

ACCESO A DATOS

Definición y tipos de ficheros

Fichero: un fichero es la sucesión de bits que finalmente son almacenados en un determinado dispositivo. Está compuesto por:

- **Nombre** (Identifica al fichero)
- **Extensión** (tipología del archivo)

Podemos organizar un fichero en:

- **De texto (ASCII):** líneas organizadas en código ASCII, por ejemplo, archivos de aplicaciones de ofimática.
- **Binarios:** ficheros cuya información está representada en código binario. Algunos ejemplos son **.bin, .dat...**

JAVA.IO.FILE

- **Crear, modificar y eliminar ficheros.**

```
static final String DIRECTORIO = "src/archivos";
File archivo = new File(DIRECTORIO, nombreArchivo);
```

- **Para crear ficheros como directorios usamos:**

```
archivo.createNewFile();
```

- **Para eliminar tanto ficheros como directorios:**

```
archivo.delete();
```

MÉTODOS ÚTILES

- Por ejemplo el método **.exists()** devuelve **true o false** en dependencia de si en el directorio pasado existe el archivo.

```
public static Boolean buscarFichero(String nombreArchivo) {
    File archivoEncontrar = new File(DIRECTORIO, nombreArchivo);
    return archivoEncontrar.exists();
}
```

- Otro método en este caso para listar los ficheros de un directorio es **.list()**. Este método **devuelve un array con los ficheros encontrados** en el directorio especificado.

```
File directorio = new File(DIRECTORIO);
String[] archivosEncontrados = directorio.list();
```

FORMAS DE ACCEDER A UN FICHERO

Cuando trabajamos con archivos, es importante elegir el método de acceso adecuado, que puede ser secuencial o aleatorio/directo.

- **Acceso secuencial:** implica leer o escribir información en un archivo de manera lineal, lo que significa que debemos pasar por todos los datos previos para llegar al que deseamos.

- **Acceso aleatorio o directo:** nos permite acceder directamente a registros o posiciones específicas

OPERACIONES DE GESTIÓN DE FICHEROS

Para el acceso secuencial debemos tener en cuenta:

CLASES DE USO DE ACCESO SECUENCIAL

Acceso con bytes

- **FileInputStream (Lectura):**

```
FileInputStream f = new FileInputStream(DIRECTORIO);
int bytes = f.read();
f.close();
```

Con el método **f.read()** conseguimos el primer byte, pudiendo así leer imágenes o archivos binarios.

- **OutputStream (Escritura):**

```
String cadena = "Hola como estas";
byte[] cadenaBytes = cadena.getBytes();

FileOutputStream f = new FileOutputStream(DIRECTORIO);
f.write(cadenaBytes);
f.close();
```

Con este método podemos escribir una cadena de bytes en un fichero. Gracias a **.getBytes()** podemos conseguir almacenar los bytes de una cadena en un array que luego escribiremos con **.write()**.

Acceso con caracteres

- **FileReader (Lectura):**

```
FileReader f = new FileReader(DIRECTORIO);
int ch = f.read();
System.out.println((char)ch);
f.close();
```

La única diferencia de los de arriba es que justo antes de mostrar el resultado por pantalla, realizamos un **casting (char)ch** para poder visualizar ese primer carácter de nuestro fichero.

- **FileWriter (Escritura)**

```
FileWriter f = new FileWriter(DIRECTORIO);
f.write("Nuevo archivo de ejemplo");
f.close();
```

En el caso que no existiera el fichero, se creará automáticamente. Si el fichero existe, será sobrescrito. En la segunda línea directamente insertamos los caracteres que deseamos incluir en nuestro fichero.

OPERACIONES CON BUFFER

Conjunto de clases cuyo fin es leer y escribir información mejorando el rendimiento del sistema.

DEFINICION Y CLASES

El **buffer** es un **espacio determinado y temporal que se aloja en memoria** para realizar ciertas operaciones.

Las clases con sufijo **Buffered** almacena en memoria interna bloques de bytes completos (**buffer**), y como ya bien sabemos, el acceso a memoria es mucho más rápido que a disco.

Cada vez que agotamos esa información del buffer y se requiere más, se vuelve a volcar otro bloque de bytes a la memoria (**buffer**).

```
/*
 * E/L - Escritura / Lectura
 *
 * Cualquier operación con el BufferedWriter o BufferedReader, tiene más rendimiento cuando se espera E/L un contenido muy largo,
 * ya que no se E/L directamente en el archivo sino en un buffer, evitando así la sobrecarga de E/S.
 */
try (BufferedWriter bw = new BufferedWriter(new FileWriter(archivo))) {
    bw.write(contenidoArchivo);
    bw.flush(); // Forzamos a escribirlo en el archivo "físico", sin esperar el cierre de flujo o el llenado del buffer.

    System.out.println("Archivo creado exitosamente: " + archivo.getAbsolutePath()); // Referenciamos donde se ha creado el archivo.

} catch (IOException e) {
    System.out.println("Error al crear el archivo " + nombreArchivo + ": " + e.getMessage());
```

FLUJO DE DATOS (STREAMS):

- Secuencia ordenada de información con un flujo de entrada y un flujo de salida.
- Se dividen en varias categorías según el tipo de información a tratar.
- Unidireccionalidad: solo lectura o escritura, no ambas simultáneamente.

TUBERIAS EN JAVA

En Java, permiten la comunicación entre hilos dentro del mismo proceso.

- PipedOutputStream
- PipedInputStream

FLUJOS EN ARRAYS

Acceso basado en bytes:

- ByteArrayInputStream
- ByteArrayOutputStream

Acceso basado en caracteres:

- CharArrayReader
- CharArrayWriter

CLASES PARA EL ANÁLISIS DE DATOS:

PushbackReader

PushBackInputStream

- Están diseñadas para el análisis de datos previo de un InputStream.

StreamTokenizer

- Clase para analizar ficheros por fragmentos (tokens).
- Capaz de reconocer identificadores, números, comillas, espacios, etc.

LineNumberReader

- Clase que cuenta el número de líneas leídas de caracteres.
- **Métodos clave:**
 - getLineNumber()
 - setLineNumber(),
 - readLine().

DOM / SAX

Estándares para la lectura de ficheros XML, ambos son parsers que verifican la sintaxis de los ficheros.

DOM

- Carga el fichero completo en memoria, generando un árbol de nodos.
- Ofrece acceso directo a todo el árbol de objetos.
- Es más lento y menos versátil que SAX.
- Aconsejable cuando se tiene un objetivo claro y se trabajará con el árbol creado en memoria.

SAX

- Analiza el fichero nodo por nodo sin cargarlo completo en memoria.
- Trabaja con eventos y requiere programación para acceder a partes específicas del fichero.
- Más rápido y eficiente en términos de memoria que DOM.
- Aconsejable para recorrer secuencialmente los elementos del fichero y realizar operaciones específicas.

PROCESAMIENTO DE XML: XPATH

XPATH es una recomendación oficial del W3C y se utiliza para la búsqueda de información en documentos XML.

Sus características clave son:

DEFINICIÓN DE ESTRUCTURAS:

- Permite definir partes del documento XML, como elementos, atributos, texto, instrucciones de procesamiento, comentarios y nodos del documento.

EXPRESIONES:

- Ofrece expresiones poderosas para manipular archivos XML, permitiendo la selección de nodos o listas de nodos en dichos archivos.

FUNCIONES ESTÁNDAR:

- Proporciona una completa librería de funciones estándar para la manipulación de cadenas, valores numéricos, fechas, comparaciones, secuencias y valores booleanos.

EXCEPCIONES

Una excepción es un evento que interrumpe el flujo de ejecución de un programa debido a diversas razones.

Se dividen en tres categorías:

Excepciones con Chequeo:

- Notificadas por el compilador en tiempo de compilación.
- No pueden ser ignoradas y obligan al programador a manejarlas.

Excepciones sin Chequeo (RuntimeExceptions):

- Originadas en tiempo de ejecución.
- Incluyen errores de programación o mal uso de una API.
- Ignoradas en tiempo de compilación.

Errores:

- Originadas en tiempo de ejecución.
- Incluyen errores de programación o mal uso de una API.
- Ignoradas en tiempo de compilación.

EXCEPCIONES ASOCIADAS A CLASES XML

En este apartado veremos algunas de las excepciones utilizadas en las clases anteriores, pero antes vamos a visualizar algunos de los métodos más importantes que debemos conocer si queremos hacer manejo de excepciones:

- **getMessage()**: devuelve un mensaje detallado sobre la excepción que se acaba de lanzar. El mensaje es instanciado en el constructor de la clase Throwable.
- **getCause()**: devuelve la causa representada en un objeto Throwable.
- **toString()**: devuelve el nombre de la clase y se le concatena el resultado de getMessage().
- **printStackTrace()**: imprime el resultado del método toString() junto con el error de sistema que devuelve la pila.
- **getStackTrace()**: devuelve un array con cada uno de los elementos de la pila. El elemento 0 del array representa el elemento más alto de la pila.
- **fillInStackTrace()**: rellena la pila del objeto Throwable con la pila actual. Le añade cualquier información previa en el seguimiento de la misma

El código incluido dentro de un bloque try/catch se conoce como código protegido.

DEFINICION CONECTOR

Serie de clases y librerías que realizan la labor de unir la capa de nuestra aplicación con la capa de base de datos

DESFASE OBJETO-RELACIONAL

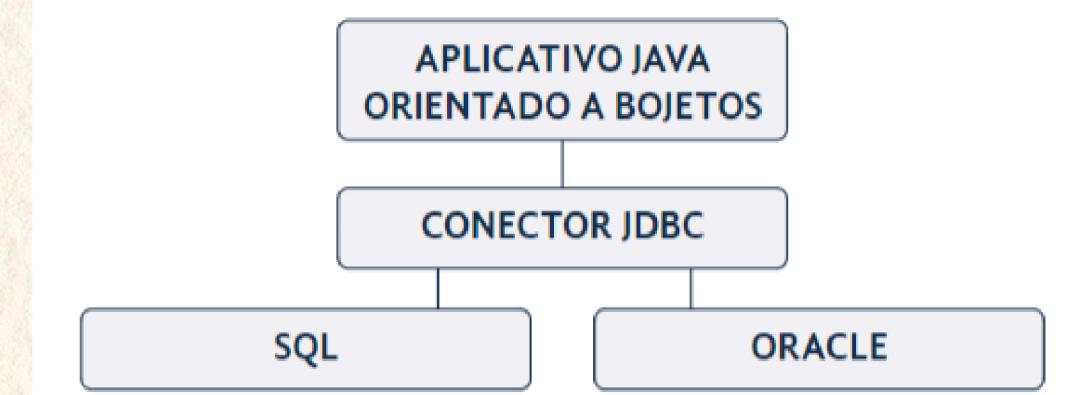
Problema que surge por las discrepancias de que la parte de las bases de datos tienen naturalezas distintas en comparación al aplicativo que se trabaja con programación orientada a objetos:

- **Diferencia entre los datos** que se usan en las bases de datos relacionales (int, varchar, objetos simples) y en la programación orientada a objetos (objetos complejos).
- Implica **realizar distintos diagramas**, debido a que vamos a tener que realizar una traducción de los objetos Java.

PROTOCOLOS DE ACCESO A BASE DE DATOS

En el lenguaje SQL, disponemos de dos protocolos de conexión:

- **JDBC (Java Database Connectivity)**: desarrollado por Sun.
- **ODBC (Open Database Connectivity)**: desarrollado por Microsoft.



Teniendo en cuenta esto, la facilidad que nos supone tener este tipo de conectores, es el hecho de no tener que desarrollar en nuestras aplicaciones clases para conectar a distintas bases de datos, sino que el conector será el que interprete hacia qué base de datos ir.

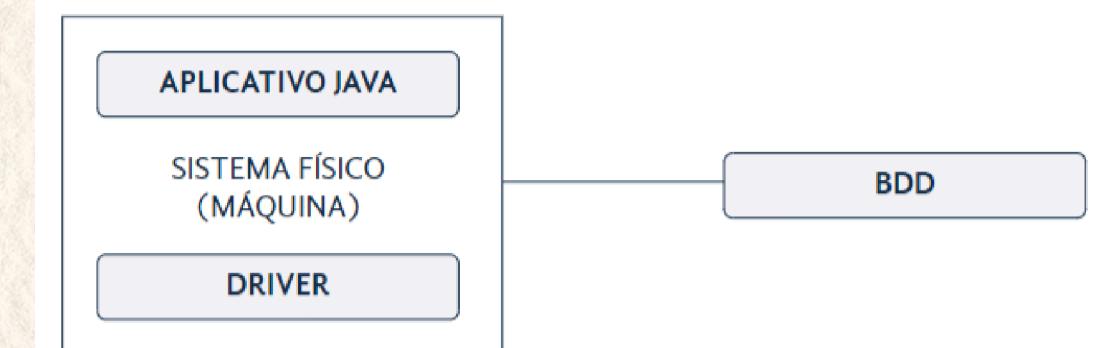
CONEXIONES: COMPONENTES Y TIPOS

COMPONENTES

- **API JDBC**: con ella tenemos una serie de librerías y clases que nos facilitan el acceso a las bases de datos relacionales.
- **Paquete de pruebas JDBC**: estas clases se encargan de validar si un driver pasa los requisitos previstos por JDBC.
- **El gestor JDBC**: realizar la unión entre nuestra aplicación Java con el driver apropiado JDBC.
 - Conexión directa.
 - Pool de conexiones.
- **El puente JDBC-ODBC**: facilita el uso de los drivers ODBC.

En relación a la arquitectura de conexión con JDBC, podemos identificar dos tipos de arquitectura:

- **Arquitectura en dos capas**: nuestra aplicación se conectará a la BDD a través de un driver.



- **Arquitectura en tres capas**: Nuestro aplicativo enviará las instrucciones a una capa intermedia (**middleware**). Esta capa cogerá la información y la enviará a la base de datos correspondiente traduciendo los comandos del aplicativo.



TIPOS

LEERLO POR ENCIMA PAG 7

CONFIGURACIÓN DE UNA CONEXIÓN EN CÓDIGO:

NO CREO QUE LO PONGA, LEERLO

BASES DE DATOS

- **Bases de datos en memoria:** este tipo de bases de datos almacena toda la información en memoria, como ya sabemos, este tipo de memoria es volátil y por lo tanto, la información permanecerá mientras el programa esté en ejecución.
- **Bases de datos embebidas:** aquella que es **parte de la aplicación que se ha desarrollado**. El motor de la base de datos corre con el mismo motor de la aplicación Java (JVM-Java Virtual Machine). Encaja perfectamente con la idea de tener un repositorio para **persistir** las transacciones sin ningún tipo de intervención por parte del usuario.
- **Bases de datos independientes:** aquí sí tenemos un gestor o administrador de base de datos dedicado a resolver ciertos problemas de mantenimiento. Las bases de datos cliente/servidor son independientes y, obviamente, las **más pesadas y potentes**.

PERSISTENCIA

Persistir los datos en una base de datos, significa el hecho de almacenar los datos de la misma de manera permanente para poder recuperarlos y usarlos en un futuro. En general cualquier operación de **CRUD**, en el **commit**, hace esto.

JAVA DB - EMBEBIDA:

Java DB está construida sobre el motor de la base de datos **Derby**. Esta está escrita por completo en el lenguaje **Java**. **Derby** soporta todas las funcionalidades estándar de una base de datos relacional.

Para embeber nuestra base de datos **Derby** en nuestra aplicación **Java**, simplemente, hay que incluir en nuestro proyecto Java el fichero **“derby.jar”** del directorio **/lib**. Este contiene tanto el propio motor de la base de datos como los conectores necesarios.

¿QUÉ ES SPRING BOOT?

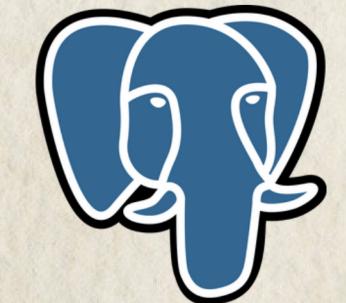
Spring Boot es un módulo dentro del ecosistema más amplio de **Spring Framework** que se enfoca en simplificar el desarrollo de aplicaciones Java, especialmente en lo que respecta a la configuración, la gestión de dependencias y el despliegue de aplicaciones.



BASES DE DATOS INDEPENDIENTES

Las bases de datos independientes tienen diferencias notables en comparación a las embebidas.

- No pueden ejecutarse bajo la misma máquina virtual de Java que usa nuestra aplicación en ejecución. En general este tipo de bases de datos, se instalan en local con fines de pruebas o aprendizaje, pero lo común es tenerla separada e instalada en otra máquina.
- Consumen más recursos, es por ello al estar en una máquina aislada, aprovechará mejor estos recursos.
- Están menos restringidas a nivel de funcionalidad, ya que no dependen de la limitación de un programa en concreto.



SENTENCIAS

CREACIÓN DE BASE DE DATOS

- **ALTER DATABASE:** cambiar características generales de una base de datos.
- **ALTER TABLE:** con este comando podremos modificar la estructura de una tabla que previamente hemos creado.
- **CREATE DATABASE:** con este comando realizaremos la creación de una nueva base de datos.

CREACIÓN Y ELIMINACIÓN DE TABLA

- **CREATE TABLE:** con este comando ejecutaremos la creación de una tabla con el nombre definido.
- **DROP DATABASE:** usaremos este comando para realizar un borrado permanente de todas las tablas de nuestra base de datos y borrar, así, dicha base de datos.
- **DROP TABLE:** con este comando podremos borrar una o más tablas en nuestra base de datos.

DELETE:

```
DELETE FROM Facturas  
-- Si no establecemos la cláusula WHERE,  
-- se eliminarán las filas de la tabla  
WHERE NombreCliente = 'Juan Pérez'
```

INSERT:

```
INSERT INTO Facturas  
(FechaFactura, CIFCliente, NombreCliente,  
VALUES  
'2023-01-15', '12345678A', 'Juan Pérez',
```

UPDATE:

```
UPDATE Facturas  
SET FechaCobro = NULL  
WHERE FechaCobro = '1900-01-01'
```

SELECT

```
SELECT *  
FROM Facturas  
SELECT DISTINCT EstadoFactura  
FROM Facturas
```

```
SELECT *  
FROM Facturas  
WHERE NombreCliente LIKE 'J%'
```

WHERE

```
SELECT *  
FROM Facturas  
WHERE YEAR(FechaFactura) = 2023
```

```
SELECT *  
FROM Facturas  
WHERE FechaCobro IS NULL
```

```
SELECT *  
FROM Facturas  
-- % + CARÁCTER ES AL FINAL,  
-- SI VA DETRÁS ES AL PRINCIPIO  
WHERE CIFCliente LIKE '%A'
```

```
SELECT *  
FROM Facturas  
WHERE Importe  
BETWEEN 100 AND 200
```

```
SELECT *  
FROM Facturas  
WHERE IdFactura IN (3, 6, 9)
```

```
SELECT *  
FROM Facturas  
WHERE IdFactura IN (3, 6, 9)  
ORDER BY Importe ASC
```

GESTIÓN DE TRANSACCIONES

Una transacción en SQL son unidades o conjuntos de **acciones que se realizan en serie** y de forma **ordenada**.

- Proporcionar **consistencia** en la base de datos realizando secuencias de alta fiabilidad. Esto puede permitir volver a versiones anteriores de forma más eficiente.
- Ofrecer **aislamiento**. (Encapsulamiento mientras otra operación puede estar en marcha con los mismos datos)

Teniendo en cuenta las transacciones, veremos tres comandos a usar:

- **Commit**: permite persistir los cambios de manera permanente en la base de datos.
- **Rollback**: permite deshacer los cambios ejecutados en una transacción.
- **Savepoint**: puntos de guardado de un punto en concreto y, en caso de **rollback**, se podrá volver a dicho punto de control.

```
-- Inicio de la transacción
BEGIN TRANSACTION;
-- Crear un savepoint
SAVE TRANSACTION PuntoGuardado;
-- Intentar realizar una operación que podría fallar
BEGIN TRY
    UPDATE Facturas
    SET NombreCliente = 'Luisa Fernandez'
    WHERE NombreCliente = 'Luisa Fernández';
    -- Confirmar la transacción si todo va bien
    COMMIT;
END TRY

BEGIN CATCH
    -- Si hay un error, hacer rollback al savepoint
    ROLLBACK TRANSACTION PuntoGuardado;
END CATCH;
```

LA INTERFAZ STATEMENT

Es la encargada de ejecutar las sentencias en nuestro programa y recoger los resultados para manipularlos más tarde.

Una vez creado el objeto **Statement**, podemos usar distintos métodos del mismo:

- **executeQuery (String)**: utilizado para las sentencias **SELECT**, nos devolverá un objeto **ResultSet** con la información de la tabla.

```
con.prepareStatement(sentencia)
ResultSet rs = pstmt.executeQuery();
```

- **executeUpdate (String)**: utilizado para las sentencias **INSERT**, **UPDATE**, **DELETE**, nos devolverá un objeto de tipo **INT**.

```
int filasAfectadas = pstmt.executeUpdate();
```

- **execute(string)**: este método devolverá **true** si devuelve un **ResultSet**, y para acceder a él, tendremos que ejecutar el método **getResultSet ()**, o **false**, si lo que estamos ejecutando, por ejemplo, es un **UPDATE**, en ese caso, si queremos saber las filas afectadas consultaríamos el método **getUpdateCount ()**.

EL MAPEO OBJETO RELACIONAL

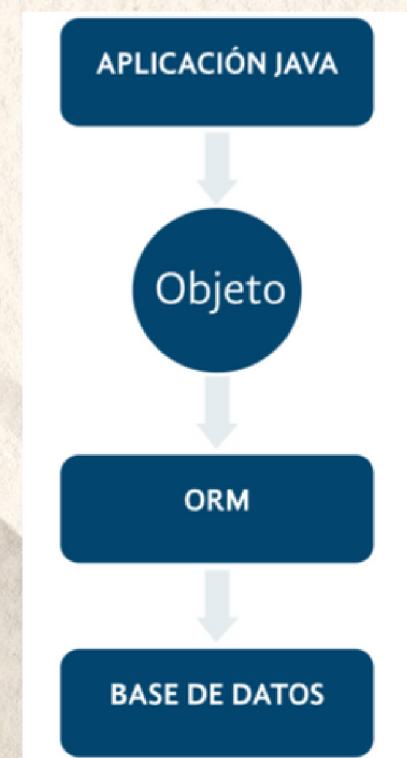
DEFINICIÓN

Para entender el mapeo objeto relacional, debemos irnos a la definición de temas anteriores sobre el **desfase objeto-relacional**.

Para solventar este tipo de problemas o diferencias de datos a almacenar, tenemos la ayuda del Mapeo Objeto Relacional.

En este caso llamado **ORM (Object Relational Mapping)**, se podría definir como **framework** que facilita el almacenamiento de los objetos a una base de datos relacional.

Un objeto plano es **equivalente a una fila** en una base de datos relacional, siendo esto lo que mapearemos



VENTAJAS

- Mejora en la **eficiencia** del desarrollo.
- Desarrollo más **orientado a objetos**.
- **Manejabilidad**.
- Facilidad para **introducir nuevas funciones** como el cacheo de información (almacenamiento en caché)

INCONVENIENTES

- **El mapeado automático** de las bases de datos **consumen muchos** recursos de sistema.
- **La sintaxis de los ORM** a veces puede complicarse si realizamos consultas muy complejas mediante las que crucemos varias tablas.

FASES

- **Fase 1:** Enfocada en los datos del objeto, esta fase incluye clases Java simples, **POJOs**, **implementaciones**, **capa de negocio** (llamada capa servicio), **y clases DAO**. También contiene métodos para interactuar con la capa de datos.

- **Fase 2:** También conocida como **fase de persistencia** o **mapeo**, involucra varios elementos:
 - **Proveedor JPA:** una librería que habilita la funcionalidad de JPA (Java Persistence API).
 - **Archivo de asignación:** un fichero XML que guarda la configuración de la asignación entre los datos de una clase Java (POJO) y los datos reales de la base de datos relacional.
 - **JPA Cargador:** funciona como una memoria caché que carga datos desde la base de datos para interacciones rápidas con las clases de servicio.
 - **Reja de objeto:** un espacio temporal donde se almacena una copia de los datos de la base de datos relacional, actuando como un punto intermedio para consultas antes de ser verificadas y transferidas a la base de datos principal.
- **Fase 3:** Conocida como **fase de datos relacionales**. En esta fase, los datos se transfieren directamente a la base de datos, pero hasta ese momento, se permanece en el espacio temporal de la caché.