Diseño y Desarrollo de Sistemas de Información

Operaciones CRUD con JDBC

- Mapeo objeto-relacional
- Sentencias "preparadas"
- *ResultSet*

Consulta a una base de datos

- La clase *Statement* permite realizar las operaciones CRUD sobre una base de datos mediante JDBC
- Un objeto de clase *Statement* se instancia con el método *createStatement()* de la clase *Connection*

```
Statement stmt = conexion.getConexion().createStatement();
```

Es conveniente implementar, en la clase *Conexion,* un método (por ejemplo, *getConexion()*) que devuelva el atributo de tipo *Connection*

- La consulta se realiza con el método *executeQuery()* de la clase *Statement*

```
ResultSet resultado = stmt.executeQuery ("select * from T");
```

- Este método devuelve un objeto de tipo *ResultSet* (conjunto de filas y columnas obtenidas del resultado de una consulta)

Consulta a una base de datos

- El objeto *ResultSet* dispone de un cursor que se posiciona en el registro (fila) que podemos consultar en cada momento. La primera vez estará en una posición anterior a la primera fila

- El método *next()* de *ResultSet* mueve el cursor a la siguiente fila. Devuelve true mientras pueda avanzar al siguiente registro, y false en el caso de llegar al último registro

- Para el recorrido de todos los registros devueltos por una consulta se usa, generalmente, un bucle similar a este:

```
while (resultado.next()) {
        // Realizar operaciones
}
```

Consulta a una base de datos

- Para obtener los datos del registro en el que está situado el cursor, se usan los métodos *getTipo(campo)* donde *Tipo* es el tipo de datos de Java que va a recibir el valor del campo
- Para especificar el campo se puede usar su propio nombre o el índice correspondiente según el orden de los campos de la consulta

Tipo Standard SQL	Método *get()*
CHAR	getString
VARCHAR	getString
SMALLINT	getShort
INTEGER	getInt
FLOAT	getFloat/getDouble
DOUBLE	getDouble
DECIMAL	getDecimal
DATE	getDate
MONEY	getDouble
TIME	getTime

getString() se puede aplicar para recuperar cualquier tipo SQL

Ejemplo

```
Statement stmt = conexion.getConexion().createStatement();
ResultSet rs = stmt.executeQuery("select * from T");
while (rs.next()) {
    int v1 = rs.getInt(1);
    String v2 = rs.getString(2);
    System.out.println(v1 + "   " + v2);
}
stmt.close();
```

Suponemos que la tabla T tiene dos campos: uno de tipo "entero" y otro de tipo "cadena"

Sentencias de modificación

- Operaciones: INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE, etc.
- El método *executeUpdate()* de la clase ***Statement*** es el que realiza las sentencias de modificación en la base de datos
- Devuelve un valor entero para indicar el número de filas afectadas o 0 si se lanza una sentencia LDD

```
Statement stmt.executeUpdate("cadena con la sentencia SQL");
```

```
String sentenciaCreacion =
"CREATE TABLE ESTUDIANTE (
     dni CHAR(9),
     nombre VARCHAR2(32),
     sexo CHAR(1)  )";

stmt.executeUpdate(sentenciaCreacion);
```

```
String sentenciaInsercion =
"INSERT INTO ESTUDIANTE VALUES
('12857876F', 'Julián', 'M')";

stmt.executeUpdate(sentenciaInsercion);
```

Las sentencias SQL se pasan como parámetros de tipo "cadena"
Hay que recordar que SQL utiliza comilla simple (') y no doble (") para los tipos char y varchar

Sentencias preparadas y parametrizadas

- En general, lo habitual es diseñar consultas genéricas y parametrizadas a las que, posteriormente, se le asignan los valores de los parámetros
- Para ello se usa la clase *PreparedStatement*, que es una extensión de la clase *Statement*
- Además, las sentencias preparadas previenen el problema de *SQL Injection,* puesto que los valores de los parámetros son tratados de manera segura y no se interpretan como parte de la consulta SQL

```
PreparedStatement ps = null;
ps = conexion.getConexion().prepareStatement ("UPDATE CAFE SET precio = ? WHERE nombre = ?");

ps.setInt (1, 75);
ps.setString (2, "Saimaza");
ps.executeUpdate();
```

```
PreparedStatement ps = null;
ps = conexion.getConexion().prepareStatement("INSERT INTO PERSONA VALUES (?,?,?)")

ps.setString (1, "24543117P");
ps.setString (2, "Laura");
ps.setInt (3, 35);
ps.executeUpdate();
```

Mapeo objeto-relacional de la base de datos

- El mapeo objeto-relacional (ORM - *Object-Relational Mapping*) es una técnica de programación que se utiliza para convertir datos entre el sistema de tipos de un lenguaje de programación orientado a objetos y el de una base de datos relacional, para mantener la persistencia de datos

- Existe software comercial y de uso libre que implementa el mapeo relacional de objetos, aunque en muchas ocasiones resulta conveniente crear las propias herramientas ORM

Mapeo objeto-relacional de la base de datos

- En la capa Modelo del proyecto se crea una clase por cada una de las tablas del esquema de base de datos. Por ejemplo, para la tabla MONITOR, tendremos la clase:

```
public class Monitor {
    private String codMonitor;
    private String nombre;
    private String dni;
    private String telefono;
    private String correo;
    private String fechaEntrada;
    private String nick;

    // Constructor por defecto
    public Monitor() {
        this.codMonitor = null;
        this.nombre = null;
        this.dni = null;
        this.telefono = null;
        this.correo = null;
        this.fechaEntrada = null;
        this.nick = null;
        };

    // Constructor con parámetros
    public Monitor(String codMonitor, String nombre, String dni,
            String telefono, String correo, String fechaEntrada, String nick) {
        this.codMonitor = codMonitor;
        this.nombre = nombre;
        this.dni = dni;
        this.telefono = telefono;
        this.correo = correo;
        this.fechaEntrada = fechaEntrada;
        this.nick = nick;
    }
```

Mapeo objeto-relacional de la base de datos

- Además de los atributos y constructores, la clase debe implementar métodos para consultar y modificar los valores de los atributos (campos en la base de datos). Estos métodos se conocen como "*getters*" y "*setters*"
- El IDE de *NetBeans* tiene una utilidad (dentro de la opción "***insert code***") para añadir, de forma automática, tanto los constructores como las funciones *get()* y *set()*

```
public String getCodMonitor() {
    return codMonitor;
}

public void setCodMonitor(String codMonitor) {
    this.codMonitor = codMonitor;
}

public String getNombre() {
    return nombre;
}

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

public String getDni() {
    return dni;
}

public void setDni(String dni) {
    this.dni = dni;
}
```

Algunas de las funciones *get()* y *set()* de la clase Monitor

Gestión de las operaciones CRUD

- En la capa Modelo se programarán también las clases necesarias para realizar las operaciones de consulta, inserción, actualización y borrado, así como otras operaciones que se necesiten para gestionar los datos

- De forma general, programaremos una clase para gestionar los datos de cada una de las tablas, que llamaremos nombreTablaDAO

- Estas clases serán las encargadas de comunicar los controladores con la base de datos

- Tendrán un atributo de tipo Conexion y sus constructores recibirán, como parámetro, el objeto Conexion de la aplicación

```
public class MonitorDAO {
    Conexion conexion = null;
    PreparedStatement ps = null;

    public MonitorDAO(Conexion c) {
        this.conexion = c;
    }
```

También tendrán un atributo de tipo *PreparedStatement* para realizar las consultas

Ejemplo de función para recuperar toda la información de una tabla

- Este método formará parte de la clase MonitorDAO.java

```
public ArrayList<Monitor> listaMonitores() throws SQLException {
    ArrayList listaMonitores = new ArrayList();

    String consulta = "SELECT * FROM MONITOR";
    ps = conexion.getConexion().prepareStatement(consulta);
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        Monitor monitor = new Monitor(rs.getString(1), rs.getString(2),
                rs.getString(3), rs.getString(4), rs.getString(5),
                rs.getString(6), rs.getString(7));
        listaMonitores.add(monitor);
    }

    return listaMonitores;
}
```

Ejemplo de función parametrizada para recuperar información

- Este método recupera los monitores cuyo nombre empiece por la letra que se pasa por parámetro

```
public ArrayList<Monitor> listaMonitorPorLetra(String letra)
        throws SQLException {
    ArrayList listaMonitores = new ArrayList();

    String consulta = "SELECT * FROM MONITOR WHERE nombre LIKE ?";
    ps = conexion.getConexion().prepareStatement(consulta);
    letra = letra + "%";
    ps.setString(1, letra);
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
        Monitor monitor = new Monitor(rs.getString(1), rs.getString(2),
                rs.getString(3), rs.getString(4), rs.getString(5),
                rs.getString(6), rs.getString(7));
        listaMonitores.add(monitor);
    }

    return listaMonitores;
}
```

Ejemplo de llamada a un método DAO desde un controlador

```
switch (opcion) {
    case "1":
    try {
        ArrayList<Socio> lSocios = pideSocios();
        vSocios.muestraSocios_Numero_Nombre(lSocios);
    } catch (SQLException e) {
        vMensajes.mensajeConsola(texto: "Error en la petición", error: e.getMessage());
    }

    break;
```

```
private ArrayList<Socio> pideSocios() throws SQLException {
    return (socioDAO.listaSocios());
}
```