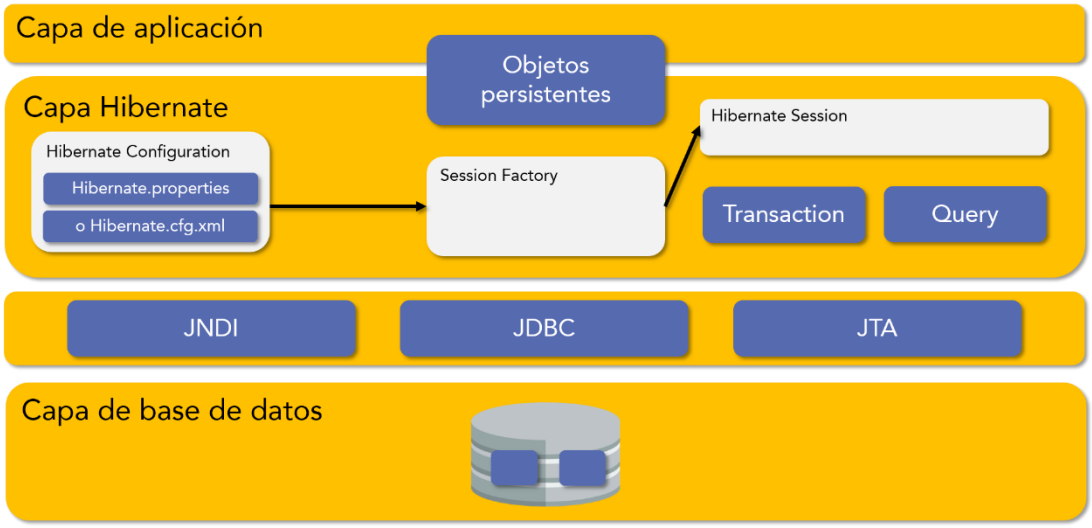
Hibernate es una herramienta para Java que facilita el mapeo de objetos entre las bases de datos relacionales y la de objetos mediante ficheros declarativos XML que permiten establecer estas relaciones.

El programa, mediante el uso de factorías y otros elementos, va a construir las consultas en lugar del usuario. Ofrece un lenguaje llamado **HQL (Hibernate Query Language)** que podrá acceder a datos a través de POO.

Arquitectura Hibernate

La **función principal** de Hibernate es mapear objetos Java normales para poder almacenarlos y recuperarlos. Para ello se puede crear una sesión a través de la clase **Session**, para, de esta manera, poder comunicarse con el motor de Hibernate.

Se utilizan distintos **niveles de capas**:



Esta clase **Session** (**org.hibernate.Session**) ofrece distintos métodos, como pueden ser **save(Object objeto)**, **createQuery(String consulta)**, **beginTransaction()**, **close()**, y otros, con los que se puede interactuar con la BD pero de manera más simple que con JDBC.

Estas sesiones trabajan como una caché, ya que ocupan poco espacio en memoria y es necesario ir creando y destruyendo sesiones continuamente.

Existen **varias interfaces**:

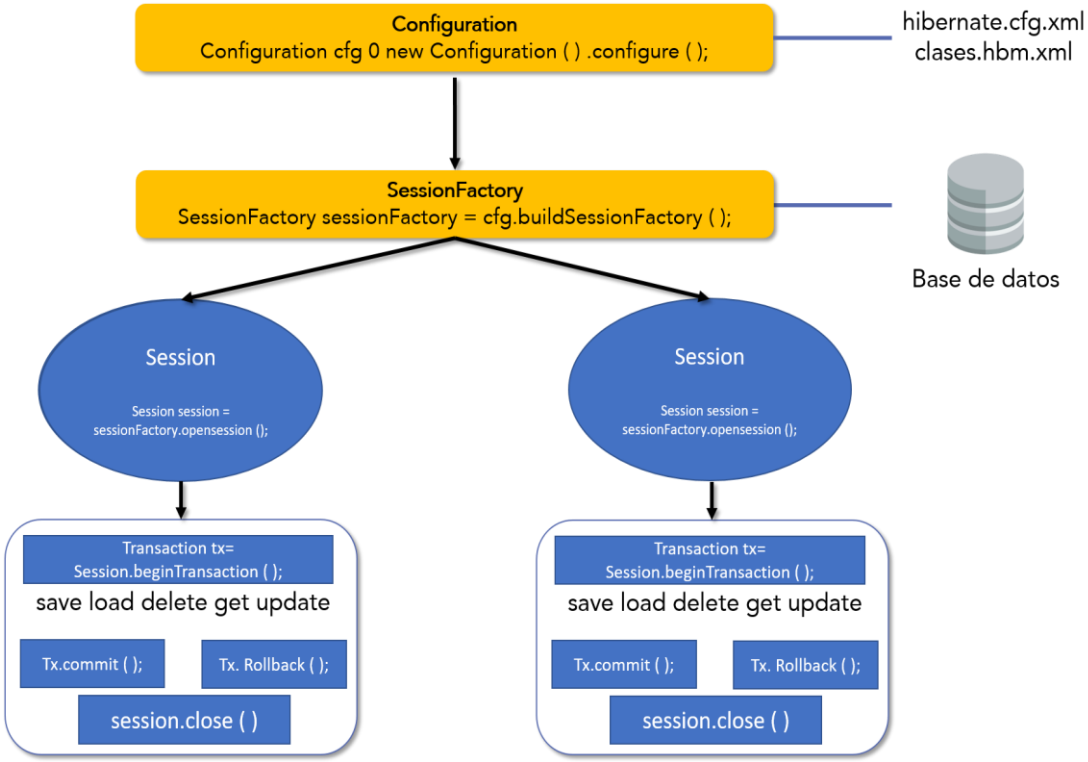
a) **SessionFactory** (*org.hibernate.SessionFactory*): esta interfaz permite la obtención de instancias **Session** y habitualmente solo existe una por sesión, aunque para cada base de datos será necesario crear una distinta.

b) **Configuration** (*org.hibernate.cfg.Configuration*): con ella se configura Hibernate. Se especifica la ubicación de los documentos que son utilizados para el mapeo de objetos y seguidamente se crea la **SessionFactory**.

c) **Query** (*org.hibernate.Query*): aquí se llevan a cabo las consultas. Se harán en HQL o en SQL nativo, según la base de datos que se esté usando.

d) **Transaction** (*org.hibernate.Transaction*): cualquier error en la transacción producirá el fallo de la misma.

Las APIs utilizadas serán las de Java como **JDBC, JTA y JNDI**.



2.3. Instalación de una herramienta ORM

En este apartado se muestra la **instalación y configuración Hibernate** en el entorno Eclipse.

Instalación del plugin

Para la instalación del *plugin* es necesario tener acceso a Internet. Se inicia Eclipse y, a continuación, en la barra de herramientas, se acude a la ruta **Help -> Intall New Software**, en la que se completa el campo **Work With** con la siguiente dirección: *http://download.jboss.org/jbosstools/updates/stable/* y se pulsa en *Add*. Por último, se pedirá el nombre, en el que se puede escribir *Hibernate* y se pulsa en OK.

Cuando pasen unos minutos aparecerá una nueva ventana con los plugins a instalar. Hay que acudir al plugin llamado **JBoss Data Services Development** para desplegarlo y seleccionar también **Hibernate Tools**. Se pulsa en ***Next*** y comenzará la descarga.

A continuación, aparecerá una ventana con la información del plugin a instalar y se pulsará en ***Next***. Durante la instalación es necesario ir confirmando las licencias ya que es software sin firmar.

Por último, una vez completada la instalación, hay que reiniciar Eclipse y se comprueba que se ha instalado correctamente entrando en la siguiente ruta en el menú: **Windows -> Show View -> Other**, donde deben aparecer las opciones de Hibernate.

Configuración del driver MySQL

Se procede a configurarlo para que pueda comunicarse con MySQL.

En primer lugar, hay que descargar el driver MySQL de la siguiente dirección: http://dev.mysql.com/downloads/connector/j/.

Desde el menú **Windows -> Preferences -> Data Management -> Connectivity -> Driver Definitions** se pulsa en el botón *Add*:

1. En la primera pestaña llamada **Name/Type** se selecciona en la opción *Vendor Filter* la característica *MySQL*, en la cual se selecciona de entre los drivers que aparecen la opción *MySQL JDBC Driver*.
2. En la segunda pestaña llamada **JAR List** habrá que pulsar en el botón ***Add JAR/Zip*** donde se localiza el conector **mysql-connector-java-5.1.38-bin.jar** en la carpeta del equipo. Si aparece el conector **mysql-connector-java-5.1.0-bin.jar** hay que eliminarlo pulsando en ***Remove JAR/Zip***.

Después, se procede a crear un proyecto nuevo en Eclipse al que se le denomina ***PrimerProyecto*** y, además, se debe configurar Hibernate para que se comunique con MySQL y, de esta manera, se creen las clases de la base de datos.

Se pulsa en el menú **File -> New -> Project -> Java Project** y después en Next. A continuación, se indica el nombre del proyecto, *PrimerProyecto*. El programa preguntará si se quiere abrir la perspectiva y se pulsará en *Yes*.

Para agregar el driver MySQL hay que pulsar con el botón derecho del ratón sobre el proyecto e ir a la ruta **Build Path -> Add Libreries**, donde se visualizará una nueva ventana y donde se elegirá la opción **Connectivity Driver Definition** y se pulsará en *Next*.

En la ventana que aparece se escoge la opción *MySQL JDBC Driver* y se pulsa en *Finish*.

De esta manera, ya estará configurado el proyecto con el driver MySQL.

Configuración del Hibernate

Una vez que se ha configuración el driver MySQL se debe crear un archivo XML para conectar con la base de datos. El archivo se llamará **hibernate.cfg.xml** y habrá que acudir a la ruta: **New -> Other -> Hibernate -> Hibernate Configuration File (cfg.xml)**.

Cuando se pulsa en *Next* aparecerá una ventana en la que habrá que seleccionar la ruta donde crear el archivo. En este caso, se seleccionará el directorio por defecto ***src***.

Al pulsar en *Next* aparecerá una nueva ventana donde será necesario configurar la conexión poniendo los siguientes **campos**:

Nombre campo	Definición
Session Factory name	Nombre de la conexión a MySQL.
Database dialect	Cómo se comunica JDBC con la BD (se escoge MySQL).
Driver Class	Elección de la clase JDBC usada para la conexión (**com.mysql.jdbc.Driver**).
Conection URL	Ruta de conexión a BD (*jdbc:mysql://localhost/prueba*).
Username	Nombre de usuario que se conectará a la BD.
Password	Clave del usuario indicado.

En la parte inferior existe una casilla llamada ***Create a console configuration*** que creará el archivo **XML Hibernate Console Configuration** con el mismo nombre que el proyecto, si se deja marcada.

Si no se deja pulsada la casilla, será necesario crear el archivo manualmente, para ello se acude a la ruta **New -> Other -> Hibernate -> Hibernate Console Configuration** y se pulsa en *Next*.

En la nueva ventana se indicará el nombre ***PrimerProyectoConsoleConfiguration*** en el campo **Name**.

Habrá que comprobar que el proyecto aparece en el campo **Project** y en archivo ***hibernate.cfg.xml*** aparece en el campo ***Configuration file***. Si todo está correcto se pulsa en **Finish**.

Por último, habrá que crear el archivo **XML Hibernate Reverse Engineering (reveng.xml)** que se encargará de crear las clases para las tablas MySQL.

Para ello, se pulsa el botón derecho del ratón sobre el proyecto y se selecciona la ruta: **New -> Other -> Hibernate -> Hibernate Reverse Engineering File (reveng.xml)**.

Al pulsar el botón *Next,* pedirá la ruta de guardado del archivo y se debe seleccionar la misma que en el archivo ***hibernate.cfg.xml***. En este caso será la carpeta ***src***.

En la siguiente ventana hay que seleccionar las tablas a mapear. En la lista **Console configuration** se debe seleccionar el nombre del archivo ***Hibernate Console Configuration*****.** En este caso será el archivo llamado ***PrimerProyectoConsoleConfiguration*** y se pulsará en ***Refresh*** para que actualice la base de datos con sus tablas.

A continuación, se seleccionan las tablas que se deseen incluir y después se pulsa en *Include* para incluirlas. Para finalizar se pulsará en *Finish*.

Generar clases de la base de datos

En primer lugar, se crean las clases en la base de datos. Se pulsa en la flecha situada en el icono **Run As** y se selecciona **Hibernate Code Generation Configurations**.

Se hace doble clic sobre la opción ***Hibernate Code Generation*** y aparecerán varias pestañas. En la pestaña Main se configura:

- **Name**: configurar nombre, *Proyecto1*.
- **Console configuration**: seleccionar *PrimerProyectoConsoleConfiguration*.
- **Output directory**: deberá ser la carpeta src seleccionada anteriormente.
- **Package**: nombre del paquete donde se crearán las clases, por ejemplo, *primerpaquete*.
- **reveng.xml**: se localiza el fichero con el mismo nombre creado anteriormente.

En la siguiente pestaña llamada ***Exporters*** se indican los archivos que se deseen generar marcando las casillas: ***Use Java 5 syntax***, ***Domain code, Hibernate XML Mappings*** e ***Hibernate XML Configuration***.

Una vez seleccionadas se pulsa en *Apply* y seguidamente en *Run*.

Cuando se ejecute, se generará un paquete llamado ***primerpaquete*** con las clases indicadas anteriormente y con sus respectivos métodos ***getters*** y ***setters*** de cada campo, así como los archivos XML que contendrán la información del mapeo.

Primera consulta en HQL

Para realizar una consulta en HQL y comprobar que funciona la conexión con la base de datos, se abrirá la perspectiva de Hibernate con la siguiente ruta: **Windows -> Perspective -> Open Perspective -> Other -> Hibernate**.

Ahora, desde la pestaña **Hibernate Configurations,** se pulsará con el clic derecho del ratón sobre *PrimerProyectoConsoleConfiguration* y se elegirá la opción **HQL Editor**.

Se abrirá una ventana con el mismo nombre que la configuración de Hibernate, en la que se pueden indicar las consultas que uno desee. En la pestaña **Hibernate Query Result** se pueden ver los resultados de las consultas realizadas. Al seleccionar una fila se podrá ver en el panel ***Property*** el contenido de la misma.

Si se quiere mostrar un mapeo de las clases y tablas de la BD, se debe pulsar con el botón derecho del ratón en **Configuration** y seleccionar **Mapping Diagram**.

Empezando a programar con Hibernate en Eclipse

Con lo realizado anteriormente no es posible programar en Java, sino que hay que llevar a cabo los siguientes pasos:

1. Descargar versión estable de Hibernate desde la url: http://hibernate.org/orm/downloads/.

2. Preparar la distribución para agregarla a Eclipse. Se crea una carpeta dentro de Eclipse donde se descomprime el archivo descargado.

3. Dentro de la carpeta ***/lib/required*** del archivo descomprimido se copia todo el contenido a la carpeta creada en Eclipse.

4. Se procede a descargar la librería **slf4j** desde http://www.slf4j.org/download.html y se buscan los ficheros **slf4j-api-1.7.21.jar y slf4j-simple-1.7.21.jar** para copiarlos a la carpeta creada en Eclipse.

5. Se pulsa en Eclipse con el botón derecho sobre el proyecto y se selecciona **Build Path -> Add Libreries**. En la nueva ventana se elige **User Library**. Se crea una librería con todos los JAR necesarios.
6. En la siguiente ventana se pulsa sobre ***User Libraries*** y seguidamente sobre el botón ***New***. Ahora pedirá un nombre para la librería, por ejemplo, se puede escribir **LibreriaHibernate**, y se pulsa en *OK*.
7. A continuación, se pulsa el botón **Add External JARs** para localizar todos los ficheros JAR de la carpeta, se seleccionan todos y se pulsa en Abrir. Tras visualizar todos los archivos se pulsa en OK y después en *Finish*.

Ahora ya es posible crear el primer programa en Java para comunicarse con la base de datos.

En primer lugar, se crea una instancia a la base de datos para poder trabajar con ella y, para ello, será necesario crear un **Singleton**.

Un **Singleton** es un patrón de diseño capaz de restringir la creación de objetos pertenecientes a una clase. Solo permite que haya una instancia por clase y un punto de acceso global.

Será implementado por un método y accederá por medio de una clase de ayuda llamada ***SessionFactory*** de la que se obtiene el objeto de sesión. Solo existe una ***SessionFactory*** por aplicación.

Se definirá una variable llamada *sesión* que recogerá el objeto devuelto por el método ***buildSessionFactory()***. A través del método ***getSessionFactory()*** se devolverá el objeto ***SessionFactory*** creado.

La clase se llamará ***HibernateUtil.java*** y estará incluida en el paquete *primerpaquete*.

Con la clase creada se puede acceder a la sesión desde cualquier parte de la aplicación.

Algunos **métodos** que se pueden usar son:

- **save()**: es un método de la sesión que se puede utilizar para guardar el objeto. Como argumento se le pasa el objeto.
- **commit()**: realizar un *commit* de la transacción actual. Se crea al método ***beginTransaction()*** de la sesión actual. Será necesario para que los datos queden almacenados en la BD.
- **close()**: cierra la sesión.

2.4. Estructura de un fichero de mapeo. Elementos y propiedades

Para relacionar las tablas con los objetos, Hibernate usa unos ficheros de mapeo que estarán en formato XML y tendrán extensión ***.hbm.xml*.** Los ficheros creados se guardarán en el mismo directorio que las clases Java. Todos formarán parte del paquete creado.

Algunas **etiquetas del archivo XML** son:

- **hibernate-mapping**: todos los ficheros de mapeo comenzarán y finalizarán con esta etiqueta.
- **class**: esta etiqueta engloba a la clase con todos sus atributos. **Name** indica el nombre de la clase; **table**, nombre de la tabla que representa al objeto; y **catalog**, el nombre de la base de datos.

CÓDIGO :

```
<class name="nombredeclase" table="nombretabla" catalog="nombreBD">
```

Dentro de **class** existe también la etiqueta **id** con los siguientes atributos: **name**, que indica el atributo clave en la clase, y **type**, que indica el tipo de datos. Además, tendrá las siguientes etiquetas: **column** para indicar el nombre de la tabla y **generator** que indica la naturaleza del campo clave. Tendrá dos opciones: **assigned**, cuando sea asignado por el usuario, o **increment**, cuando sea autogenerado por la base de datos.

CÓDIGO :

```
<id name="atributo" type="tipodato">
    <column name="nombretabla" />
    <generator class="assigned" />
</id>
```

- **set**: mapea colecciones. Se definen varios atributos: ***name***, indica nombre del atributo; ***table***, tabla donde se tomará la colección; la etiqueta ***key deine,*** que es el nombre de la columna; y el ***elemento once-to-many,*** que define la relación.

CÓDIGO :

```
<set name="atributo" table="tabla" inverse="true" lazy="true" fetch="select">
    <key>
        <column name="tabla" not-null="true" />
    </key>
    <one-to-many class="relación" />
</set>
```

2.5. Clases persistentes

Según lo anteriormente explicado, entre las etiquetas **hipernate-mapping**, existe la etiqueta ***class*** que define una clase.

Se denominan **clases persistentes** a las clases que implementan las entidades del problema, implementadas por la interfaz ***Serializable***. Son similares a las tablas y un registro será un objeto persistente en esa clase. También poseerán atributos y **métodos *get* y *set*** con los que se puede acceder a los mismos.

Utilizan ***JavaBeans*** para el nombrado, así como la visibilidad privada en los campos. Como son objetos privados, se crean métodos públicos para que puedan retornar un valor de un atributo, método ***getter***, o cargar un atributo, método ***setter***.

Para que una clase pueda considerarse como persistente debe cumplir las siguientes condiciones: poseer un id, un constructor vacío, métodos *getter* y *setter* y sobreescribir los métodos ***equals()*** y ***hascode()***.

2.6. Sesiones, estados de un objeto. Transacciones

Si se desea utilizar la **persistencia en Hibernate** es necesario obtener un objeto **Session** usando la clase ***SessionFactory***.

CÓDIGO :

```
// Inicializa el entorno Hibernate
Configuration cfg = new Configuration (). configure ();

// Crea el ejemplar de SessionFactory
SessionFactory sessionFactory = cfg.buildSessionFactory(
        new StandardServiceRegistryBuilder (). configure (). build () )

// Obtiene un objeto Session
Session session = sessionFactory.openSession ();
```

Para cargar el fichero de configuración ***hibernate.cfg.xml*** hay que llamar a ***Configuration().configure()*** y será necesario crear un objeto ***StandardServiceRegistry*** que contenga los servicios para crear **SessionFactory**.

Transacciones

Con un objeto **Session** se representa una única unidad de trabajo en la que se almacenan los datos. Se abren a través de **SessionFactory**.

Cuando se crea la sesión, también se está creando la transacción para esa sesión. Una vez se ha completado todo el trabajo en dicha transacción, habrá que cerrar todas las sesiones. ***Por ejemplo***:

CÓDIGO :

```
Session session = sesion.openSession(); //crea la sesión
Transaction tx = session.beginTransaction(); //crea la transacción
//Código de persistencia
Tx.commit();   //valida la transacción
session.close() //cierra la sesión
```

Estados de un objeto Hibernate

Hibernate podrá tener los siguientes **tres estados**:

1) **Transitorio (Transient)**: será transitorio si ha sido recién instanciado por el operador *new*, no está asociado a una **Session**, no tiene representación persistente en la BD y no se le asigna ningún valor. Solo utiliza la **Session** para crear un objeto persistente. Las instancias recién instanciadas se crean como ***transitorias***, aunque se pueden crear ***transitorias persistentes*** si se asocian a una sesión.

2) **Persistente (Persistent)**: será persistente cuando se encuentre en la base de datos. Se caracterizan por haber sido creados y almacenados en una sesión o devueltos en una consulta.

3) **Separado (Detached)**: se encontrará en este estado cuando se cierra la sesión a través del método ***close()***. Una instancia que sea separada será un objeto persistente y su sesión habrá sido cerrada. Podrá ser asociada a una nueva **Session** más tarde haciéndola persistente de nuevo.

2.7. Carga, almacenamiento y modificación de objetos

Para realizar la **carga de objetos** se pueden usar los siguientes **métodos** de *Session*:

Método	Descripción
<T> T load(Class<T> Clase, Serializable id)	Devuelve instancia persistente, si no existe instancia devuelve una excepción.
Object load(String nombreClase, Serializable id)	Similar al anterior, pero indicando el parámetro con formato ***String.***
<T> T get(Class<T> Clase, Serializable id)	Devuelve instancia persistente, si no existe devuelve ***null.***
Object get(String nombreClase, Serializable id)	Similar al anterior, pero indicando en el primer parámetro el nombre de la clase.

Para realizar el **almacenamiento, la modificación y el borrado** de los objetos se pueden utilizar los siguientes **métodos** de *Session*:

Método	Descripción
Serializable save(Object obj)	Guarda el objeto. Hace que la instancia transitoria sea persistente.
void update(Object objeto)	Actualiza el objeto en la base de datos. Se debe pasar por ***load()*** o ***get().***
void delete(Object objeto)	Elimina el objeto de la base de datos. Debe ser cargado por ***load()*** o ***get().***

2.8. Consultas SQL.

En Hibernate se utiliza un lenguaje de consultas orientado a objetos denominado **HQL (Hibernate Query Language)** que es muy potente, pero a la vez sencillo de utilizar. Es una extensión orientada a objetos de SQL, por lo que las instancias de HQL y SQL nativas se representan con una instancia ***org.hibernate.Query*.** Ofrece distintos **métodos** para la realización de consultas:

MÉTODO	DESCRIPCIÓN
Iterator iterate ( )	Devuelve en un objeto Iterator el resultado de la consulta.
List list ( )	Devuelve el resultado de la consulta en un List.
Query setFetchSize (int Size)	Fija el número de resultados a recuperar en cada acceso a la base de datos al valor indicado en Size.
Int executeUpdate ( )	Ejecuta la sentencia de modificación o borrado. Devuelve el número de entidades afectadas.
String getQueryString ( )	Devuelve la consulta en un String.
Object uniqueResult ( )	Devuelve un objeto (cuando se sabe que la consulta devuelve un objeto) o nulo si la consulta no devuelve resultados.

Query setCharacter (int posición, char valor) **Query setCharacter (String nombre, char valor)**	Asigna el *valor* indicado en el método a un parámetro de tipo CHAR. *posición* indica la posición del parámetro dentro de la consulta y *nombre* el nombre del parámetro dentro de la consulta.
Query setDate (int posición, Date fecha) **Query setDate (String nombre, Date fecha)**	Asigna la fecha a un parámetro de tipo DATE.
Query setDouble (int posición, double valor) **Query setDouble (String nombre, double valor)**	Asigna valor a un parámetro de tipo decimal.
Query setInteger (int posición, int valor) **Query setInteger (String nombre, int valor)**	Asigna valor a un parámetro de tipo entero.
Query setString (int posición, String valor) **Query setString (String nombre, String valor)**	Asigna valor a un parámetro de tipo VARCHAR.

Query setParameterList (String nombre, Collection valores)	Asigna una colección de valores al parámetro cuyo nombre se indica en nombre.
Query setParameter (int posición, Object valor)	Asigna el valor al parámetro indicado en *posición.*
Query setParameter (String nombre, Object valor)	Asigna el valor al parámetro indicado en *nombre.*
Int executeUpdate ( )	Ejecuta una sentencia UPDATE o DELETE, devuelve el número de entidades afectadas por la operación.

Algunos **otros** pueden ser:

- **createQuery()** -> realizar una consulta.
- **list()** o **iterate()** -> recuperar datos de una consulta.
- **uniqueResult()** -> ofrecer un atajo cuando el resultado devuelve un objeto.

Parámetros en las consultas

Hibernate soporta **dos tipos de parámetros**: los que tienen nombre y parámetros de estilo JDBC.

Los primeros son identificadores de la forma*:* ***nombre*** en la cadena de consulta.

Tienen una numeración que comienza con el primer parámetro y el valor 0, el segundo con el 1, y así sucesivamente. Estos parámetros con nombre tienen ciertas

ventajas: son insensibles al orden en la cadena de consulta, aparecen múltiples veces y son autodocumentados.

Cuando se quieren asignar valores se utilizan los métodos **setXXX**. La manera más sencilla es utilizando el método ***setParameter()***.

También se pueden usar otros métodos como, por ejemplo, ***setInteger()*** para dar valor al campo, o ***setString(),*** para asignar una cadena, o ***setDate(),*** que asigna un valor a un parámetro tipo Date.

Consultas sobre clases no asociadas

Al realizar las consultas, si no están las tablas asociadas a las clases, se puede usar la clase ***Object*** para la recuperación de los datos. Estos resultados de las consultas se reciben en un *array* en el cual el primer parámetro será la primera clase a la que hace referencia, el siguiente elemento se referirá a la siguiente clase, y así sucesivamente.

CÓDIGO :

```
String hql = "from Empleados e, Departamentos d
                where e.departamentos.deptNo = d.deptNo order by apellido";
Query cons = session.createQuery (hql);
iterator q = cons.iterate ();
while (q.hasNext()) {
        Object [] par = (Object []) q.next ();
        Empleados em = (Empleados) par [0];
        Departamentos de = (Departamentos) par[1];
System.out.printf ("%s, %.2f, %s, %s %n",
        em.getApellido (), em.getSalario (), de. getDnombre (), de getLoc ());
}
```

Funciones de grupo en las consultas

Utilizar funciones de grupo en las consultas HQL o SQL proporciona unos resultados con un único valor usando el método ***uniqueResult()***.

Si en las consultas participan distintas funciones de grupo y devuelven varias filas, se pueden usar objetos que son devueltos por las consultas.

Objetos devueltos por las consultas

Cuando se quieren obtener objetos que no están asociados a ninguna clase, es necesario crear una clase nueva y utilizarla sin necesidad de realizar ningún mapeo. Cada fila, en este caso, devolverá un objeto de esa clase.

También se podrá hacer a través de un **array de objetos**, con la clase **Object**. Dentro de cada fila será necesario crear un array para acceder a los atributos o columnas.

INSERT, UPDATE y DELETE

Al usar el lenguaje HQL se pueden utilizar las funciones INSERT, UPDATE y DELETE.

- La instrucción **INSERT** permite crear o insertar nuevos registros en una tabla de la base de datos. La **sintaxis** será la siguiente:

CÓDIGO :

```
INSERT INTO table (column1 [, column2, column3 ... ]) VALUES (value1 [, value2, value3…])
```

- La función **UPDATE** actualiza registros de una tabla y **DELETE** los elimina. Para usar UPDATE y DELETE se aplicará la siguiente **sintaxis**:

CÓDIGO :

```
(UPDATE | DELETE) [FROM] NombreDeLaEntidad [WHERE condición]
```

En el que FROM y WHERE son opcionales.

Para ejecutar un método UPDATE o DELETE en HQL, se hará a través del método ***executeUpdate(),*** que devolverá las entidades afectadas por la consulta. Además, será necesario realizar el *commit* para que la transacción sea válida.

3. Bases de datos objeto-relacionales y orientadas a objetos

En este tercer y último punto de la unidad se identifican los pros y contras de las BD que almacenan objetos, así como se muestra la manera de gestionar distintos tipos de objetos. Se aprenderá a gestionar la persistencia de objetos simples y estructurados, además de poder modificar los objetos almacenados en las bases de datos.

Contenidos principales:

- Características de las bases de datos objeto-relacionales.
- Gestión de objetos con SQL. Especificaciones en estándares SQL.
- Acceso a las funciones del gestor desde el lenguaje de programación.
- Características de las bases de datos orientadas a objetos.
- Tipos de datos: básicos y estructurados.
- La interfaz de programación de aplicaciones de la base de datos.

3.1. Características de las bases de datos objeto-relacionales

Las bases de datos que se han visto hasta ahora ofrecen la posibilidad de crear entidades y atributos. En cada una de estas entidades, se posibilitan relaciones entre atributos y entidades. Todas ellas siguen una serie de reglas o pautas entre las entidades y atributos a la hora de llevarlas a cabo.

Las bases de datos pueden modelar o ir modificando todas estas características, mientras que las bases de datos **objeto-relacionales** son como una extensión del modelo relacional que se ha visto anteriormente. No utilizan algunas reglas que antes sí se llevaban a cabo, pero añaden otra nueva solución a la hora de modelar la información.

La mayor diferencia que hay entre los dos modelos es la existencia de **tipos**.

Los **tipos** se crean mediante sentencias DDL y van a ser la base del desarrollo de las bases de datos objeto-relacionales.

Cuando haya que definir el modelo objeto-relacional, se añaden a las características anteriores (entidades, relaciones, atributos y reglas), los tipos.

La principal característica que se debe destacar de las bases de datos objeto-relacionales es que combinan el modelo relacional, evolucionando con la incorporación de conceptos del modelo orientado a objetos. Por ejemplo, este tipo de bases de datos permiten la creación de nuevos tipos de datos manteniendo todo lo que ya posee del modelo entidad/relación, entidades, atributos, relaciones y reglas.

En el modelo objeto-relacional:

1. Cada **registro** de una tabla se considera un **objeto**.

2. La definición de la **tabla** se considera su **clase**.

Este modelo tiene capacidad para gestionar tipos de **datos complejos**, lo cual contradice algunas de las restricciones establecidas por el modelo relacional.

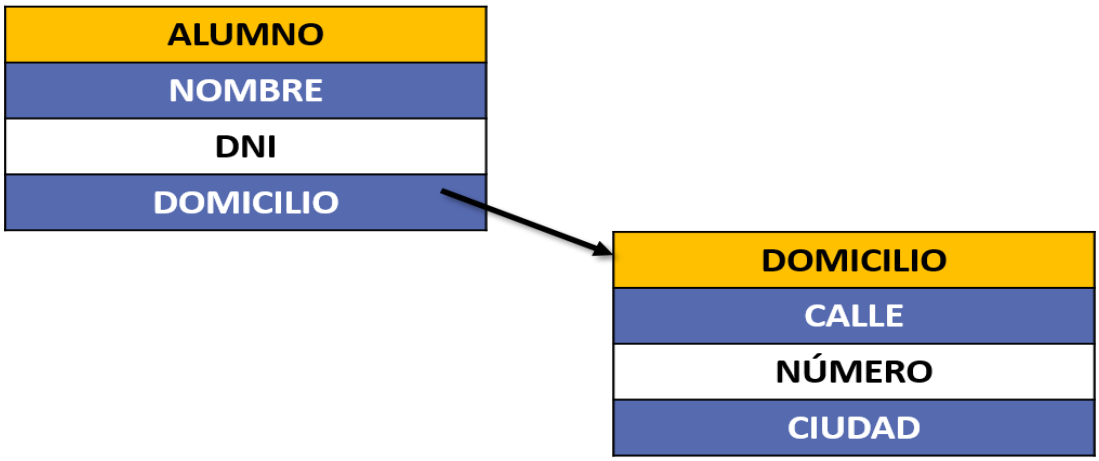
- Permite **campos multievaluados**, con lo cual, contradice frontalmente la 1FN.

ALUMNOS		
DNI	**NOMBRE**	**ASIGNATURAS**
123456P	Paco	{M01, M03A}
321456O	Marta	{M05, M02B}

- Las tablas dejan de ser elementos bidimensionales para pasar a convertirse en **estructuras de datos bastante más complejas**.
  Por ejemplo: una columna puede ser tipo "Asignatura" conteniendo toda la información de esta y no únicamente su clave ajena.

ALUMNOS		
DNI	**NOMBRE**	**ASIGNATURAS**
123456P	Paco	{**aNombre**: Bases de datos **aCodigo**:M02A}

- De la misma manera, es posible que los valores que adopte un campo sean registros de otra tabla, es decir, **colecciones de objetos**. Es lo que se conoce como "**Modelo Relacional Anidado**".



Por otra parte, las bases de datos relacionales se caracterizan porque se basan en una **Programación Orientada a Objetos (POO)** que se desarrolla según tres **elementos** fundamentales: **encapsulamiento, herencia y polimorfismo**.

Encapsulamiento

Es el mecanismo que se sigue para **agrupar los atributos y métodos dentro de un nuevo concepto que se denomina clase**.

Mediante el uso de clases, se puede abstraer un tipo de objeto como un conjunto de datos y acciones que están agrupadas.

El encapsulamiento establece que **se deben tener definidos todos los métodos** que vayan a formar parte de las clases que intervengan. Estos métodos deben acompañar al objeto.

Herencia

Es el mecanismo mediante el cual una clase derivada va a heredar los atributos de otra. **Actúa de forma jerárquica**, es decir, puede seguir el principo de **derivación** (de arriba a abajo) o de **generalización** (de abajo a arriba).

Todo aquello que esté definido en la clase principal, no es necesario que vuelva a definirse en los tipos derivados, a menos que se quiera realizar cualquier modificación.

Polimorfismo

Se utiliza el polimorfismo cuando un**a clase derivada debe verse como la clase principal**. Esto significa que se dispone de un objeto de una clase derivada, pero se refiere a él como si fuera un objeto de una clase principal.

Además, permite ejecutar diversas funciones que han sido sustituidas, como, por ejemplo: ***overrided*** **(sobrecargadas)** u ***overwritten*** **(sobreescritas)**. PL/ SQL utiliza la sobrecarga.

3.2. Gestión de objetos con SQL. Especificaciones en estándares SQL

Los objetos han conseguido entrar en el mundo de las bases de datos en forma de dominios, actuando como el tipo de datos de una columna determinada.

A continuación, se detallan dos **implicaciones** muy importantes que se producen por el hecho de utilizar una clase como un dominio:

- **Es posible almacenar múltiples valores en una columna de una misma fila, ya que un objeto suele contener múltiples valores**. Sin embargo, si se utiliza una clase como dominio de una columna, en cada fila esa columna solo puede contener un objeto de la clase (se sigue manteniendo la restricción del modelo relacional de contener valores atómicos en la intersección de cada fila con cada columna).

- **Es posible almacenar procedimientos en las relaciones** porque un objeto está enlazado con el código de los procesos que sabe realizar (los métodos de su clase).

Otro modo de incorporar objetos en las bases de datos relacionales es **construyendo tablas de objetos**, donde cada fila es un objeto.

Un sistema objeto–relacional es un sistema relacional que **permite almacenar objetos en sus tablas**. La base de datos sigue sujeta a las restricciones que se aplican a todas las bases de datos relacionales y conserva la capacidad de utilizar operaciones de concatenación, ***JOIN***, para implementar las relaciones “al vuelo”.

Los usuarios pueden definir sus propios tipos de datos, a partir de los tipos básicos provistos por el sistema o por otros tipos de datos predefinidos anteriormente por el usuario. Estos tipos de datos pueden pertenecer a dos categorías distintas: **objetos** y **colecciones**.

Definición de tipos de objetos

En Oracle los tipos de objetos son **aquellos tipos de datos que han sido definidos por el usuario**. Los objetos se presentan de forma abstracta, pero realmente se siguen almacenando en columnas y tablas.

Para poder crear tipos de objetos se debe hacer uso de la ***sentencia CREATE TYPE***. Está compuesta por los siguientes **elementos**:

- Para identificar el tipo de objetos se utiliza un **nombre**.
- Existen unos **atributos** que pueden ser de un tipo de datos básico o de un tipo definido por el usuario, que representan la estructura y los valores de los datos de ese tipo.
- Unos **métodos** que son procedimientos o funciones. Pueden ser de varios **tipos**:
  - **MEMBER**: son métodos que actúan con objetos. Serán procedimientos o funciones.
  - **STATIC**: métodos estáticos independientes de las instancias del objeto. Pueden ser procedimientos o funciones.
  - **CONSTRUCTOR**: inicializan el objeto. Tienen como valores los atributos del objeto y que lo devuelve inicializado.

EJEMPLO CÓDIGO :

```
CREATE OR REPLACE TYPE persona AS OBJECT (
nombre VARCHAR2(30);
telefono VARCHAR2(20)
);
```

Se puede observar en el ejemplo que cada objeto/persona tendrá dos atributos, el **nombre y el teléfono**.

Se muestra, a continuación, la **definición de un tipo objeto**:

CÓDIGO :

```
CREATE or REPLACE TYPE nombreObjeto AS OBJECT (
  atributo TIPO,
  atributo2 TIPO,
  ...,
  MEMBER FUNCTION nombreFuncion RETURN Tipo,
  MEMBER FUNCTION nombreFuncion2(nomVar TIPO) RETURN Tipo,
  MEMBER PROCEDURE nombreProcedimiento
  PRAGMA RESTRICT_REFERENCES (nombreFuncion,restricciones),
  PRAGMA RESTRICT REFERENCES (nombreFuncion2,restricciones),
  PRAGMA RESTRICT_REFERENCES (nombreProcedimiento,restricciones)
);
```

Donde **restricciones** pueden ser cualquiera de las siguientes o incluso una combinación de ellas:

- **WINDS**: evita que el método pueda modificar las tablas de la base de datos.
- **RNDS**: evita que el método pueda leer las tablas de la base de datos.
- **WNPS**: evita que el método modifique variables del paquete PL/SQL.
- **RNPS**: evita que el método pueda leer las variables del paquete PL/SQL.

Definición del cuerpo de un objeto:

CÓDIGO :

```
CREATE OR REPLACE TYPE BODY nombreObjeto AS
  MEMBER FUNCTION nombreFuncion
  RETURN Tipo
  IS
    BEGIN
      ...
  END nombreFuncion;
  MEMBER FUNCTION nombreFuncion2(nomVar TIPO)
  RETURN Tipo
  IS
    BEGIN
      ...
  END nombreFuncion2;
  MEMBER PROCEDURE nombreProcedimiento
  IS BEGIN
      ...
  END nombreProcedimiento;
END; /
```

Por ejemplo:

EJEMPLO CÓDIGO - DEFINICIÓN CLASE :

```
CREATE or REPLACE TYPE persona AS OBJECT (
   idpersona number,
   dni varchar2(9),
   nombre varchar2(15),
   apellidos varchar2(30),
   fecha_nac date,
   MEMBER FUNCTION muestraEdad RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES (muestraEdad,WNDS)
);
```

EJEMPLO CÓDIGO – CUERPO DE LA CLASE :

```
CREATE OR REPLACE TYPE BODY persona AS
   MEMBER FUNCTION muestraEdad
   RETURN NUMBER
   IS
      BEGIN
      return to_char(sysdate, 'YYYY')-to_char(fecha_nac, 'YYYY');
   END muestraEdad;
END;
```

Una vez definida la clase y el cuerpo, se puede utilizar este objeto como si de un tipo cualquiera se tratase.

CÓDIGO

```
DECLARE
  Trabajador1 Persona;
Begin

    Trabajador1:= NEW Persona(1, '123456', 'Alberto', 'Oliva', '22/12/1989');

    DBMS_OUTPUT.put_line('El empleado '||Trabajador1.nombre|| ' ha sido creado');
END;
```

Colecciones

Para poder contar con atributos multivaluados es necesario que se organicen en una estructura de datos, conocida como ***Array*** o **Colección**. Para crear un atributo de tipo array se debe utilizar la siguiente sintaxis.

CÓDIGO:

```
-- Lista de un tipo primitivo varchar2
create type colec_nombres as varray(10) of varchar2(20);


-- Lista de un tipo de usuario: Persona
create type colec_personas as varray(10) of Persona;
```

Una vez que ya se ha creado un tipo de colección se puede utilizar dentro de la definición de una clase:

CÓDIGO:

```
Create or Replace Type Departamento as Object(
  NombreDept Varchar2(100),
  Empleados colec_personas
);
```

Tablas derivadas de objetos

De la misma forma que se crean objetos haciendo uso de los tipos definidos, puede hacerse también con las tablas:

CÓDIGO:

```
create table Empleados(
  NombreDept Varchar2(50),
  Datos_Empleado Persona
);
```

Para hacer un ***insert***, se debe indicar el tipo de objeto que se está insertando:

CÓDIGO:

```
insert into Empleados
values ('Informática', Persona(1, '1111', 'Paco', 'Gonzalez', '23/12/1988'));
```

3.3. Acceso a las funciones del gestor desde el lenguaje de programación.

Al igual que las bases de datos relacionales disponen de SGBD relacionales, las bases de datos orientadas a objetos también tienen. Dentro de los SGBD-OO bajo licencias de software libre se puede destacar Matisse o NeoDatis ODB.

Mattisse puede ser utilizado con los lenguajes como Java, C o .NET y está orientado a trabajar con un mayor número de datos y una compleja estructura. NeoDatis ODB trabaja también con los lenguajes Java y .Net (entre otros), soporta consultas nativas y es bastante simple e intuitivo.

Acceso desde Java

En NeoDatis ODB, la conexión y desconexión se ha de realizar utilizando la clase ODB.

EJEMPLO CÓDIGO:

```
ODB odb = ODBFactory.open("C:/Unidad3/Servidor/neodatis.test");
odb.close();
```

Para la inserción de objetos será necesario disponer de una clase creada con sus atributos, constructor, métodos getters y setters, y hacer uso del método store. Ejemplo suponiendo la creación de un objeto tipo Libro:

EJEMPLO CÓDIGO:

```
Libro libro= new Libro ("Alicia en el país de las maravillas", "Madrid", 144);
odb.store(libro);
```

Aunque ahora no se va a profundizar en ello, la clase OBD dispone de más métodos que son los que se han de utilizar para realizar consultas y modificaciones en la base de datos, como por ejemplo: ***getObject()*** o ***delete()***.

Modelo cliente/servidor de una base de datos

NeoDatis ODB puede utilizarse como una BD cliente/servidor en la cual el cliente y el servidor pueden estar tanto en la misma máquina como en máquinas diferentes.

Se iniciará el servidor configurando los siguientes **parámetros**:

- El puerto en el que se ejecutarán y recibirán las conexiones de los clientes debe estar libre y se usará el método **openServer(puerto)**.

EJEMPLO CÓDIGO:

```
ODBServer server = ODBFactory.openServer(8000);
```

- La base de datos que será manejada por el servidor. Se utilizará el método **addBase()** donde se especifica el nombre y el archivo de base de datos.

*Ejemplo con fichero de base de datos **neodatisServidor.test**, que se encuentra en la carpeta **C:/Unidad3/Servidor**. Para dirigirse a ella se escribiría:*

EJEMPLO CÓDIGO:

```
server.addBase("base", "C:/Unidad3/Servidor/neodatisServidor.test");
```

- Se inicia el servidor con un subproceso en segundo plano a través del método **StartServer(true)**. Se escribiría:

EJEMPLO CÓDIGO:

```
Server.startServer(true);
```

A continuación, se creará el cliente con el método **openClient()**. Existen dos formas para hacerlo:

1) Si está en la misma máquina:

EJEMPLO CÓDIGO:

```
ODB odb = server.openClient("base");
```

2) Si está en otra máquina, se debe utilizar **ODBFactory**:

EJEMPLO CÓDIGO:

```
ODB odb = ODBFactory.openClient("localhost", 8000, "base");
```

Para poder usar el cliente/servidor se debe añadir el paquete **org.neodatis.odb.ODBServer**. También hay que tener en cuenta que las bases de datos que se implementen deberán ser **Serializable**.

Para asegurarse de que se cierra la base de datos es conveniente usar el bloque **try/finally** (con él se cierra el cliente y el servidor).

3.4. Características de las bases de datos orientadas a objetos

Las **características** que estarán asociadas a unas **BDOO** son las siguientes:

- El almacenamiento de datos se hace como **objetos**.
- Estos objetos deben ser identificados mediante un id único, también denominado **OID,** que no podrá ser modificado.
- En cada objeto se debe indicar qué métodos y atributos tiene, además de indicar la interfaz por la que se puede acceder a él.
- Básicamente tiene que tener las características de un SGBD y de un sistema OO.

El **manifiesto *Malcolm Atkinson*** creado en 1989 define trece **características** obligatorias basadas en dos criterios: **SGBDOO y SGBD**.

1) Se deben soportar **objetos complejos**.

2) Se deben soportar **mecanismos de identidad** de los objetos.

3) Se debe soportar la **encapsulación**.

4) Se deben soportar los **tipos o clases**.

5) Los tipos o clases deben ser capaces de **heredar de sus ancestros**.

6) Se debe soportar el **enlace dinámico**.

7) El DML debe ser **complejo**.

8) El conjunto de todos los tipos de datos debe ser **extensible**.

Características **obligatorias** de SGBD:

9) Se debe proporcionar **persistencia** a los datos.

10) El SGBD debe ser capaz de gestionar **bases de datos muy grandes**.

11) El SGBD debe soportar **concurrencia**.

12) El SGBD debe ser capaz de recuperarse de **fallos de hardware y software.**

13) El SGBD debe proporcionar una forma sencilla de **consultar** los datos.

Ventajas e inconvenientes de este sistema:

Ventajas	Inconvenientes
Gran capacidad de modelado	Falta modelo de datos universal
Muy extensible	Falta experiencia, el uso es limitado
Única interfaz entre lenguaje de manipulación de datos y el lenguaje de programación	Faltan estándares
Consultas más expresivas	Tiene competencia en los SGBDR y los SGBDOR
Da soporte a transacciones largas	Optimizar las consultas compromete la encapsulación
Se adecúa a aplicaciones avanzadas	Es muy complejo
	Falta soporte a las vistas
	Falta soporte a la seguridad

3.5. Tipos de datos: básicos y estructurados

Los **tipos de datos** pueden ser: atómicos, estructurados y colecciones.

- **Atómicos**: como *boolean, short, long, unsigned long, unsigned short, float, doublé, char, string, enum, octect*.

- **Tipos estructurados**: como *date, time, timestamp, Interval*.

- **Colecciones** *<interfaceCollection>*:
  - *set<tipo>*: grupo desordenado de objetos que no admite duplicados.
  - *bag<tipo>*: grupo desordenado de objetos que permite duplicados.
  - *list<tipo>*: grupo ordenado de objetos que permite duplicados.
  - *array<tipo>*: grupo ordenado de objetos a cuya posición se puede acceder. Son de tamaño dinámico.
  - *dictionary<clave,valor>*: grupo de objetos del mismo tipo, con cada valor asociado a una clave.

A través de las **clases** se pueden establecer el estado y el comportamiento de cada objeto. Además, son instanciables, es decir, se pueden crear instancias de objetos individuales.

El lenguaje ODL especifica los atributos, las relaciones entre los tipos y especifica la signatura de las operaciones. Algunas palabras reservadas son:

- **class**: declara el objeto.
- **extent**: define la extensión.
- **key[s]**: declara la lista de claves.
- **attribute**: declara un atributo.
- **struct**: declara un tipo estructurado.
- **enum:** declara un tipo enumerado.
- **relationship:** declara una relación.
- **inverse:** declara una relación inversa.
- **extends:** define la herencia simple.

El lenguaje de consultas OQL

OQL es el lenguaje de consultas estándar para las BDOO. Sus **características** principales son:

- Es OO y está basado en el modelo ODMG.
- Lenguaje declarativo del tipo SQL. Sintaxis básica similar al SQL.
- Acceso declarativo a los objetos (propiedades y métodos).
- Posee una semántica bien definida.
- No incluye operaciones de actualización, solo consultas. Para modificarlo se realiza a través de métodos que poseen los objetos.
- Posee operadores sobre colecciones y cuantificadores.

La **sintaxis** básica será de un **SELECT** similar a SQL:

- *SELECT <valores>.*
- *FROM <colecciones y miembros típicos>.*
- ***[****WHERE <condición>].*

En **FROM** las colecciones pueden ser extensiones. Es común utilizar una variable para que vaya tomando valores de los objetos de la colección. Se puede especificar mediante **IN** o **AS**

- *FROM Archivos x*
- *FROM x IN Archivos*
- *FROM Archivos AS x*

Para acceder a los atributos y objetos es necesario hacerlo mediante las expresiones de **camino**. Comenzarán con el nombre del objeto, seguido de atributos que están conectados a través de un punto o nombres de relaciones.

Operadores de comparación

Los valores podrán ser comparados a través de los siguientes operadores:

- = Igual a
- > Mayor que
- >= Mayor o igual que
- < Menor que
- <= Menor o igual que
- != Distinto de

Para comparar cadenas se pueden usar:

- **IN**: comprueba si existe un carácter en una cadena.
  Sintaxis: *carácter IN cadena*
- **LIKE**: compara si dos cadenas son iguales.
  Sintaxis: *cadena1 LIKE cadena2*
- **_ o ?**: indica posición, representa cualquier carácter.
- *** o %:** representa cadena de caracteres.

Cuantificadores y operadores sobre colecciones

A través del uso de cuantificadores es posible corroborar si todos los miembros, o al menos uno de ellos, cumplen alguna condición:

- Todos los miembros: *FOR ALL x IN colección: condición.*
- Al menos uno: *EXISTS x IN colección: condición EXISTS x.*
- Solo uno: *UNIQUE x.*
- Algunos/cualquier: *colección comparación SOME/ANY condición.*

Los operadores **AVG, SUM, MIN, MAX y COUNT** pueden aplicarse en cualquier colección, siempre y cuando tengan sentido en el elemento.

3.6. La interfaz de programación de aplicaciones de la base de datos

Existe una API sencilla que no requiere mapeo. Se trata de **NeoDatis Object Database**. Es una base de datos orientada a objetos de código abierto que funciona con Java, .Net, Groovy y Android.

Para poder usarla hay que descargarla desde la **web**:

http://neodatis.wikidot.com/downloads

Una vez descargado el fichero comprimido, se copia el fichero **.jar** a la carpeta del proyecto para poder definirlo. Desde la **carpeta /doc/javadoc** se accede a la documentación de la API de ***Neodatis ODB*.**

Como vimos anteriormente, para abrir la base de datos se utiliza la clase **ODBFactory** con el método **open()** que devolverá un **ODB (interfaz principal).**

Para **almacenar** objetos se utilizará el método **store()**.

Una vez guardados se pueden **recuperar** a través del método **getObjects()**.

Para **validar** los cambios y cerrar la conexión se usará el método **close()**.

UF3. PERSISTENCIA EN BD NATIVAS XML

En esta Unidad Formativa se desarrollan las aplicaciones que gestionan la información almacenada en bases de datos nativas XML, evaluando y utilizando clases específicas.

1. Bases de datos XML

En esta unidad existen varios objetivos y, el alumno, al término de la misma, tendrá que haber sido capaz de poder realizar bases de datos que no usan el lenguaje SQL para realizar las consultas, ni estructuras fijas de almacenamiento.

Será capaz de consultar informaciones y crear, modificar y eliminar datos con nuevos lenguajes como pueden ser **XML o JSON**.

Contenido principal en esta unidad:

- Bases de datos nativas XML.
- Estrategias de almacenamiento.
- Establecimiento y cierre de conexiones.
- Colecciones y documentos.
- Creación y borrado de colecciones, clases y métodos.
- Añadir, modificar y eliminar documentos; clases y métodos.
- Realización de consultas, clases y métodos.
- Tratamiento de excepciones.

1.1. Bases de datos nativas XML

Las bases de datos basadas en XML se diferencian del resto en que su unidad mínima será el mismo documento XML, por lo que no poseerá campos ni almacenará datos. Lo único que almacena son los documentos XML. Es un sistema de gestión que cumple con el siguiente **procedimiento**:

Se define un modelo lógico para un documento XML y se almacenan y recuperan los documentos según este modelo. Existe una relación con la forma de almacenamiento en la que incorpora las características ASID de cualquier SGDB. Posee varios niveles de datos y una alta complejidad. Permite el uso de tecnologías de consulta y transformación propias de XML, XQuery, XPath, XSLT, para acceder y tratar la información.

Ventajas e inconvenientes de estas bases de datos:

Ventajas	Inconvenientes
Acceso y almacenamiento directamente en formato XML	Complicación al indexar documentos para realizar búsquedas
Motor de búsqueda de alto rendimiento	No ofrece funciones de agregación
Sencillez al añadir nuevos documentos XML	Almacenamiento como documento o nodo; resulta complicado formar nuevas estructuras
Datos heterogéneos	

1.2. Estrategias de almacenamiento

Para almacenar documentos XML, tanto los que están centrados en datos como los centrados en contenido, existen varias formas:

1. Directo al fichero.
2. En una base de datos (SGBD relacional).
3. En una base de datos XML.

Almacenamiento directo a ficheros

No se pueden realizar muchas funciones en ellos ya que las operaciones no se pueden realizar sobre el contenido y están definidas por el sistema.

Almacenamiento sobre una base de datos

En los tipos de documentos centrados en datos se puede realizar un mapeo entre los datos del documento y los del SGBD.

Esto es debido a que su estructura es regular y bien controlada, por lo que puede transformarse en un esquema relacional, aunque posee el inconveniente de que solo se pueden almacenar datos que interesa conservar, por lo que cuando se quiera reconstruir el documento se creará otro distinto.

Hay **tres tipos de almacenamiento** sobre una base de datos:

1. **Directamente sobre un campo:** usar este método tiene la ventaja de que el formato se mantendrá, pero no se podrá consultar nada en el documento.
2. **Mapeo basado en tablas**: los datos en este mapeo serán vistos como filas y estarán agrupados en forma de tablas. Además, cada fila se corresponderá con una tupla y cada dato contendrá un campo de esa tabla.
3. **Mapeo basado en objetos**: los datos serán tratados como objetos serializados que irán a una tabla y las propiedades irán a las columnas.

Almacenamiento sobre una base de datos XML

En el caso de que se utilice un documento basado en el contenido, será irregular y no será posible controlar las posibles estructuras y realizar distintos mapeos, además de que habrá que obtener el mismo documento almacenado, por lo que la solución más factible es la utilización de un SGBD XML nativo.

1.3. Establecimiento y cierre de conexiones

eXist es un SGBD libre de código abierto en el que se almacenan datos XML con un motor de base de datos escrito en Java y que es capaz de soportar los estándares de consulta XPath, XQuery y XSLT, además de indexar documentos. Asimismo, sirve para una gran cantidad de protocolos.

En el Sistema de Gestión de Bases de Datos existen aplicaciones capaces de realizar consultas directamentes sobre las BD.

Los documentos XML se almacenan en colecciones y, además, estas podrán estar anidadas. Estas colecciones serían carpetas y los documentos XML estarían dentro de cada una de ellas, pudiéndose almacenar cualquier tipo de documento.

Existe además una ruta en la que se almacenarán los archivos más importantes de la BD y será **eXist\webapp\WEB-INF\data**. Entre esos archivos destacan:

1. **dom.dbx**: es el almacén central de datos. En él se almacenan todos los nodos del documento.
2. **collections.dbx**: en él se almacena la jerarquía de las colecciones y, estas, se relacionarán con los documentos que contienen, además de incluir un *id* único a cada documento.

Instalación de eXist

Es posible instalarlo de **varias maneras**:

- Como servidor autónomo.
- Dentro de una aplicación Java.
- En un servidor J2EE.

Para descargar la última versión de la BD es posible acceder a la siguiente url: http://exist-db.org/exist/apps/homepage/index.html. En este caso, se ha descargado la versión **eXist-db-setup-3.4.0.jar**. Para proceder a la instalación, una vez descargado el archivo de instalación hay que ejecutarlo desde la línea de comandos (**java -jar eXist-db-setup-3.4.0.jar**) o haciendo doble clic sobre el icono correspondiente y siguiendo el asistente.

El proceso de instalación a través del asistente es muy sencillo, solo hay que prestar atención al directorio de instalación y a la hora de poner el password del administrador. En este caso se puede poner **admin**.

Una vez instalado, habrá que acceder a la base de datos a través del acceso directo creado en el escritorio (si así se hubiera indicado), o desde el menú de la aplicación donde se seleccionará **eXist-db XML Database**. También es posible lanzar el archivo **start.jar** que se sitúa en la carpeta de **eXist**.

Puede darse el caso de que no se inicie la BD al lanzarla y muestre una pantalla con mensajes de error. Este problema surge cuando se quiere conectar la BD, ya que utiliza el puerto 8080, como la mayoría de servidores web. Dicho puerto está ocupado por otra aplicación.

Para solucionar esto es imprescindible cambiar el puerto. Para ello será necesario editar el archivo de configuración **%HOME_EXIST%/tools/jetty/etc/jetty.xml**, en la etiqueta **<Call name="addConnector">** cambiar el puerto y en la propiedad **SystemProperty** de la etiqueta **<Set name="port">** escribir el nuevo valor, por ejemplo 8084.

Quedaría de la siguiente manera:

EJEMPLO CÓDIGO:

```
<SystemProperty name="jetty.port" default="8080"/>
<SystemProperty name="jetty.port" default="8084"/>
```

Cuando se lance la BD se creará también un icono en la barra de tareas del sistema desde el que se pueden lanzar varias herramientas e iniciar el servidor.

Primeros pasos con eXist

Si se ha arrancado la base de datos, se podrá iniciar directamente desde el navegador escribiendo en la barra de direcciones:

http://localhost:8084/exist/

A la hora de realizar consultas y de trabajar con las bases de datos es posible hacerlo de varias formas:

- **Desde el eXide (Open eXide):** es una herramienta desde la que se pueden realizar consultas a documentos de la BD, cargarlos y crear o buscar colecciones, entre otros.

  Para acceder al eXide, puede hacerse desde la url: **http://localhost:8084/exist/apps/eXide/index.html.**

  Para navegar por las carpetas y documentos de la BD, es posible hacerlo desde la pestaña **directory**, y dentro de **/db/apps/demo/data** se encontrarán algunos archivos XML. Para el uso de operaciones se hará desde la barra de herramientas.

- **Desde el *dashboard* (Open dashboard):** el cuadro de instrumentos es el administrador de aplicaciones de la BD. Es capaz de soportar plugins y las aplicaciones poseen su propia interfaz gráfica, mientras que los plugins se ejecutan dentro del dashboard como diálogos.

- **Desde el cliente java (Open Java Admin Client):** es el que se utiliza en este temario para realizar las consultas. Para la conexión será necesario:

  a) El driver de eXist:

CÓDIGO:

```
Driver = “org.eXist.xmldb.DatabaseImpl”.
```

b) Cargar el driver en memoria:

CÓDIGO:

```
c = Class.forName(driver);
```

c) Crear una instancia de la base de datos:

CÓDIGO:

```
Database database = (DataBase) c.newInstance();
```

d) Registrar la instancia:

CÓDIGO:

```
DatabaseManager.registerDatabase(database);
```

e) Acceder a la colección. Para ello necesitaremos la URI, colección deseada, usuario y clave de acceso:

CÓDIGO:

```
URI = "xmldb:eXist://localhost:8080/exist/xmlrpc"
coleccion = "/db/Libros/Esp"
usuario = "admin";
usuarioPwd = "admin";
```

El acceso a la conexión sería:

CÓDIGO:

```
Collection collect = DatabaseManager.getCollection(URI coleccion, usuario, usuarioPwd);
```

Esto permite obtener en la variable collect la colección deseada (/db/Libros/Esp) que está situada en la base de datos eXist con la ruta de acceso: xmldb:eXist://localhost:8080/exist/xmlrpc, usuario y contraseña admin/admin.

El cliente de administración de eXist

A través del cliente de administración de *eXist*, que se ejecutará una vez conectado el usuario, se podrán realizar distintas **operaciones,** tales como:

1. Creación y borrado de colecciones.
2. Incluir y eliminar documentos a las colecciones.
3. Modificar documentos.
4. Creación de copias de seguridad y restauración de las mismas.
5. Administración de usuarios.
6. Realización de consultas Xpath.
7. Navegar por colecciones y escoger un contexto a través del cual se ejecutarán las consultas.

Al pulsar sobre el icono de los prismáticos se abrirá una ventana llamada ***Diálogo de consulta***, desde la cual se pueden realizar consultas en la parte superior y, tras pulsar en el botón ***Ejecutar***, aparecerá el resultado de dicha consulta en la parte inferior. Las consultas realizadas pueden ser guardadas en archivos externos.

1.4. Colecciones y documentos.

En una colección se engloban un conjunto de documentos XML que forman una estructura en árbol. En esta estructura en árbol cada recurso o documento pertenece a una colección. Las colecciones puedes ser vistas como carpetas en las cuales se almacenan recursos.

La estructura en árbol permite establecer caminos que referencien los recursos a los que se quiere acceder. Para acceder a los datos de estos recursos se trabajarán con dos tipos de estándares que permiten acceder y obtener datos desde documentos XML: **XPath** y **XQuery**.

XPath es un lenguaje basado en rutas de XML y se utiliza para navegar de forma jerárquica en un XML.

XQuery es un lenguaje de consulta de documentos XML. Abarca tanto ficheros XML como base de datos relacionales con funciones para obtener XML a partir de registros. XQuery contiene a XPath.

XQuery tiene almacenada en sí misma a **XPath** por lo que cualquier consulta en XPath es válida también en XQuery; adicionalmente, XQuery permite mayor funcionalidad.

1.5. Creación y borrado de colecciones, clases y métodos.

Una colección es un conjunto de documentos XML que forman una estructura en árbol.

Colecciones con la administración de eXist

Partiendo de la existencia de un fichero XML que sea el que contiene los datos, para subir una colección desde el cliente eXist a la base de datos se puede hacer desde el usuario Admin.

El objetivo es crear una colección de documentos XML que sirva para realizar posteriores consultas. Para ello se ha de hacer clic en Browse Collections y, posteriormente, al hacer clic en Create Collection para crear la colección añadiendo el nombre correspondiente. Al entrar en esta nueva colección creada ya se podrá subir los documentos XML que hayan de ir dentro de esta colección. Estos documentos deben estar bien formados para que no se produzcan errores en la subida de los mismos.

Colecciones con Java

Para crear colecciones usando el lenguaje de programación Java se debe hacer uso principalmente de dos clases: ***Collection*** y **CollectionManagementService**. Ambas clases pertenecen a la librería xmldb.jar. Se deberá añadir esta librería al proyecto para poder usar las clases.

En la clase **Collection**, el método *getService()* (que recibe dos parámetros: nombre y versión), devuelve una instancia de un servicio dependiendo de los datos introducidos como parámetros.

En la clase **CollectionManagementService** se usará el método *createCollection()* que recibe un nombre. Este método crea una colección en la base de datos con el nombre que se le pasa como parámetro.

EJEMPLO CÓDIGO:

```
CollectionManagementService servicio = (CollectionManagementService)
            contexto.getService("CollectionManagementService", "1.0");
Collection nuevaColeccion = servicio.createCollection("Libros");
```

Para la eliminación de colecciones se debe usar el método *removeCollection()* de la clase CollectionManagementService al que se le pasa el nombre de la colección como parámetro:

EJEMPLO CÓDIGO:

```
servicio.removeCollection("Libros");
```

1.6. Añadir, modificar y eliminar documentos; clases y métodos.

Para el uso de los documentos o recursos a través del código Java se utilizarán varios métodos pertenecientes a la clase anteriormente vista Collection:

- ***createResource(idRecurso, tipoRecurso):*** Crea en la colección correspondiente un nuevo recurso con el idRecurso y tipo que se le pasan como parámetro.
- ***storeResource(recurso):*** Almacena en la colección el recurso que se le pasa como parámetro.
- ***removeResource(recurso):*** Elimina el recurso que se le pasa como parámetro de la colección.

EJEMPLO CÓDIGO:

```
1 Resource nuevoRecurso = contexto.createResource(archivo.getName(),
                                                   "XMLResource");
2 nuevoRecurso.setContent(archivo);
3 contexto.storeResource(nuevoRecurso);
```

En este código, primero se crea el recurso vacío asignándole el nombre del archivo (tipo File) que contiene los datos. Posteriormente, en la línea número 2, mediante el método *setContent* se le incorpora el contenido del archivo al nuevo recurso. Por último, en la línea 3 se almacena el recurso.

De forma adicional a estos métodos, existen otros métodos que pueden ser útiles para trabajar con recursos. Algunos de ellos:

- ***listResources():*** Devuelve todos los ids de los recursos que tiene la colección en forma de array de String.
- ***gerResourceCount():*** Devuelve el número de recursos que tiene una colección.

Para la eliminación de recursos debemos usar, en la clase Collection, el método *removeResource()* al que se le debe pasar el rescurso a eliminar como parámetro. Antes de la eliminación del recurso necesitaremos disponer de él. Para ello se puede usar el método *getResource(idRecurso)*, que debe recibir el id del recurso como parámetro.

EJEMPLO CÓDIGO:

```
Resource recursoAEliminar = contexto.getResource("00001");
contexto.removeResource(recursoAEliminar);
```

1.7. Realización de consultas, clases y métodos.

Para poder trabajar con los datos de los recursos se utilizarán los dos tipos de estándares que vimos anteriormente: **XPath** y **XQuery**. Estos estándares permiten realizar consultas de sistemas XML nativos.

Cabe destacar que XQuery es el estándar propuesto por la W3C (World Wide Web Consortium) para la realización de consultas y procesamiento de datos XML. Esto se debe a que XQuery es una extensión de XPath, soportando sus consultas, pero potenciando sus posibilidades.

Expresiones XPath

XPath es un lenguaje que permite seleccionar nodos de un documento XML y también calcular valores a partir de su contenido. Se basa en la selección arbórea del documento.

Un documento XML está formado por distintos **tipos de nodos**:

Tipo de nodo	Descripción	Representación/Ejemplo
Raíz	Raíz del árbol	/
Elemento	Cualquier elemento (etiquetas) del árbol	<identificador> <departamento> <libro>
Texto	Caracteres entre las etiquetas	ID02154, Informática, Análisis

Atributo	Propiedades añadidas a los elementos	Se representa con @. tipo=”A” color=”azul”
Comentario	Etiquetas de comentario	<!-- Libro 1 --> <!-- Departamento 1 -->
Espacio de nombres	Contiene espacio de nombres	<espacio xmlns=”http://www.mipagina.com” />
Instrucción de proceso	Instrucciones de proceso	Se representan con: <?.....?> <?xml versión=”1.0” encoding=”ISO-8859-1”?>

EJEMPLO CÓDIGO XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<universidad>
<espacio xmlns="http://www.estoesunejemplo.com/prueba" />
   <!--DEPARTAMENTO 1-->
   <departamento tipo=”A” color=”azul”>
      <nombre>Informática</nombre>
      <identificador>0001</identificador>
   </departamento>
   <!--DEPARTAMENTO 2-->
   <departamento tipo=”A” color=”rojo”>
      <nombre>Matemáticas</nombre>
      <identificador>0002</identificador>
   </departamento>
</universidad>
```

Sobre cada nodo es posible realizar una serie de acciones:

Acción	Descripción
/nombreDelNodo	Para seleccionar un nodo concreto. Ej: /universidad
prefix:*	Selecciona nodos un espacio de nombres determinado.
text()	Selecciona el contenido del elemento. Ej: //nombre/text()
node()	Selecciona todos los nodos: los elementos y su texto. Ej: /universidad/node()
processing-instruction(),	Selecciona los nodos que son instrucciones de proceso.
Comment()	Selecciona los nodos de tipo comentario. Ej: /universidad/comment()

Al utilizar la sintaxis de XPath se usarán **descriptores de ruta o de camino** que sirven para seleccionar nodos o elementos situados en ciertas rutas. Cada descriptor se puede dividir en **tres partes**:

1. **Eje**, que indicará el nodo o nodos en los que realizar la búsqueda.
2. **Nodo de comprobación**, seleccionados dentro del eje.
3. **Predicado**, que permite restringir nodos, además de representarse entre corchetes.

Para escribir la ruta en XPath se puede hacer de dos formas: **abreviada y compleja.** En este tema se utilizará solamente la sintaxis abreviada, por lo que los descriptores se formarán nombrando la etiqueta separada por */.*

Si comienza con / significa que la ruta comienza **desde la raíz** y serán **rutas absolutas**. En caso contrario serán rutas relativas. En el caso de empezar con // puede comenzar desde cualquier parte de la colección.

Listado de órdenes al realizar consultas

Orden	Descripción
/	Devuelve todos los datos de la colección.
Node()	Devuelve las etiquetas dentro de esa carpeta sin incluirla.
Text()	Devuelve el texto dentro de la carpeta.
*****	Nombra a cualquier etiqueta.
[], (<, >, <=, >=, =, !=, or, and y not)	Se usarán corchetes para seleccionar elementos concretos y podrán usarse comparadores.
[1]	Número entre corchetes. Posición del elemento en el conjunto.
Last()	Último elemento del conjunto seleccionado.

Position()	Devuelve número igual a posición del elemento.
Count()	Cuenta número elementos seleccionados.
Sum()	Devuelve suma del elemento seleccionado (si la etiqueta la considera String hay que convertirla con la función number).
Max(), min(), avg()	Devuelve el máximo, el mínimo y la media del elemento seleccionado.
Name()	Devuelve el nombre del elemento seleccionado.
Concat(cad1, cad2…)	Concatena cadenas.
Start-with(cad1,cad2)	Verdadera cuando la cad1 tiene como prefijo a cad2.
Contains(cad1,cad2)	Verdadera cuando la cad1 contiene a la cad2.
String-lenght(argumento)	Devuelve número de caracteres de su argumento.
Div()	Realiza divisiones.
Mod()	Calcula resto de la división.

Data(expresión XPath)	Devuelve texto de la expresión sin etiquetas.
Number(argumento)	Convierte a número el argumento, que puede ser cadena, booleano o nodo.
Abs(num)	Devuelve valor absoluto del número.
Ceiling(num)	Devuelve entero más pequeño mayor o igual que expresión numérica.
Floor(num)	Devuelve entero más grande menor o igual que expresión numerica.
Round(num)	Redondea valor de la expresión numérica.
String(argumento)	Convierte argumento en cadena.
Compare(exp1, exp2)	Compara dos expresiones, devuelve 0 si son iguales, 1 si exp1>exp2 y -1 si exp1<exp2.
Substring(cadena,comienzo,num)	Extrae de la cadena, desde el comienzo, el número de caracteres indicados en num.
Substring(cadena,comienzo)	Extrae de la cadena, desde el comienzo hasta el final.
Lower-case(cadena)	Convierte a minúscula la cadena.

Upper-case(cadena)	Convierte a mayúscula la cadena.
Translate(cadena1,caract1,caract2)	Reemplaza en la cadena1, los caracteres seleccionados en caract1 por los correspondientes en caract2.
Ends-with(cadena1,cadena2)	Devuelve true si cadena1 termina en cadena2.
Year-from-date(fecha)	Devuelve año de la fecha (año-mes-día).
Month-from-date(fecha)	Devuelve mes de la fecha.
Day-from-date(fecha)	Deuvelve día de la fecha.
Para encontrar más funciones visitar: http://www.w3schools.com/xsl/xsl_functions.asp	

Nodos atributos XPath

Cuando se elaboran los nodos es posible crear tantos atributos como se desee en cada uno de ellos. Estos no se consideran como hijos de los nodos sino como etiquetas de cada uno de ellos en las que constará un nombre, un valor de tipo String y un posible espacio de nombres.

Para referirse a los atributos de los elementos se utiliza **@** antes del nombre.

Axis XPath

Un Axis especifica la dirección que se va a evaluar, es decir, si se va a mover de arriba hacia abajo o viceversa; o si se incluye el nodo actual o no.

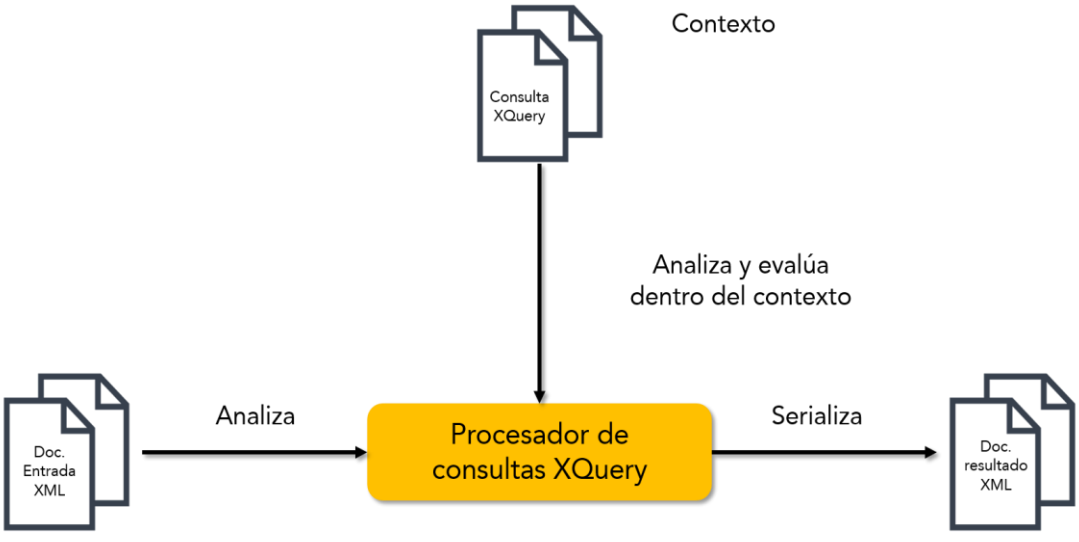
NOMBRE DE AXIS	RESULTADO
ancestor	Selecciona los antepasados (padres, abuelos, etc.) del nodo actual.
ancestor-or-self	Selecciona los antepasados (padres, abuelos, etc.) del nodo actual y este actúa en sí.
attribute	Selecciona los atributos del nodo actual.
child	Selecciona los hijos del nodo actual.
descendant	Selecciona los descendientes (hijos y nietos) del nodo actual.
descendat-or-self	Selecciona los descendientes (hijos y nietos) del nodo actual y este actúa en sí.
following	Selecciona todo el documento después de la etiqueta de cierre del nodo actual.

following-sibling	Selecciona todos los hermanos que siguen al nodo actual.
parent	Selecciona el padre del nodo actual.
self	Selecciona el nodo actual.

La sintaxis para utilizar ejes es: ***Nombre_de_eje::nombre_nodo[expresión].***

Consultas XQuery

Una consulta en el lenguaje XQuery es una expresión capaz de leer datos de uno o más documentos en XML y devuelve otra secuencia de datos en XML.



Como XQuery contiene a XPath, toda consulta realizada en XPath es posible realizarla en XQuery.

XQuery permite seleccionar información basada en ciertos tipos de criterios, buscar información en uno o más documentos, unir varios datos de distintos documentos o

colecciones, organizar, unir y resumir datos, transformar y reestructurar datos XML en otro vocabulario, realizar cálculos sobre números y fechas y manejar cadenas de caracteres a formato de texto.

Las **consultas** realizadas en XQuery pueden usar las **funciones** siguientes:

- **collection("/ruta")**: en ella se indica la ruta que hace referencia a una colección.
- **doc("/ruta/documento.xml")**: en ella se indica la ruta de un documento de una colección y el nombre de dicho documento.

Las consultas en XQuery se pueden realizar utilizando la expresión **FLWOR** (se lee como *flower* y se corresponde con las siglas ***For, Let, Where, Order y Return)***. Permite manipular, transformar y organizar los resultados de las consultas.

Sintaxis general de **FLWOR**:

for <variable> in <expresión XPath>

let <variables vinculadas>

where <condición XPath>

order by <expresión>

return <expresión de salida>

- **For:** esta expresión se utiliza para almacenar nodos en una variable. Dentro de la expresión ***for*** se escribe el XPath que seleccionará los nodos. Las variables se nombran comenzando con $.

    - Al realizar consultas en XQuery es necesario indicar una orden **Return**, en la que se establece lo que devuelve la consulta.

- **Let:** se asignan valores resultantes de expresiones XPath a variables para que la representación quede más simplificada. Una vez creadas las expresiones es posible poner varias líneas ***let*** o separar las variables por comas.

- **Where:** esta expresión filtra los elementos eliminando el resto de valores que no cumplan con dicha condición.

- **Order by:** se utiliza para ordenar los datos según el criterio establecido.

- **Return:** esta expresión construye el resultado de la consulta en XML. Es posible añadir etiquetas XML a la salida, las cuales habrá que encerrar entre llaves {}.

Asimismo, es posible añadir condicionales ***if-then-else***. Si se añaden, es preciso tener en cuenta que la cláusula ***else*** es obligatoria, ya que siempre tendrá que devolver un valor.

Con la función ***data ()*** es posible extraer en texto el contenido de los elementos y atributos.

Operadores y funciones más comunes en XQuery

Los **operadores y funciones** en el lenguaje XQuery son prácticamente los mismos que para XPath. Los más comunes son:

Matemáticos	+, -, *, div, idiv(división entera), mod
Comparación	=, !=, <, >, <=, >=, not()
Secuencia	Unión (|), intersect, except
Redondeo	Floor(), ceiling(), round()

Funciones de agrupación	Count(), min(), max(), avg(), sum()
Funciones de cadena	Concat(), string-lenght(), starts-with(), ends-with(), substring(), upper-case(), lower-case(), string()
Uso general	**Distinct-values()**[se extrae valores y crea nueva secuencia sin duplicados]; **empty()**[devuelve *cierto* cuando expresión entre paréntesis está vacía], **exists()**[devuelve *cierto* cuando contiene al menos un elemento]
Comentarios	Van encerrados entre caras sonrientes: (: Comentario :)

Sentencias de actualización de eXist

Para actualizar nodos se utilizan sentencias que permiten realizar altas, bajas y modificaciones de los nodos y elementos en el documento XML. Se pueden usar en cualquier punto del documento. Para un efecto inmediato es necesario usarlas dentro de la cláusula ***Return*** de una sentencia ***FLWOR***.

Todas las sentencias deben comenzar con **UPDATE** y seguidamente la instrucción. A continuación, se muestran las **sentencias más comunes**:

- **Insert**: inserta nodos.
  - ***Into***: contenido que se añade como último hijo de los nodos.

    update insert ELEMENTO into EXPRESIÓN XPATH

  - ***Following***: contenido que se añade justo después del nodo.

    update insert ELEMENTO following EXPRESIÓN XPATH

- ***Preceding***: contenido que se añade antes de los nodos.

update insert ELEMENTO preceding EXPRESIÓN XPATH

- **Replace**: sustituye el nodo especificado en NODO con un valor nuevo.

update replace NODO with Valor_Nuevo

- **Value**: actualiza el valor del nodo especificado en NODO con valor nuevo.

update valur NODO with 'VALOR NUEVO'

- **Delete:** elimina los nodos especificados.

update delete expr xpath

- **Rename:** renombra los nodos devueltos en NODO por el nuevo nombre.

update rename NODO as Nuevo_Nombre

1.8. Tratamiento de excepciones

A continuación, se muestran las excepciones **XMLDBEXCEPTION** y **XQEXCEPTION**, y su tratamiento, utilizando los bloques **try-catch.**

XMLDBException

Se lanza al producirse un error en la *API* ***XML:DB***. Lanzará dos códigos de error: el primero, el código *XML de la API*; y el segundo, que estará definido por el proveedor específico.

El error del proveedor viene definido por ErrorCodes.VENDO_ERROR. **XMLDBException** hereda de ***java.lang.Exception***.

Al ocurrir un error con **XMLDBException**, para poder acceder al contenido del mismo, es posible usar el método ***getMessage(),*** que permite mostrar una cadena con el contenido de dicho error.

El error **NullPointerException** se ejecutará siempre y cuando no se ejecute la base de datos.

Una mala redacción de la URI hará ejecutar ***XMLDBException*** al asignar la colección en ***getCollection*** y, a partir de este punto, se ejecutarán todos los ***NullPointerException***. Asimismo, estos se ejecutarán si se escribe mal la carpeta de la colección.

Por último, si se escribe mal la *query*, se ejecutará el error ***XMLDBException*** al crear el ***service.query*** y permitirá visualizar la línea del error.

XQException

Cuando se produce esta excepción se disponen de dos métodos para conocer qué tipo de error ha tenido lugar:

- ***getMessage()*:** muestra una cadena con el contenido del error.
- ***getCause()*:** indica la causa de dicho error para, de esta manera, poder darle solución.

UF4. COMPONENTES DE ACCESO A DATOS

En esta Unidad Formativa se desarrollarán los diferentes componentes de acceso a datos, su programación, concepto, propiedades y atributos principales.

1. Programación de componentes de acceso a datos

Los objetivos de esta unidad son, entre otros, conocer el uso de herramientas que ayudan a desarrollar componentes de acceso a datos, programarlos para ser capaces de gestionar la información que se encuentra almacenada en una base de datos, examinar dichos componentes implementándolos en una aplicación, utilizar un mismo componente en distintas bases de datos y, por último, definir modelos para comunicar con la base de datos utilizando el patrón *Modelo-Vista-Controlador*.

Contenidos principales de la unidad formativa:

- Concepto de componente y sus características.
- Propiedades y atributos.
- Eventos y asociación de acciones a eventos.
- Persistencia del componente.
- Herramientas para el desarrollo de componentes no visuales.
- Empaquetamiento de componentes.

1.1. Concepto de componente y características

Actualmente existen muchas definiciones del concepto de componente, pero una de las más difundidas es la de **Szyperski,** 1998:

> *"Un **componente** es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido e incorporado al sistema, y está compuesto por otros componentes de forma independiente, en tiempo y espacio".*

En definitiva, un componente es una unidad de software que está formada por partes de código que ofrecen una funcionalidad específica. Trabajar con componentes supone poder ensamblar módulos, independientemente del lugar en el que vayan a ser utilizados.

Características

Para que un elemento pueda ser considerado como componente debe reunir las siguientes **características**:

- Deber ser **independiente** de la plataforma que se vaya a utilizar, ya sea tipo Hardware, Software o un Sistema Operativo.
- Debe ser **identificable** , que permita su clasificación.
- Debe poseer su **propio contenido** sin necesitar fuentes externas para realizar el fin por el que fue desarrollado.
- Debe **poder cambiar** por una versión mejor o por otro componente que lo mejore.
- Solo puede tener acceso a través de su **propia interfaz**, así los **servicios ofrecidos no deben variar**, aunque su implementación sí.
- Debe estar **bien documentado** y servir para **varias aplicaciones**.
- Se distribuye a través de **paquetes**, a la hora de ser cargado en una aplicación se debe hacer durante su ejecución y ser distribuido a través de un paquete donde se almacenan todos sus elementos.

Especificar, implementar o empaquetar un componente dependerá de la tecnología utilizada. Estas tecnologías basadas en componentes incorporan dos elementos:

- **Modelo de componentes**: en el que se se especifican las reglas para el diseño de los componentes.
- **Plataforma de componentes**: infraestructura de software requerida para la ejecución de los componentes.

Algunos **ejemplos** de tecnologías basadas en componentes son:

- **Plataforma .NET**, de Microsoft para sistemas Windows.
- **JavaBeans y EJB (Enterprise JavaBeans),** de Oracle Corporation.

Ventajas e inconvenientes

Es importante conocer las ventajas y los inconvenientes que proporcionan el uso de componentes.

Recordemos que un **componente**, es una pieza de software que permite describir o ejecutar distintas funciones a través de una interfaz definida. Este componente puede ser unido a otros para ejecutar distintas tareas. Esto ofrece una serie de ventajas:

- **Reutilización** de dicho software.
- **Disminución de la complejidad,** ya que es posible ejecutar diferentes partes para comprobar su funcionamiento.
- **Mejora del mantenimiento**, al encapsular funcionalidades autocontenidas.
- **Facilidad en la detección de errores.**
- **Aumento de la calidad** del software, ya que es posible mejorarlo con el paso del tiempo.

Por otro lado, existen algunos **inconvenientes**:

- Solo es posible encontrar estos componentes en las GUIs.
- No siempre se localizan los componentes necesarios para un proyecto.
- Faltan estándares y procesos de certificación que garanticen la calidad de los mismos.

1.2. Propiedades y atributos

En esta unidad se explican los ***JavaBeans***, tecnología de componentes basada en Java. Frecuentemente se usa el término **Bean** para hacer referencia a un JavaBean.

Un ***Bean o JavaBean*** es un componente reutilizable, construido en lenguaje Java. Es una clase que se define a través de las propiedades, los métodos que ofrece y los eventos que genera.

Para **definir un Bean** se requieren ciertas **reglas**:

- Tener uno o más **constructores**, al menos uno sin argumentos.
- Debe implementar la interfaz **Serializable**.
- Debe ofrecer **acceso** (público) a sus propiedades mediante métodos **get y set**.
- Los nombres de los **métodos (getter y setter)** deben obedecer a ciertas normas, como son:
    - El nombre debe estar precedido por **get** para una propiedad no booleana.
    - Si es booleana debe ir precedido por **get** o **is**.
    - Para almacenar un valor debe tener el prefijo **set**.
    - Es imprescindible poner la primera letra en mayúscula para completar algún método.
    - Los métodos (tanto getter como setter) tienen que ser públicos.
    - Los métodos marcados como *setter* tienen que devolver tipo *void* y recibir un argumento de la propiedad a la que dan valor.
    - Los métodos *getter* no devuelven tipo *void*, pero sí devuelven valor del mismo tipo que el que recibe el método *setter*.

Las **propiedades de un *Bean*** son los atributos capaces de determinar su apariencia y comportamiento. Por ejemplo, "*Película*" puede tener las siguientes propiedades: *id, nombre, género, duración*, etc.

Para obtener información de alguna de ellas se usarán los métodos anteriormente nombrados *getter* y, para cambiar el valor, se utilizarán los métodos *setter*, de la siguiente forma:

public TipoPropiedad getNombrePropiedad() { … }

public void setNombrePropiedad (TipoPropiedad valor) { … }

EJEMPLO CÓDIGO:

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

Las **propiedades de un Bean** pueden ser de cuatro **tipos**:

A) Simples:

Es la más común y representan un único valor.

EJEMPLO CÓDIGO:

```
// Propiedad nombre
public String getNombre() {
	return nombre;
}
public void setNombre(String nuevoNombre) {
	nombre = nuevoNombre;
}

// Propiedad booleana conectado
public boolean isConectado(){
    return conectado;
}
public void setConectado(boolean nuevoValor){
    conectado=nuevoValor;
}
```

B) Indexadas:

Están representadas por un *array* de valores a los que se puede acceder mediante un índice. Para acceder a esos valores es necesario usar los métodos *getter* y *setter*, tanto para el acceso a valores del array completo, como a valores individuales.

EJEMPLO CÓDIGO:

```
// get y set para el array completo
public int[] getNumeros() {
	return numeros;
}
public void setNumeros(int[] nuevoValor) {
	numeros = nuevoValor;
}

// get y set para un elemento de array
public int getNumeros(int indice) {
	return numeros[indice];
}
public void setNumeros(int indice, int nuevoValor) {
	numeros[indice] = nuevoValor;
}
```

C) Ligadas:

Denominadas también **propiedades *compartidas***. Se encuentran asociadas a eventos. Al cambiar la propiedad se notifica a otros objetos y, de esta manera, se les permite llevar a cabo alguna acción.

Para crear una propiedad ligada se utilizan una serie de métodos de la clase ***PropertyChangeSupport***. Los métodos ***addPropertyChangeListener()*** y ***removePropertyChangeListener()*** se deben implementar en el bean que tiene contiene la propiedad ligada (bean fuente).

En los beans que reciben el cambio (bean receptor) es preciso implementar el método **propertyChange()** de la interfaz **PropertyChangeListener()** para que el Bean sea capaz de oir los eventos de cambio de propiedad.

Por tanto, se denomina **bean fuente** al que posee la propiedad ligada y, **bean receptor**, al que recibe la notificación. Este último bean receptor debe implementar **PropertyChangeListener**, como puede verse en el siguiente ejemplo.

EJEMPLO CÓDIGO:

```
public class BeanFuente implements Serializable {

	private PropertyChangeSupport propertySupport;

	// En el constructor se indica el objeto al que dar soporte (this)
	public BeanFuente() {
		propertySupport = new PropertyChangeSupport(this);
	}

	// ----RESTO DE CODIGO----//

	public void addPropertyChangeListener(PropertyChangeListener lis) {
		propertySupport.addPropertyChangeListener(lis);
	}

	public void removePropertyChangeListener(PropertyChangeListener lis) {
		propertySupport.removePropertyChangeListener(lis);
	}
}

public class BeanReceptor implements PropertyChangeListener {

	// ----RESTO DE CODIGO----//
	public void propertyChange(PropertyChangeEvent evento) {
		System.out.println("Valor previo: " + evento.getOldValue());
		System.out.println("Nuevo valor: " + evento.getNewValue());
	}
}
```

D) Restringidas:

Similares a las ligadas, pero con la diferencia de que los objetos pueden ser vetados si no poseen ciertas características. Se proporcionan dos **métodos**:

- ***addVetoableChangeListener()***: capaz de capturar eventos tipo *PropertyVetoEvent*.
- ***removeVetoableChangeListener():*** permite que el objeto deje de escuchar eventos.

Para poder definir ambos métodos se utilizará un objeto denominado ***VetoableChangeSupport***. Se implementa el método ***vetoableChange()*** de la interfaz ***VetoableChangeListener*** para que pueda oír eventos.

El método recibe una copia del objeto ***PropertyChangeEvent*** y lanza excepciones del tipo ***PropertyVetoException***.

Se obtiene el mensaje del objeto excepción (***getMessage()***) y el evento que la desencadenó (***getPropertyChangeEvent()***). El *Bean* captura la excepción y le asigna a la propiedad el valor anterior sin notificarle a las demás partes el nuevo valor.

Las propiedades restringidas son más difíciles de implementar que las ligadas.

EJEMPLO CÓDIGO:

```
public class BeanFuente implements Serializable {

	private VetoableChangeSupport soportaVeto;

	// En el constructor se indica el objeto al que dar soporte (this)
	public BeanFuente() {
		soportaVeto = new VetoableChangeSupport(this);
	}

	// ----RESTO DE CÓDIGO----//

	public void addVetoableChangeListener(VetoableChangeListener lis) {
		soportaVeto.addVetoableChangeListener(lis);
	}

	public void removeVetoableChangeListener(VetoableChangeListener lis) {
		soportaVeto.removeVetoableChangeListener(lis);
	}
}

public class BeanReceptor implements VetoableChangeListener {

	public void vetoableChange(PropertyChangeEvent evento)
                                          throws PropertyVetoException {
		// Ahora se comprueban condiciones.
		System.out.println("Valor previo: " + evento.getOldValue());
		System.out.println("Nuevo valor: " + evento.getNewValue());
		//Se lanza excepción si no se cumple el cambio considerado
		throw new PropertyVetoException("Aquí el mensaje de error",
		   evento);
	}
}
```

1.3. Eventos y asociación de acciones a acontecimientos

En el apartado anterior, cuando se comentaron las propiedades ligadas y restringidas ya se introdujo el uso de eventos. Los eventos se utilizan para la **comunicación de un Bean con otro**.

Para lanzar eventos se usa el método ***firePropertyChange() o fireVetoableChange(),*** según se tenga que implementar algún veto o no. Ambos métodos pueden recibir tres parámetros: nombre, valor antiguo y valor nuevo. Tanto el antiguo como el nuevo deben ser de tipo Object.

Este receptor debe implementar el método ***propertyChange()*** o ***vetoableChange()*** de la interfaz ***PropertyChangeListener*** o ***VetoableChangeListener***. Si este es llamado, entonces se modifica el valor de la propiedad. El evento proporciona varios métodos: ***Object getOldValue(),*** para obtener el valor antiguo; ***Object getNewValue(),*** para obtener el valor nuevo, y ***String getPropertyName(),*** para obtener el nombre.

EJEMPLO CÓDIGO:

```
public void setStockActual(int cantidadNueva)  {
      int cantidadAnterior = stockActual;
      stockActual = cantidadNueva;
      if (stockActual < getStockLimite()) { // Falta stock
            try {
                soportaVeto.fireVetoableChange("stockActual",
                                    cantidadAnterior, cantidadNueva);
            } catch (PropertyVetoException e) {
                // Aquí va código de excepción
            }
      }
}
```

1.4. Persistencia del componente

En una de las reglas que deben de cumplir los *Bean* está el hecho de que estos deben implementar la interfaz ***Serializable***. Esta implementación permite almacenar su estado en un momento determinado para poder recuperarlo posteriormente. El proceso consiste en que todos los valores se trasladarán a una cadena de bytes y se almacenarán en un fichero desde el que se podrán reconstruir después.

Es necesario tener en cuenta que las clases que implementan **Serializable** deben tener un constructor sin argumentos y que todos los campos, excepto **static** y **transient,** son serializados. Cuando un campo no se desee serializar, se hará uso del modificador trasient.

EJEMPLO CÓDIGO PARA NO SERIALIZAR:

```
transient int valor;
```

Hay que tener en cuenta que se puedan guardar todas las características del Bean que permitan que este se reconstruya al estado anterior en el que se encontraba.

1.5. Herramientas para el desarrollo de componentes no visuales

En la explicación de este apartado se van a utilizar dos *Bean*. Para el primero se usará un tipo de propiedad ligada. Se trata de un *Bean* fuente llamado *Materiales,* con una propiedad ligada llamada *cantidadActual* de tipo *int*. En el caso del segundo, se usará un *Bean* receptor denominado *PedidoMateriales* que estará centrado en los cambios de dicha propiedad, cuando la cantidad de material sea inferior al mínimo.

***Supuesto práctico**:*

*Se debe realizar un pedido cuando la cantidad de material es inferior al mínimo. Para ello se creará la clase **Materiales** y la clase **PedidoMateriales** con sus atributos.*

*Para crear un JavaBeans con NetBeans, en primer lugar, es necesario abrir NetBeans y seleccionar **File-> New Project -> Java -> Java Class Library** y, después, pulsar el botón **Next**.*

*Tras esto, se escribe el nombre de la nueva librería, por ejemplo, **PrimeraLibreria**, la ubicación y la carpeta donde se situará el proyecto y se pulsa en **Finish**.*

Una vez creado el proyecto, ya es posible comenzar a crear los JavaBeans.

*Para llevarlo a cabo, se pulsa el botón derecho del ratón y se selecciona la siguiente ruta: **New -> Other -> JavaBeans Objects -> JavaBean Component** y se escribe el nombre del Bean (Materiales) y, después, se selecciona el nombre del paquete del que formará parte. De esta manera se habrá creado un Bean Materiales con un código por defecto.*

*Al crear el Bean se creará también una propiedad de ejemplo llamada **sampleProperty** con sus respectivos métodos get y set. Se creará un objeto **PropertyChangeSupport** llamado **propertySupport** para definir así, de forma fácil y sencilla, los métodos **addPropertyChangeListener()** y **removePropertyChangeListener()**.*

*A continuación, se modifica el código del JavaBean. Para ello es necesario eliminar el atributo **sampleProperty** y sus métodos get y set, y añadir los siguientes atributos:*

- *private String descripcion;*
- *private int identificador;*

- *private int cantidadactual;*
- *private int cantidadminima;*
- *private float precio;*

*Después se **añade el siguiente constructor**:*

EJEMPLO CÓDIGO:

```
public Materiales (int identificador, String descripción, int
cantidadactual; int cantidadminima, float precio)  {
      propertySupport = new PropertyChangeSupport (this);
      this.identificador = identificador;
      this.descripcion = descripcion;
      this.cantidadactual = cantidadactual;
      this.cantidadminima = cantidadminima;
      this.precio = precio;
}
```

*Tras este paso se generan los getter y setter para los atributos. Se hace clic con el botón derecho del ratón debajo de los constructores y se selecciona **Insert Code -> Getter and Setter**, y, a continuación, se marcan los atributos que aparecen. En ellos se insertarán dichos métodos y se pulsará el botón **Generate**.*

*Para crear un método llamado **setCantidadactual**(), en el que se crea un evento si la cantidad actual es menor a la mínima, se usará el método **firePropertyChange()**) y no se actualizará la cantidad antigua.*

*Para crear el JavaBean Materiales en el paquete, se elimina todo el código de la clase y se incluye **PropertyChangeListener** a la cláusula **implements**. Será necesario sobreescribir el método **propertyChange()**.*

*A continuación, se añaden los atributos para la clase, además de los métodos getter y setter para estos atributos, y se cambia el método **propertyChange()**. Se añaden también los constructores.*

*Una vez creado el JavaBeans, se genera el **JAR** pulsando el botón derecho del ratón y, posteriormente, la opción **Build**. NetBeans mostrará en la ventana Output mensajes con los directorios creados.*

*Si no existe ningún error, se creará el fichero **PrimeraLibreria.jar** en la carpeta **dist** del proyecto. Para poder modificar los JavaBeans es preciso pulsar la opción **Clean and Build** y, de esta manera, se construye de nuevo la librería.*

1.6. Empaquetamiento de componentes

Una vez se ha creado el *componente* es necesario empaquetarlo para proceder a su distribución y uso.

Para distribuirlo se crea un fichero ***JAR*** que contiene un fichero de manifiesto **(MANIFEST.MF)** que describe el contenido. Se deberán incluir las clases y los recursos que lo forman. Cuando se crean, hay que asegurarse de que existe una línea vacía entre entradas y que la última línea es un salto de línea. No debe haber espacios al final de las líneas.

Este fichero MANIFEST.MF se suele encontrar en la carpeta **META-INF**. Una vez preparado, se crea el fichero *.jar* usando el comando jar de Java:

EJEMPLO:

```
jar cfm fichero.jar META-INF/MANIFEST.MF carpeta/*.class
```

En el que:

- **c**: indica que se crea un fichero.
- **f**: salida va a un fichero.
- **m**: indica las líneas del fichero manifiesto.
- **fichero.jar**: nombre que se le da al fichero.
- **META-INF/MANIFEST.MF**: indica dónde se encuentran las líneas de manifiesto que se añadirán al fichero.
- **carpeta/*.class**: indica dónde están los ficheros de entrada.

Con este procedimiento, obtenemos el componente *JavaBeans* empaquetado para utilizarlos en cualquier programa Java.

Bibliografía

Alicia Ramos Martín y Mª Jesús Ramos Martín, *Acceso a Datos*. Garceta, 2016.

José Eduardo Córcoles Tendero y Francisco Montero Simarro, *Acceso a Datos*. Ra-Ma, 2013.

P. Pablo Garrido Abenza, *Comenzando a programar con Java*, Universidad Miguel Hernández de Elche, 2015.

Webgrafía

docs.oracle.com

docs.oracle.com/javase/8/docs/api/java

www.ibm.com

docs.jboss.org/hibernate/orm/current/javadocs

ILERNA Online