UAX FP

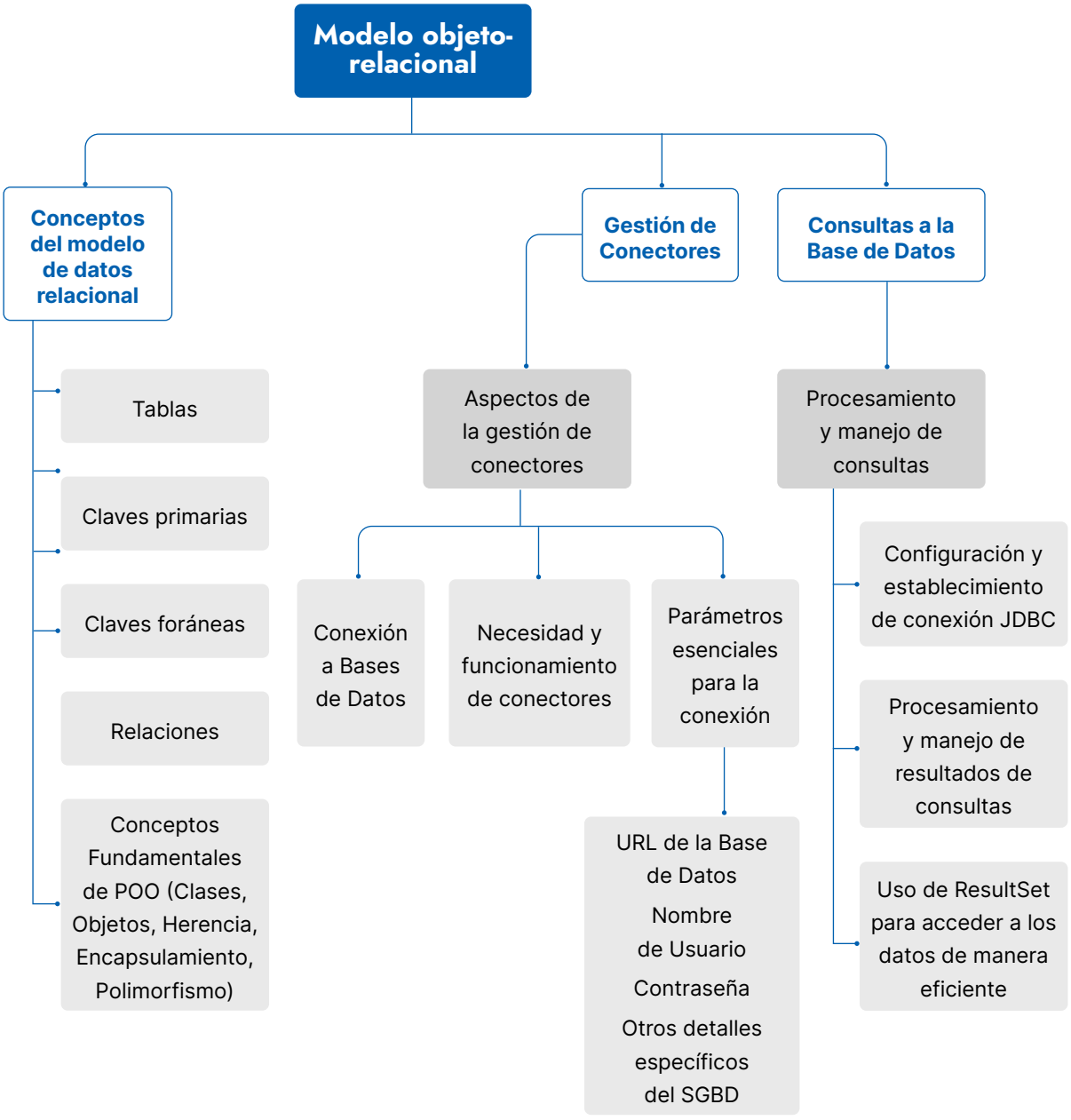
UF2

Modelo objeto-relacional

Acceso a datos

ÍNDICE

Mapa conceptual



01 El desfase objeto-relacional

Los **modelos conceptuales** nos ayudan a modelar una realidad compleja, y se basan en un proceso de abstracción de la realidad. Cada modelo tiene una forma de plasmar esta realidad, pero todas ellas son más próximas a la mentalidad humana que a la memoria de un ordenador.

Cuando modelamos una base de datos, hacemos uso del modelo conceptual entidad-relación y, posteriormente, llevamos a cabo un proceso de paso a tablas y normalización de este modelo, para tener un **modelo relacional de datos.**

En el caso de la **programación orientada a objetos,** intentamos representar la realidad mediante objetos y las relaciones entre ellos. Se trata de otro tipo de modelo conceptual, pero que pretende representar la misma realidad que el relacional.

Así pues, tenemos dos aproximaciones diferentes para representar la realidad de un problema:

- El modelo relacional de la base de datos.
- El modelo orientado a objetos de nuestras aplicaciones.

1.1. Representación de la información con el modelo relacional

El modelo relacional se basa en tablas y en la relación entre ellas:

Cada tabla tiene tantas **columnas** como **atributos** queremos representar, y tantas **filas** como **registros** o elementos de ese tipo contenga.

Las tablas tienen una **clave principal**, que identifica cada uno de los registros, y puede estar formada por uno o varios atributos.

La relación entre tablas se representa mediante **claves externas**, que consisten en incluir en una tabla la clave principal de otra tabla, como «referencia» a esta.

Cuando se elimina un registro de una tabla cuya clave primaria está referenciada por otra, hay que asegurarse de que mantenemos la integridad referencial de la base de datos. Entonces, ante este borrado de una clave primaria, podemos:

- No permitir el borrado `(NO ACTION)`.
- Realizar el borrado en cascada, borrando también todos los registros que hicieron referencia a la clave primaria del registro borrado `(CASCADE)`.
- Establecer en nulos `(SET NULL)`, de forma que la clave externa que referenciaba a la clave primaria de la otra tabla toma el valor de `NULO`.

Los diferentes campos de las tablas pueden tener también ciertas restricciones asociadas, como pueden ser:

- Restricción de valor no nulo, de forma que el campo no puede ser nulo en ningún caso.
- Restricción de unicidad sobre uno o varios campos, de forma que el valor tiene que ser único en toda la tabla. Las claves primarias poseen ambas propiedades: valor no nulo y unicidad.
- Restricción de dominio, o, lo que es lo mismo, pueden tener un conjunto de valores posibles predeterminado.

1.2. Representación de la información con el modelo orientado a objetos

Al igual que el modelo entidad-relación, el modelo orientado a objetos es un modelo de datos conceptual, pero que centra la importancia en el modelado de objetos.
Un objeto puede representar cualquier elemento conceptual: **entidades, procesos, acciones...** Un objeto no representa únicamente las características o propiedades, sino que se centra también en los procesos que estos sufren.

DEFINICIÓN

En términos del modelo orientado a objetos, decimos que un **objeto** son datos más operaciones o comportamiento.

- Un objeto es una entidad con ciertas **propiedades** y determinado **comportamiento.**
- En términos de **POO,** las propiedades se conocen como **atributos,** y el conjunto de valores de estas determinan el **estado** del objeto en un determinado momento.
- El comportamiento viene determinado por una serie de funciones y procedimientos que denominamos **métodos** y que modifican el estado del objeto.
- Un objeto tendrá además un **nombre** por el que se identifica.
- Una **clase** es una abstracción de un conjunto de objetos, y un objeto tiene que pertenecer necesariamente a alguna clase.
- Las **clases definen los atributos y métodos** que poseerán los objetos de esta clase.
- Un objeto se entiende como una instancia de una clase.

1.3. Modelo relacional frente a modelo OO

Conceptualmente, el **modelo orientado a objeto** es un modelo dinámico, que se centra en los objetos y en los procesos que estos sufren, pero que no tiene en cuenta, de partida, su **persistencia.** Tenemos que conseguir, pues, poder guardar los estados de los objetos de forma permanente, y cargarlos cuando los necesitemos en la aplicación, así como mantener la **consistencia** entre estos datos almacenados y los objetos que las representan en la aplicación.
Una forma de ofrecer esta persistencia a los objetos sería hacer uso de un SGBD relacional, pero nos encontramos con algunas complicaciones.

La primera, desde el punto de vista conceptual, es que **el modelo entidad-relación se centra en los datos, mientras que el modelo orientado a objetos se centra en los objetos,** entendidos como agrupaciones de datos, y las **operaciones** que se realizan sobre ellos.

Otra diferencia, bastante importante, es la **vinculación de elementos** entre uno y otro modelo. Por un lado, el modelo relacional añade información extra a las tablas en forma de clave externa, mientras que, en el modelo orientado a objetos, no necesitamos estos datos externos, sino que la vinculación entre objetos se hace a través de referencias entre ellos. Un objeto, por ejemplo, tampoco necesitará una clave primaria, puesto que el objeto se identifica por sí mismo.

Las **tablas** en el modelo relacional tienen una clave primaria, para identificar los objetos, y claves externas, para expresar las relaciones, mientras que en el modelo orientado a objetos estas desaparecen, expresando las **relaciones** entre objetos mediante referencias. Además, la forma de expresar estas relaciones también es diferente.

Por otro lado, a la hora de manipular los datos, hay que tener en cuenta que el modelo relacional dispone de lenguajes (SQL principalmente) pensados solo para esta finalidad, mientras que en un lenguaje orientado a objetos se trabaja de forma diferente, y por eso habrá que incorporar mecanismos que permiten realizar desde el lenguaje de programación estas consultas.

Además, cuando obtenemos los resultados de la consulta, nos encontramos también con otro problema, y es la **conversión de resultados.** Una consulta a una base de datos siempre devuelve un resultado **en forma de tabla,** por lo que habrá que transformar estas en estados de los objetos de la aplicación.

Otras características que habrá que tener en cuenta son las propias de la **POO** que no son modeladas con el **modelo E/R:**

Encapsulación

El diseño orientado a objetos permite ocultar el interior de los objetos, declarando los atributos como privados y accediendo a ellos a través de los métodos públicos. Este acceso a través de las tablas es mucho más complejo de conseguir.

Herencia y polimorfismo

La herencia aún puede simularse ligeramente mediante las especializaciones de las entidades. El polimorfismo no puede modelarse.

Tipos estructurados

Los sistemas gestores de bases de datos relacionales obligan a declarar tipos atómicos, y no pueden generar tablas a partir de otras previamente construidas.

Todas estas diferencias suponen lo que se conoce como **desfase objeto-relacional,** y que nos obligará a hacer determinadas conversiones entre objetos y tablas cuando queramos guardar la información en un SGBDR.

En este apartado y los siguientes, veremos cómo superar este desfase desde diferentes aproximaciones.

02 Gestión de conectores

2.1. Protocolos de acceso a bases de datos. JDBC

Cuando hablamos de protocolos de acceso a bases de datos, nos encontramos con dos normas principales de conexión:

- **ODBC *(open data base connectivity):*** se trata de una **API *(application programming interface)*** desarrollada por Microsoft para sistemas Windows, que permite añadir diferentes conectores a varias bases de datos relacionales basadas en SQL, de forma sencilla y transparente. Mediante ODBC, las aplicaciones pueden abrir conexiones con la base de datos, enviar consultas y actualizaciones y gestionar los resultados.
- **JDBC *(Java data base connectivity):*** define una API multiplataforma, que podemos utilizar en nuestros programas en Java para la conexión a los SGBDR.

Por ser multiplataforma y por estar orientada a Java, trabajaremos con esta última. Mediante JDBC:

- Se ofrece una **API,** encapsulada en clases, que garantiza uniformidad en la forma en que las aplicaciones se conectan a la base de datos, independientemente del SGBDR subyacente.
- Necesitaremos un controlador por cada SGBDR al que nos queramos controlar.

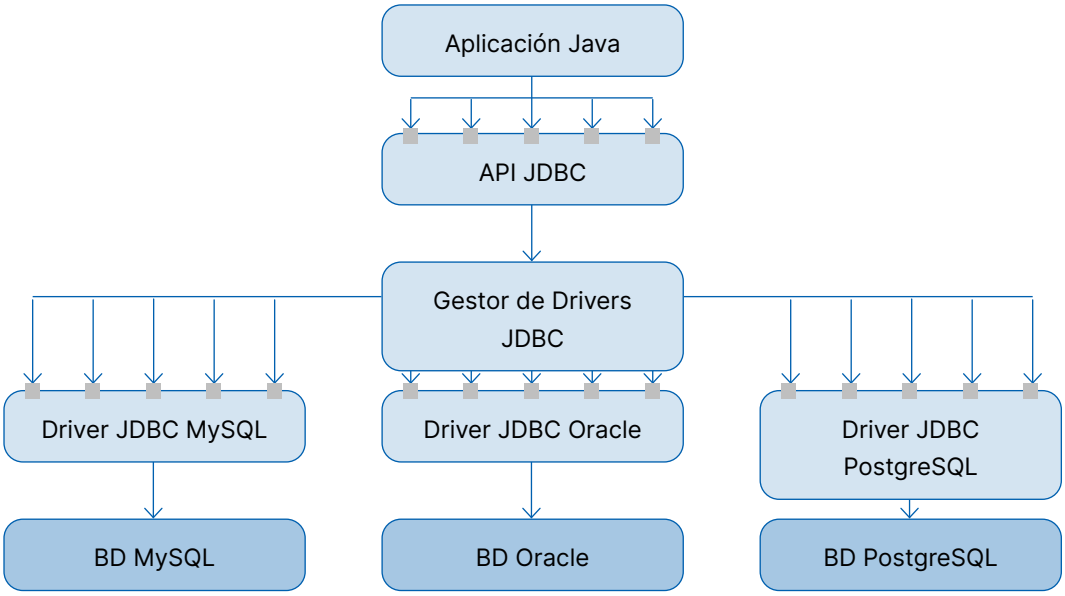
Java no dispone de ninguna biblioteca específica ODBC, pero, para no perder el potencial de estas conexiones, sí se incorporaron unos drivers especiales que actuan de adaptadores entre JDBC y ODBC, de forma que se permite, mediante este puente, conectar cualquier aplicación Java con cualquier conexión ODBC. Actualmente, casi todos los SGBD disponen de drivers JDBC. En caso de no disponer, podemos recurrir a este último.

2.2. Arquitectura de JDBC

La **biblioteca estándar de JDBC** ofrece un conjunto de interfaces sin implementación. De esta implementación se encargarán los controladores o drivers de cada SGBD en cuestión. Las **aplicaciones,** para acceder a la base de datos, tendrán que utilizar las interfaces de JDBC, de forma que para la aplicación sea totalmente transparente la implementación que realiza cada SGBD.
Como vemos, las aplicaciones Java acceden a los diferentes métodos que especifica el API en forma de interfaces, pero son los controladores quienes acceden a la base de datos.
Hay que decir que las aplicaciones pueden utilizar varios controladores JDBC de forma simultánea, y acceder, por lo tanto, a varias bases de datos.

IMPORTANTE

La aplicación especificará un **controlador JDBC** mediante una URL (localizador universal de recursos) al gestor de drivers, y este es quien se encarga de establecer de forma correcta las conexiones con las bases de datos a través de los controladores.



Esquema 1. Arquitectura de JDBC.

IMPORTANTE

Tal y como está construido el mecanismo de conexión, de manera modular, el cambio de driver y conector no afecta a la lógica del programa, ya que la API JDBC es la misma, independientemente del SGBDR.

Los controladores pueden ser de diferentes tipos:

Tipo I o controladores puente

Caracterizados por hacer uso de tecnología externa a JDBC y actuar de adaptador entre JDBC y la tecnología concreta utilizada. Un ejemplo es el puente JDBC-ODBC.

Tipo II o controladores con API parcialmente nativa o controladores nativos

Están formados por una parte Java y por otra que hace uso de bibliotecas del sistema operativo. Su uso se debe a algunos SGBD que incorporan conectores propietarios que no siguen ningún estándar (suelen ser anteriores a ODBC/JDBC).

Tipo III o controladores Java vía protocolo de red

Son controladores desarrollados en Java que traducen las llamadas JDBC a un protocolo de red contra un servidor intermedio. Se trata de un sistema muy flexible, puesto que los cambios en la implementación de la base de datos no afectan las aplicaciones.

Tipo IV o Java puros 100 %

También denominados «de protocolo nativo»; se trata de controladores escritos totalmente en Java. Las peticiones al SGBD se hacen a través del protocolo de red que utiliza el propio SGBD, por lo que no se necesita código nativo al cliente ni un servidor intermediario. Es la alternativa que ha acabado imponiéndose, dado que no requiere ningún tipo de instalación.

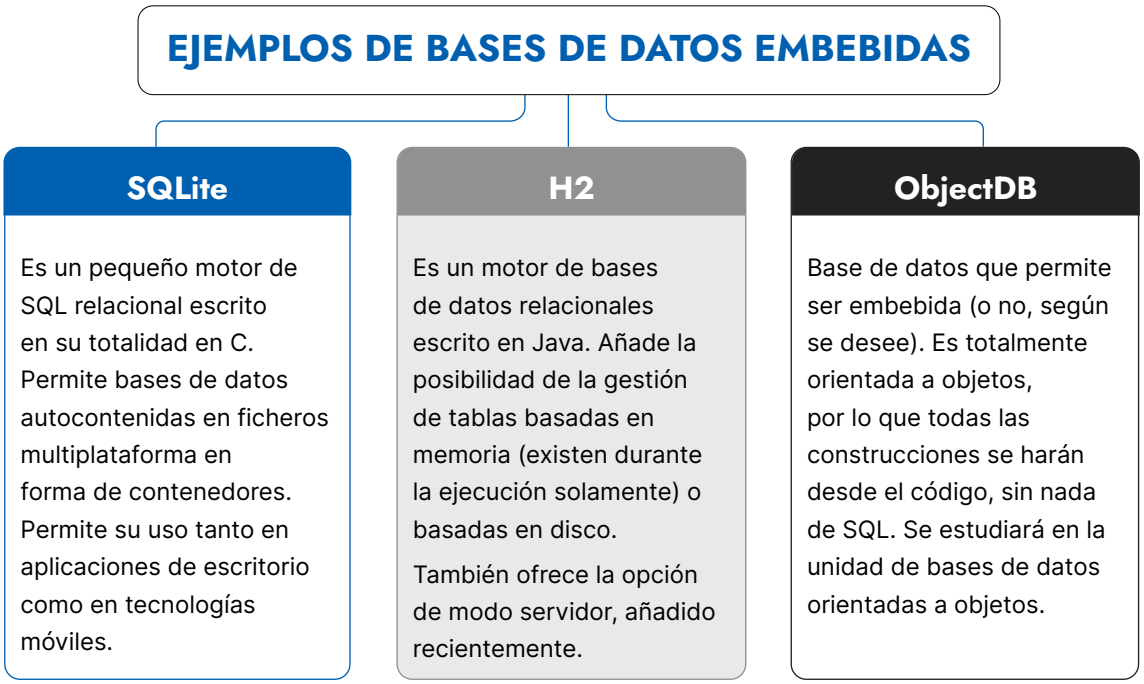
En la infografía (Arquitectura de JDBC), fijémonos, además, en que son los fabricantes los que deben proporcionar el driver, que es quien adapta las peticiones de JDBC a su propia BD.

2.3. Bases de datos embebidas

Hoy en día existen lo que se denominan bases de datos embebidas. A efectos prácticos, son bases de datos que eliminan el servidor de la ecuación, por lo que **no necesitan el sistema gestor.** Son lo que se conoce como bases de datos en un único fichero, que almacenan tanto la estructura de la información como la propia información.

Dicho fichero se almacena en la propia estructura del proyecto, lo que provoca que el acceso sea muy rápido y eficaz, pero perdemos la potencia de toda la gestión que nos ofrecen los servidores. El mayor uso de estas bases de datos está apareciendo en los dispositivos móviles, por la propia encapsulación de las aplicaciones.

De cara al programador, dado que al trabajar con JDBC nos oculta que la BD está embebida, en el proyecto no cambiará nada.



Esquema 2. Esquema de bases de datos embebidas.

03 Conexión a la base de datos

3.1. Establecimiento de la conexión

Para crear una conexión a la base de datos, una vez cargado el driver, deberemos especificar dos conceptos básicos, junto con algunas opciones más. Estos mismos datos son los que nos solicita cualquier cliente para conectarse al servidor:

HOST

Debemos indicar la dirección IP o el nombre de la máquina donde se está ejecutando el servidor.

PUERTO

Habitualmente **MySQL** escucha por el puerto 3306, y Postgres por el 5432, aunque dicho puerto puede modificarse por decisiones de seguridad o por tener varios servidores ejecutándose a la vez.

Toda esta información la recogeremos en lo que se denomina **«cadena de conexión»,** que será un **string** con el contenido en un cierto formato. Esta cadena de conexión tendrá un formato **URL** donde el protocolo será JDBC en vez del clásico HTTP o FTP. Después del protocolo JDBC, se indica el driver (en nuestro caso. **MySQL**) y, a continuación, el resto de los cuatro conceptos indicados.

En Java, la clase necesaria para gestionar el driver es **`java.sql.DriverManager.`** Los drivers los intenta cargar del sistema al leer la propiedad **`jdbc.drivers`**, pero podemos indicar que está cargado mediante la instrucción:

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

La clase que centralizará todas las operaciones con la base de datos es **`java.sql.Connection`**, y la debemos obtener del **`DriverManager`** con cualquiera de los tres métodos estáticos que tiene:

`static Connection getConnection(String url)`	Retorna una conexión, si es posible, a la base de datos cuyos parámetros están especificados en la URL.
`static Connection getConnection(String url, Properties info)`	Retorna una conexión, si es posible, a la base de datos; algunos parámetros están especificados en la URL y otros en un objeto de propiedades **`(Properties)`**.
`static Connection getConnection(String url, String user, String password)`	Retorna una conexión, si es posible, a la base de datos cuyos parámetros están especificados en la URL. Los datos de usuario y contraseña se suministran en dos parámetros adicionales

VÍDEO LAB

Para consultar el siguiente vídeo sobre Conexión a bases de datos, escanea el código QR o pulsa aquí.

EJEMPLO

Un ejemplo básico de conexión sería:

```
String connectionUrl = "jdbc:mysql://localhost:3308";
Connection conn = DriverManager.getConnection(connectionUrl);
```

Aquí vemos que vamos a utilizar el conector JDBC para el driver **MySQL**. El SGBD estará escuchando en nuestro **`localhost`**, en el puerto 3308. Como podemos intuir, esta conexión fallará, debido a que no nos hemos autenticado, y, hoy en día, aparte de las bases de datos embebidas, en prácticamente todos los SGBD debemos hacerlo. Una posibilidad será utilizar el tercer método que ofrece:

```
String connectionUrl = "jdbc:mysql://localhost:3308";
Connection conn =
DriverManager.getConnection(connectionUrl,"root","toor");
```

De este modo, ya nos validamos con el usuario root y la contraseña toor.

3.2. Parámetros de la conexión

En la cadena de conexión, de manera obligatoria tenemos que indicar el **host** y el **puerto**.

Podemos además determinar ya una base de datos concreta marcándola como activa. Para realizarlo, debemos añadir al terminar la URL de conexión una barra más el nombre de la base de datos.

Así, quedaría:

```
String connectionUrl = "jdbc:mysql://localhost:3308/BDJuegos";
Connection conn =
DriverManager.getConnection(connectionUrl,"root","toor");
```

Permite conectarse al servidor de la máquina local en el **puerto 3308** y marcar como activa la base de datos denominada **`BDJuegos`**.

IMPORTANTE

Este mecanismo de conexión no garantiza ni comprueba que la base de datos exista. Solo dará error en caso de algún error en cuanto al puerto, servidor o datos de credenciales de los usuarios.

MÁS PROPIEDADES DURANTE LA CONEXIÓN

IMPORTANTE

Para poder indicar más propiedades a la conexión o parámetros adicionales, deben indicarse de la misma manera que se indican los parámetros con las peticiones **`GET`** y **`PUT`**, siguiendo el protocolo HTTP. Dichos parámetros se indican inicialmente con el símbolo «?», seguido de una lista de pares **`atributo=valor`**.

EJEMPLO

Un primer ejemplo de una URL de conexión completa sería:

```
String connectionUrl =
"jdbc:mysql://localhost:3308/
BDJuegos?user=root&password=toor";
Connection conn = DriverManager.getConnection(connectionUrl);
```

Como puede observarse, en el método **DriverManager.getConnection** no se pasan más atributos, ya que dicha información está en la cadena de conexión. Con más atributos aún:

```
String connectionUrl =
"jdbc:mysql://localhost:3308/
BDJuegos?user=root&password=toor&useUnicode=true
   &characterEncoding=UTF-8";
```

A modo de resumen, aquí podemos consultar la sintaxis completa de la URL de conexión:

jdbc:mysql://[host][,failoverhost...]

```
[:port]/[database]
[?propertyName1][=propertyValue1]
[&propertyName2][=propertyValue2]
[&propertyName3][=propertyValue3]...
```

Como conclusión, podemos parametrizar dicha cadena de conexión, para evitar que esté *hard-coded:*

```
/* Suponemos que las variables siguientes están obtenidas
del fichero de configuración o introducidas por el usuario
directamente */
String usuario= "root";
String passwd=  "toor";
String dbName=  "BDJuegos";

String connectionUrl = "jdbc:mysql://localhost:3308/" + dbName
+"?user=" + usuario + "&password="+ passwd
+"&useUnicode=true&characterEncoding=UTF-8";
Connection conn = DriverManager.getConnection(connectionUrl);
```

CLAVES Y CONSEJOS

Con el término ***hard-coded*** queremos hacer referencia al hecho que se produce cuando la información aparece ***ad hoc*** dentro de nuestro programa, almacenada en variables, en vez de en ficheros de configuración o parámetros que se indican al ejecutar el programa. Es una mala praxis que hay que evitar, ya que reduce el mantenimiento de los programas.

3.3. Organizar y centralizar la conexión

Nuestra aplicación va a conectarse a una base de datos (o a más de una). A dicha base de datos podemos hacerle muchas peticiones, y, si estamos implementando una aplicación **multihílo**, este número de peticiones puede incrementarse mucho. Por ese motivo, debemos tener controlado dónde y cuándo se crean y se cierran las conexiones. Una buena idea es crear una clase que encapsule todos estos procesos.
El esqueleto de dicha clase sería el siguiente:

EJEMPLO

```
public class ConnexioBD {
      private Connection laConnexion = null;
      // variables de acceso a la base de datos
      private void connect() {
          // realizar la conexión. Ojo: método privado. Invocable
  desde dentro
       }
      // cierra la conexión, si está abierta
      public void disConnect() {
           if (laConnexio != null) {
               laConnexio.close();
          }
      }
      // Si no se ha creado, la creo. En cualquier caso, la devuelvo
  public Connection getConexion(){
           if (laConnexio == null) {
               this.connect();
          }
           return this.laConnexio;
      }
}
```

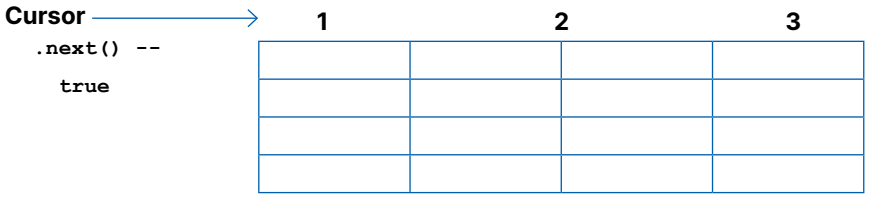
El patrón Singleton es un patrón de diseño de software que se utiliza para garantizar que una clase tenga una sola instancia y proporcionar un punto de acceso global a esa instancia. Esto es útil cuando solo se necesita una única instancia de una clase para coordinar acciones en todo el sistema, como manejar conexiones a una base de datos, registros de eventos o configuraciones globales.
Aquí hay una explicación básica del patrón Singleton:

1. **Constructor privado:** La clase Singleton tiene su constructor marcado como privado para evitar que se pueda instanciar desde fuera de la clase.
2. **Instancia estática:** La clase Singleton tiene una variable estática privada que contiene la única instancia de la clase.
3. **Método estático de acceso:** Se proporciona un método estático público que devuelve la única instancia de la clase. Si la instancia aún no existe, este método crea una nueva instancia y la devuelve; de lo contrario, simplemente devuelve la instancia existente.

04 Metainformación de la base de datos

4.1. El objeto ResultSet

Es el momento de recuperar información, e incluso metainformación. Pero antes de entrar en detalles de cómo hacerlo, vamos a sentar las bases de cómo procesar la información recuperada. Para ello debemos entender cómo nos proporciona el SGBD la información. La respuesta es con una salida en formato tabulado: **`Resultset.`**



IMPORTANTE

Un **`Resultset`** es una tabla resultado de una consulta (habitualmente **`Select`** o similar), con tantas columnas como la selección realizada (entre el **`Select`** y el **`From`**), y tantas filas como registros hayan satisfecho el **`From`** (en caso de haberlo).

Para procesar el resultado de dicho **Resultset,** haremos uso del método **`next()`**, que tiene dos funciones:

- Devuelve **`true`** si quedan resultados por procesar.
- Adelanta el cursor hasta la siguiente fila.

!

El cursor se sitúa antes de la primera fila.

Recuperaremos los elementos de las columnas mediante métodos **`getXXX(int pos)`**, siendo:

- **`XXX`** → el tipo de datos que queremos recuperar: int, double, String, etc.
- **`Pos`** → el número de la columna, empezando la primera por 1.

Opcionalmente, si conocemos el nombre de la columna (en la tabla, o el asignado durante la selección de información), podemos indicarlo mediante un **`String`**.

EJEMPLO

Ejemplos de recuperación de datos dentro de un **`Resultset`**:

`getInt(1)`	Recupera el entero que ocupa la columna 1.
`getString("nombre")`	Recupera el **`String`** de la columna etiquetada como «nombre».
`getObject(3)`	Recupera un objeto (útil) cuando no sabemos el tipo que contiene la columna 3.

CLAVES Y CONSEJOS

Debido al diferente tamaño de las empresas y, por ello, a la mayor o menor complejidad en su gestión, resulta necesario distinguir entre **organización formal** e **informal** de una empresa. En la **organización formal,** la empresa diseña de manera clara e intencionada el modelo de jerarquía y el de organización departamental. En la **organización informal,** el conjunto de las relaciones surge de forma espontánea entre las personas que integran la empresa, sin haber sido provistas por ninguna autoridad

Cuando queremos acceder a los métodos **`getXXX()`** debemos ser cuidadosos, ya que pueden aparecer varios errores; afortunadamente, la mayoría están contemplados por la excepción de Java **`SQLException`**. Esta excepción puede aparecer en algunos de los siguientes casos:

- Cuando el índice de la columna no es correcto (fuera de rango de las columnas del **`ResultSet`**).
- Cuando la etiqueta de la columna es incorrecta.
- Cuando intentamos acceder a un **`ResultSet`** que ha sido ya cerrado.
- Algún error indeterminado del servidor de la base de datos.

Así pues, aunque se verá más adelante, el proceso de trabajar con **`Resultset`** será:

```
Connection laConnexion = DriverManager.
getConnection(connectionUrl, config);
       // recuperamos datos de alguna manera
ResultSet rst= ....
// mientras queden datos
while (rst.next()){
    // procesado de una fila
    // accederemos a cada una de las columnas
 }
```

IMPORTANTE

Dentro del bucle **`while`** no deberemos realizar otro **`next()`**, ya que, en caso de hacerlo, nos saltaremos una fila. Recuerda:

- Antes de acceder a ninguna columna, debemos hacer un **`next()`**. Al principio, el cursor está antes de la primera fila.
- **`next()`** avanza y devuelve true si está situado en una fila después de avanzar.
- Al final del bucle, el cursor está situado al final de la última fila (sin acceso a información).

4.2. Metadatos de la base de datos

Para consultar información de la base de datos, Java dispone de la interfaz **`DatabaseMetaData`**, obtenida a partir de la conexión que tenemos establecida. Así pues, accederemos a ella:

```
Connection laConnexion = DriverManager.
getConnection(connectionUrl, config);
DatabaseMetaData dbmd=laConnexion.getMetaData();
```

Este objeto contiene los métodos siguientes:

MÉTODO	DEVUELVE...
`String getDatabaseProductName()`	El nombre del SGBD.
`String getDriverName()`	El nombre del driver.
`String getURL()`	La URL de conexión.
`String getUserName()`	El nombre de usuario.

`ResultSet getTables(String catalogo, String esquema, String NombreTabla, String[] tipo)`	Un **`Resultset`** con las tablas del catálogo indicado. En catálogo indicamos el nombre de la base de datos de la que queremos recuperar las tablas. **`esquema`** puede dejarse con valor **`null`**. El resto de campos es para filtrados por nombre o por tipo (tabla, vista, etc.). Devuelve una fila por tabla. Para ver las columnas devueltas, consulta la documentación respectiva
`ResultSet getColumns(String catalogo, String esquema, String NombreTabla, String NombreColumna)`	Devuelve un **`Resultset`** con las columnas de la tabla «**`NombreTabla`**», de la base de datos de nombre «**`catalogo`**». El resto a **`null`**. El **`nombreColumna`** es para filtrados. Devuelve una fila por campo. Para ver las columnas devueltas, consulta la documentación respectiva.

Estas tres consultas devuelven, para el catálogo indicado y la tabla indicada, un Resultset con estas características:

MÉTODO	DEVUELVE...
`ResultSet getPrimaryKeys( String catalogo, String esquema, String patronNombreTabla)`	Los campos que son clave principal.
`ResultSet getImportedKeys( String catalogo, String esquema, String patronNombreTabla)`	Los campos que son apuntados por una clave ajena.
`ResultSet getExportedKeys( String catalogo, String esquema, String patronNombreTabla)`	Los campos de donde sale un clave ajena.

Con estos métodos ya podemos consultar lo imprescindible de la metainformación de la base de datos.

05 Consultas a la base de datos

5.1. Operaciones sobre la base de datos

Antes de lanzar una consulta, debemos componer la propia consulta (lo que sería la sentencia SQL). Habitualmente, esta consulta la compondremos utilizando un **`string`** y adicionalmente algunos argumentos, como veremos.
Para crear las sentencias, tenemos las interfaces **`Statement`** y **`PreparedStatement`**, obtenidas a partir de los objetos **`Connection`**. Posteriormente, las ejecutaremos con **`execute()`** (consulta que no genera resultados) o **`executeQuery()`** (consulta que sí genera resultados y deberemos procesarlos).

5.2. Tipos de sentencias

A. Sentencias fijas

DEFINICIÓN

Las **sentencias fijas** son consultas que son «constantes», es decir, que no dependen de ningún argumento. Son las más simples.

Se crea el **`Statement`** y se lanza la consulta.

```
String SQL="Select * from Persona";
Statement stm=laConexion.createStatement();
ResultSet rst=stm.executeQuery(SQL);
while (rst.next()){
    System.out.println(rst.getString("nombre") +
                       " " + rst.getString("apellidos")+
                       ": " + rst.getInt("edad"));
}
```

Como la tabla Persona está formada por los campos **`idPersona`**, **`nombre`**, **`apellidos`** y **`edad`**, podemos acceder a dichas columnas mediante su posición (empezando por la 1) o por su nombre. Fíjate en que, además, para acceder a las columnas de cada una de las filas del **`ResultSet`** utilizamos **`getXXX`**, donde **`XXX`** es el tipo de datos de cada columna. Si lo desconocemos, siempre podemos recurrir a **`getObject()`**.

Otra consulta podría ser una actualización o inserción, como, por ejemplo:

```
String SQL = "Insert into Persona(nombre,apellidos,edad)" +
             " values ('Isabel','Grau Sainz',30);";
Statement stm = laConexion.createStatement();
int filas = stm.executeUpdate(SQL);
if (filas == 1) {
    System.out.println("Inserción realizada con éxito");
} else {
    System.out.println("Error en la inserción");
}
```

Aquí, el detalle es que la sentencia se ejecuta con un executeUpdate(), ya que se modifica la base de datos, y dicha ejecución devuelve un entero, que es el número de filas afectadas. En caso de ser 1, es que la inserción ha tenido éxito.

B. Sentencias variables

En el ejemplo anterior, se presenta el problema de que la sentencia está ***hard-coded*** (valores dentro del código), y ya hemos comentado que esto representa un problema.
Los valores deben estar en variables; por lo tanto, podríamos mejorar esta consulta reescribiendo el **query** de esta manera:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class Main {
    public static void main(String[] args) {
        try {
            // Establecer conexión con la base de datos
            Connection laConexion = DriverManager.
getConnection("url_de_la_base_de_datos");

            // Definición de los datos
            String nombre = Leer.leerTexto("Dime el nombre:
");
            String apellidos = Leer.leerTexto("Dime los
apellidos: ");
            int edad = Leer.leerEntero("Dime la edad: ");
```

```
            // Construcción de la consulta SQL
            String SQL = "Insert into Persona(nombre,
apellidos, edad) " +
                         "values ('" + nombre + "', '" +
apellidos + "', " + edad + ")";

            // Ejecución de la consulta
            Statement stm = laConexion.createStatement();
            int filas = stm.executeUpdate(SQL);

            // Verificación de la inserción
            if (filas == 1) {
                System.out.println("Inserción realizada con
éxito");
            } else {
                System.out.println("Error en la inserción");
            }

            // Cerrar la conexión
            laConexion.close();
        } catch (SQLException e) {
            System.out.println("Error de conexión o consulta
SQL: " + e.getMessage());
        }
    }
}
```

En esta versión:

- Se han agregado las importaciones necesarias al principio del código.
- Se ha incluido una clase **`Main`** con un método **`main`** para ejecutar el código.
- Se ha añadido un bloque **`try-catch`** para manejar posibles excepciones de SQL.
- Se ha definido la conexión a la base de datos (aunque la URL está pendiente de rellenar).
- Se ha mejorado la construcción del SQL utilizando variables en lugar de valores hard-coded.
- Se ha implementado un control de excepciones para posibles errores de conexión o consultas SQL.

CLAVES Y CONSEJOS

Este tipo de consultas debemos evitarlas en sentencias de validación de usuarios; para ello utilizaremos las sentencias preparadas.

C. Sentencias preparadas

Ejemplo de cómo utilizar sentencias preparadas en Java para evitar la inyección SQL:

EJEMPLO

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class Main {
    public static void main(String[] args) {
        try {
            // Establecer conexión con la base de datos
           Connection laConexion = DriverManager.getConnection("url_de_
la_base_de_datos");

            // Definición de los datos
            String nombre = Leer.leerTexto("Dime el nombre: ");
            String apellidos = Leer.leerTexto("Dime los apellidos: ");
            int edad = Leer.leerEntero("Dime la edad: ");

            // Construcción de la consulta SQL con sentencia preparada
             String SQL = "INSERT INTO Persona(nombre, apellidos, edad)
VALUES (?, ?, ?)";
                    PreparedStatement preparedStatement = laConexion.
prepareStatement(SQL);

            // Asignar valores a los placeholders
            preparedStatement.setString(1, nombre);
            preparedStatement.setString(2, apellidos);
            preparedStatement.setInt(3, edad);
```

```
            // Ejecución de la consulta
            int filas = preparedStatement.executeUpdate();

            // Verificación de la inserción
            if (filas == 1) {
                System.out.println("Inserción realizada con éxito");
            } else {
                System.out.println("Error en la inserción");
            }

            // Cerrar la conexión
            laConexion.close();
        } catch (SQLException e) {
             System.out.println("Error de conexión o consulta SQL: " +
e.getMessage());
        }
    }
}
```

En este ejemplo:

- Se ha cambiado la consulta SQL por una sentencia preparada, donde los valores se representan con interrogantes (?).
- Se ha creado un objeto **`PreparedStatement`** a partir de la consulta SQL.
- Se han asignado los valores a los placeholders utilizando métodos como **`setString()`** y **`setInt()`**.
- Se ha ejecutado la consulta con **`executeUpdate()`** sobre el **`PreparedStatement`**.

D. Metadatos de las consultas

Independientemente del tipo de sentencia que ejecutemos (preparada o no), siempre que realicemos una consulta **`(executeQuery())`** el resultado es un **`ResultSet`**, que, como hemos comentado, es una tabla con los datos. Podemos consultar metainformación de esa tabla retornada por la **`Select`**, mediante un objeto que podemos extraer del ResultSet, que es el **`ResultSetMetaData`**.

Dicho objeto contiene:

`int getColumnCount()`	Devuelve cuántas columnas tiene el **`ResultSet`** (campos de la consulta). Muy cómodo en consultas tipo **`Select*`**, que ***a priori*** no sabemos lo que tienen.
`String getColumnName(int index)` `String getColumnTypeName(int index)`	Estos dos métodos nos indican, a partir del índice de la columna, el nombre del campo y el tipo de dato de cada campo.

5.3. Scripts

IMPORTANTE

Un script, que habitualmente tenemos creado en un fichero externo, no es más que un conjunto de **sentencias SQL** ejecutadas en un orden de arriba abajo.

Podríamos coger como estrategia ir leyendo línea a línea el fichero e ir ejecutando, pero JDBC permite ejecutar un conjunto de instrucciones en bloque. Para ello, lo primero que debemos hacer es habilitar la ejecución múltiple, añadiendo un parámetro a la conexión, que es

```
allowMultipleQueries=true.
```

A continuación, deberemos de cargar el fichero y componer un **`string`** con todo el script. Para normalizarlo y hacerlo totalmente portable, deberemos ir con cuidado con los saltos de línea, ya que, dependiendo del sistema, es un «**`\n`**» o una combinación «**`\r\n`**».
Para ello, podemos ir leyendo línea a línea y guardándolo en un **`StringBuilder`**, añadiendo como separadores el `System.getProperty("line.separator")`.
Después, solo necesitaremos crear una sentencia con dicho **`string`** y ejecutarla con un **`executeUpdate()`**. En el ejemplo siete, queda solucionado muy sencillo.

5.4. Transacciones

Si queremos **proteger la integridad de los datos**, así como evitar situaciones de bloqueos inesperados en aplicaciones multihílo, deberemos proteger nuestras operaciones, especialmente las que modifiquen datos mediante el uso de transacciones.

DEFINICIÓN

Una **transacción** define un entorno de ejecución en el que las operaciones de guardado se quedan almacenadas en memoria hasta que finalice esta. Si en un determinado momento algo falla, se devuelve el estado al punto inicial de la misma, o algún punto de marcado intermedio. **Por defecto, al abrir una conexión se inicia una transacción.**

Cada ejecución sobre la conexión genera una transacción sobre sí misma. Si queremos deshabilitar esta opción para que la transacción englobe varias ejecuciones, deberemos marcarlo mediante

```
laConexion.setAutoCommit(false);
```

Para aceptar definitivamente la transacción, lo realizaremos mediante `laConexion.commit();` y para cancelar la transacción, `laConexion.rollback();`

5.5. ResultSet actualizables

IMPORTANTE

Los **`Resultset`** que obtenemos de las consultas, por lo general, nos servirán para cargar datos que mostrar en nuestros programas.

Muchas veces, dichos datos serán modificados, y, por lo tanto, aparte de cargar los datos, deberemos guardar la información. Ahí es donde aparecen los **`ResultSet`** actualizables, que dependerán del modo que se creó la sentencia (preparada o no), con la sintaxis:

```
public abstract Statement createStatement(
    int arg0,    // resultSetType
    int arg1,    // resultSetConcurrency
    int arg2)    // resultSetHoldability
    throws SQLException
```

Donde **ResultSetType**:

TYPE_FORWARD_ONLY	Por defecto. Un solo recorrido.
TYPE_SCROLL_INSENSITIVE	Permite ***scroll*** o rebobinado a posición absoluta o relativa. Los datos son los que se cargan en la apertura. MySQL solo soporta este (versión 8).
TYPE_SCROLL_SENSITIVE	Permiten ***scroll***, y, si hay modificaciones en la base de datos, los cambios son visibles en el **ResultSet**.

IMPORTANTE

No todos los SGBD soportan todos los tipos. Puede consultarse con **DatabaseMetaData.supportsResultSetType(type).**

Donde **ResultSetConcurrency**

CONCUR_READ_ONLY	Solo se soporta lectura. Cambios solo con update. Opción por defecto en MySQL.
CONCUR_UPDATABLE	Permite actualizar el **ResultSet**.

Donde **ResultSetHoldability** determina el comportamiento cuando se cierra una transacción con un **commit:**

HOLD_CURSORS_OVER_COMMIT	El **ResultSet** no se cierra al finalizar la transacción.
CLOSE_CURSORS_AT_COMMIT	El **ResultSet** se cierra. Mejora el rendimiento.

Como hemos visto, el cursor no solo admite avanzar.

- **`Next`**, **`previous`**, **`first`**, **`last`**: adelante, atrás, al principio o al final: devuelven **`true`** si se ha posicionado sobre una fila y **`false`** si se ha salido del **`ResultSet`**.
- **`beforeFirst`** y **`afterLast`**: se sitúan antes del primero o después del último.
- **`relative(int n): se desplaza n`** filas hacia adelante.
- **`absolute(int n): se sitúa en la fila n.`**

A. Borrados

Después de situar el cursor sobre la fila que vamos a eliminar, podemos eliminarla del **`ResultSet`** (y de la base de datos) con el método **`.deleteRow()`**. Al borrar una fila, el cursor quedará apuntando a la fila anterior a la borrada.

B. Actualizaciones

Después de situar el cursor sobre la fila deseada, debemos actualizar las columnas que queremos, mediante el método **`updateXXX(int,nuevoValor)`**, donde se asigna a la columna **i-ésima** (o con su nombre) el valor nuevo valor del tipo **`XXX`**. Modificados todos las columnas, se guardan los cambios con **`updateRow()`**.

C. Inserciones

Si queremos insertar una nueva fila en un **`ResultSet`**, primero debemos generarla en blanco, y esto se consigue con el método **`rst.moveToInsertRow()`**, que crea una fila «virtual» en blanco. Sobre esta fila le aplicamos los métodos **`updateXXX(int,nuevoValor)`**, y finalmente procederemos a insertar la nueva fila con **`rst.insertRow()`**.

IMPORTANTE

- Estas operaciones de actualización, borrado e inserción solo pueden realizarse sobre consultas que tienen origen en una tabla sin agrupaciones.
- Para evitar complejidad en nuestros programas, cabe valorar la conveniencia de «traducir» las actualizaciones de **`ResultSet`** a SQL puro y ejecutarlo nativamente en la bases de datos mediante nuevas sentencias.

UAX FP