Diseño y Desarrollo de Sistemas de Información

Operaciones CRUD con JDBC

- Mapeo objeto-relacional
- Sentencias "preparadas"
- *ResultSet*
- Los componentes *JPanel* y *JTable*
- Gestión de eventos de ratón
- *Look and Feel*

Consulta a una base de datos

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

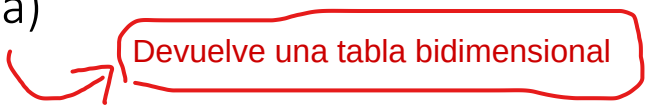
```
Statement stmt = conexion.getConexion().createStatement();
```

En la clase *Conexion* debemos tener un método *getConexion()* que devuelva el objeto 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 obtenidas del resultado de una consulta)



Consulta a una base de datos

- El objeto *ResultSet* dispone de un cursor que se sitúa 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 la consulta se usa, normalmente, un bucle como 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 *getXXXX(campo)* donde XXXX es el tipo de datos de Java en el que queremos que nos devuelva 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();
```

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 que indicar el número de filas afectadas o 0 si se usa una sentencia LDD

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

```
String sentenciaCreacion =
"CREATE TABLE ESTUDIANTE (
     dni CHAR(9),
     nombre VARCHAR(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 de caracteres"
Hay que recordar que SQL utiliza comilla simple (') y no doble (") para los tipos char y varchar

Sentencias preparadas y parametrizadas

- En general, lo más 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*
- Las sentencias preparadas previenen el problema de SQL Injection

```
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 para convertir los 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 implementan 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 incluiremos una clase por cada una de las tablas de nuestro esquema de base de datos. Por ejemplo, para la tabla MONITOR, tendremos la clase:

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

    // Constructor por defecto
    public Monitor() {
            codMonitor = null;
            nombre = null;
            dni = null;
            telefono = null;
            correo = null;
            fechaEntrada = null;
            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 funciones para consultar y modificar los valores de los atributos (campos en la base de datos). Estas funciones 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 las clases necesarias para realizar las operaciones de consulta, inserción, actualización y borrado, así como otras operaciones que se necesiten para acceder a 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 el controlador

```java
@Override
public void actionPerformed(ActionEvent e) {
    switch (e.getActionCommand()) {
        case "SalirAplicacion":
            vPrincipal.dispose();
            System.exit(0);
            break;
        case "GestionMonitores":
            vMonitor.setVisible(true);
            vSocio.setVisible(false);
            pPrincipal.setVisible(false);
            utilTablas.dibujarTablaMonitores(vMonitor);

            try {
                    pideMonitores();
                } catch (SQLException ex) {
                    vMensaje.Mensaje("error", "Error en la petición\n"
                            + ex.getMessage());
                }
            break;
```

La función para "dibujar" un *jTable* se verá en las siguientes diapositivas

```java
private void pideMonitores() throws SQLException {
    ArrayList<Monitor> lMonitores = monitorDAO.listaMonitores();
    utilTablas.vaciarTablaMonitores();
    utilTablas.rellenarTablaMonitores(lMonitores);
}
```

Las funciones para "vaciar" y "rellenar" un *jTable* se verán en las siguientes diapositivas

El contenedor *JPanel*

- *JPanel* es un contenedor que puede albergar componentes gráficos
- Será el tipo de contenedor que usaremos para diseñar las pantallas y se añadirán a la ventana principal (*JFrame*) de la aplicación
- Para crear un nuevo *JPanel* solo será necesario indicarlo al crear una nueva clase en el IDE
- Una vez creado, se añadirá a la ventana principal de esta forma:

```
vPrincipal.getContentPane().setLayout(new CardLayout());
vPrincipal.add(vMonitor);
```

Usaremos el *Layout CardLayout* para poder tener más de un panel en la misma posición, mostrándolos y ocultándolos según las necesidades de la aplicación

- Este código forma parte del constructor del controlador. Previamente se habrá declarado e instanciado el objeto vMonitor de tipo VistaMonitor (clase de un *JPanel*)

El componente *JTable*

- Un *JTable* es un componente que permite dibujar una tabla
- Una de las formas más rápidas y sencillas de utilizar un *JTable* teniendo toda su funcionalidad consiste en instanciar, como modelo de datos, un *DefaultTableModel* y luego un *JTable* , pasándole el modelo en el constructor

```
DefaultTableModel modelo = new DefaultTableModel();
JTable tabla = new JTable();
tabla.setModel(modelo);
```

- En nuestro proyecto:

```
// Se sobreescribe el método isCellEditable para hacer que las filas
// no se puedan editar al hacer doble click
public DefaultTableModel modeloTablaMonitores = new DefaultTableModel() {
    @Override
    public boolean isCellEditable(int row, int column) {
        return false;
    }
};
```

El objeto *JTable* lo diseñaremos con la herramienta de diseño del IDE Netbeans. Por tanto, la sentencia *JTable jTableMonitores = new JTable()* estará en el código de la vista diseñada para la gestión de los monitores

El componente *JTable*

- Para diseñar este componente únicamente lo situaremos en el panel correspondiente. El resto del diseño y su operatividad se programará
- *DefaultTableModel* tiene todos los métodos necesarios para modificar los datos de la tabla que contiene, añadir filas o columnas, asignarle un nombre a cada columna, etc.
- Ejemplo de función para "dibujar" una tabla

jTableMonitores es un objeto de tipo *JTable* que se encuentra en la vista *VistaMonitores*

```
public void dibujarTablaMonitores(VistaMonitores vMonitor) {
    vMonitor.jTableMonitores.setModel(modeloTablaMonitores);

    String[] columnasTabla = {"Código", "Nombre", "DNI",
        "Teléfono", "Correo", "Fecha Incorporación", "Nick"};
    modeloTablaMonitores.setColumnIdentifiers(columnasTabla);

    // Para no permitir el redimensionamiento de las columnas con el ratón
    vMonitor.jTableMonitores.getTableHeader().setResizingAllowed(false);
    vMonitor.jTableMonitores.setAutoResizeMode(JTable.AUTO_RESIZE_LAST_COLUMN);

    // Así se fija el ancho de las columnas
    vMonitor.jTableMonitores.getColumnModel().getColumn(0).setPreferredWidth(40);
    vMonitor.jTableMonitores.getColumnModel().getColumn(1).setPreferredWidth(240);
    vMonitor.jTableMonitores.getColumnModel().getColumn(2).setPreferredWidth(70);
    vMonitor.jTableMonitores.getColumnModel().getColumn(3).setPreferredWidth(70);
    vMonitor.jTableMonitores.getColumnModel().getColumn(4).setPreferredWidth(200);
    vMonitor.jTableMonitores.getColumnModel().getColumn(5).setPreferredWidth(150);
    vMonitor.jTableMonitores.getColumnModel().getColumn(6).setPreferredWidth(60);
}
```

El componente *JTable*

- Ejemplo de función para mostrar los datos de la tabla MONITOR a partir de una lista de monitores

```
public void rellenarTablaMonitores(ArrayList<Monitor> monitores) {
    Object[] fila = new Object[7];
    int numRegistros = monitores.size();
    for (int i = 0; i < numRegistros; i++) {
        fila[0] = monitores.get(i).getCodMonitor();
        fila[1] = monitores.get(i).getNombre();
        fila[2] = monitores.get(i).getDni();
        fila[3] = monitores.get(i).getTelefono();
        fila[4] = monitores.get(i).getCorreo();
        fila[5] = monitores.get(i).getFechaEntrada();
        fila[6] = monitores.get(i).getNick();
        modeloTablaMonitores.addRow(fila);
    }
}
```

- Ejemplo de función para vaciar el contenido de la tabla MONITOR

```
public void vaciarTablaMonitores() {
    while (modeloTablaMonitores.getRowCount() > 0) {
        modeloTablaMonitores.removeRow(0);
    }
}
```

El componente *JTable*

- Ejemplo para solicitar los datos de la tabla MONITOR y ponerlos en el *JTable* correspondiente

```
@Override
public void actionPerformed(ActionEvent e) {
    switch (e.getActionCommand()) {
        case "SalirAplicacion":
            vPrincipal.dispose();
            System.exit(0);
            break;
        case "GestionMonitores":
            vMonitor.setVisible(true);
            vSocio.setVisible(false);
            pPrincipal.setVisible(false);
            utilTablas.dibujarTablaMonitores(vMonitor);

            try {
                    pideMonitores();
                } catch (SQLException ex) {
                    vMensaje.Mensaje("error", "Error en la petición\n"
                            + ex.getMessage());
                }
            break;
```

Las funciones *dibujarTablaMonitores()*, *vaciarTablaMonitores()* y *rellenarTablaMonitores()* son métodos de una clase propia llamada *utilTablas*. En este caso se ha decidido agrupar estas funciones en una clase independiente para hacer más legible el código del controlador

```
private void pideMonitores() throws SQLException {
    ArrayList<Monitor> lMonitores = monitorDAO.listaMonitores();
    utilTablas.vaciarTablaMonitores();
    utilTablas.rellenarTablaMonitores(lMonitores);
}
```

Gestión de eventos con "asignación directa"

- Otra forma de que un componente responda a un evento consiste en asignárselo directamente de la siguiente forma:

```
// En este caso, se le añaden los listener de Mouse a la tabla jTableMonitores
vMonitor.jTableMonitores.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent evt) {
        vMonitorjTableMonitoresMouseClicked(evt);
    }
});
```

- Y programar el código correspondiente que debe ejecutarse cuando se produzca el evento:

```
private void vMonitorjTableMonitoresMouseClicked(MouseEvent evt) {
    int fila = vMonitor.jTableMonitores.getSelectedRow();
    // TODO - Rellenar todos los textfields con los valores de las columnas
    // de la fila que esté señalada en la tabla cuando se haga un click de ratón
}
```

Look and Feel

- El *look and feel* (aspecto y comportamiento) es la forma en que los componentes de Swing (*JLabel*, *JButton*, *JTextField*, *JTable*, *JComboBox*, etc) se muestran dentro de una interfaz de usuario.
- *Look and Feel* cambia el fondo, el tipo de letra, bordes, colores y, en general, el aspecto de los componentes
- El tema que, por defecto, utiliza el IDE de *Netbeans* en sus aplicaciones es *Nimbus*
- Para que nuestra aplicación tenga el aspecto *Nimbus* añadiremos este código inmediatamente después del *main()*

```
public static void main(String[] args) {
    try {
        for (LookAndFeelInfo info : UIManager.getInstalledLookAndFeels()) {
            if ("Nimbus".equals(info.getName())) {
                UIManager.setLookAndFeel(info.getClassName());
                break;
            }
        }
    } catch (Exception e) {
        // If Nimbus is not available, you can set the GUI to another look and feel.
    }
    ControladorLogin cLogin = new ControladorLogin();
}
```

https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html