1 Exemplos JDBC

1.1 Sumario

2 Exemplos JDBC

2.1 Exemplos Java DB Derby

2.1.1 Exemplo 1: Conexión mediante JDBC a Java DB Derby amosando os resultados das consultas por pantalla

NOTA: A base de datos empregada para iste exemplo e máis información sobre Java DB Derby na seguinte ligazón: Traballando con Java DB Derby no IDE NetBeans

2.1.1.1 Código fonte Exemplo1

```
package exemplo;
import java.sql.*;
public class Exemplo1 {
    public static void main(String args[]){
        try {
            Class.forName("org.apache.derby.jdbc.ClientDriver");
            Connection conexion = DriverManager.getConnection(
                "jdbc:derby://localhost:1527/BDExemplo;create=true;user=app;password=abc1234");
            Statement consulta = conexion.createStatement();
            ResultSet taboa = consulta.executeQuery("SELECT * FROM app.exemplo");
            while(taboa.next())
            System.out.println(taboa.getInt(1)+"\t"+taboa.getString(2)+"\t"+taboa.getInt(3));
        }
        catch(ClassNotFoundException e){ System.out.println(e); }
        catch(SQLException e){ System.out.println(e); }
        catch(Exception e){ System.out.println(e); }
        }
}
```

2.1.1.2 Explicación código fonte Exemplo1

```
try {
01    Class.forName("org.apache.derby.jdbc.ClientDriver");
02    Connection conexion = DriverManager.getConnection(
03        "jdbc:derby://localhost:1527/BDExemplo;create=true;user=app;password=abc1234");
04    Statement consulta = conexion.createStatement();
05    ResultSet taboa = consulta.executeQuery("SELECT * FROM app.exemplo");
06    while(taboa.next())
07    System.out.println(taboa.getInt(1)+"\t"+taboa.getString(2)+"\t"+taboa.getInt(3));
    }
```

Liña 01: Cárgase o driver para poderse conectar mediante jdbc a unha base de datos Derby
Liñas 02-03: Establécese a conexión á base de datos Derby mediante DriverManager mediante a variable *conexion*(obxecto Connection). Pechado entre parénteses teremos a **URL** da conexión á base de datos:

- **jdbc:derby:**// -->Protocolo de conexión á base de datos Derby mediante jdbc
- **localhost:1527**/ -->*Máquina:porto* onde existe á base de datos Derby á acceder
- **BDExemplo** --> Nome da base de datos Derby
- **create=true** --> Crear a conexión
- **user=app** --> Usuario con permisos de acceso á base de datos, neste caso **app**
- **password=abc1234** --> Contrasinal do usuario con acceso á base de datos, neste caso **abc1234**

Liña 04: Créase a variable *consulta*(obxecto Statement) que permitirá executar sentencias SQL
Liña 05: Contén os resultados das consultas SQL na variable *taboa*(obxecto ResultSet), isto é, a variable *taboa* contén as filas obtidas ao executar a sentencia **SELECT * FROM app.exemplo**
Liñas 06-07: Namentres existan rexistros que ensinar na variable táboa imprímeos por pantalla de forma tabulada.

```
08    catch(ClassNotFoundException e){ System.out.println(e); }
09    catch(SQLException e){ System.out.println(e); }
10    catch(Exception e){ System.out.println(e); }
```

Liña 08: Excepción **ClassNotFoundException**, que terá lugar polo xeral cando o programa non atope o Driver.
Liña 09: Excepción **SQLException**, que terá lugar cando existan erros de SQL: erros ao insertar datos, erros de sintaxe nas consultas, ...
Liña 10: Excepción xenérica **Exception** para calquera tipo de excepción.

2.1.2 Exemplo 2: Conexión mediante JDBC a Java DB Derby amosando os resultados das consultas nun JTable

NOTAS:

1. **A base de datos empregada para iste exemplo e máis información sobre Java DB Derby na seguinte ligazón: Traballando con Java DB Derby no IDE NetBeans**
2. **Para entender iste exemplo é recomendable botarlle unha ollada ao Exemplo 1**

Partindo do código do **Exemplo1** imos modificalo para poder amosar os resultados das consultas nun compoñente Swing **JTable**, para isto crearemos unha clase que herde da clase **AbstractTableModel**. Con ista clase é posible implementar, dunha forma máis completa e eficiente, os métodos necesarios para crear un modelo de táboa. Os modelos de táboa son obxectos que implementan a interface TableModel, a través deles é posible persoalizar moito máis e mellor o comportamento dos compoñentes JTable, permitindo empregar ao máximo as súas potencialidades.
Para crear un **TableModel** como subclase de AbstractTableModel necesitaremos implementar únicamente os seguintes tres métodos:

1. **public int getRowCount()** --> Devolve o número de filas que terá a táboa
2. **public int getColumnCount()** --> Devolve o número de columnas que terá a táboa
3. **public Object getValueAt(int row, int column)** --> Devolve o dato indicado dunha celda da táboa mediante o par (fila, columna), considerando que (0,0) representa a primeira fila e a primeira columna respectivamente.

A maiores implementaremos un cuarto método: **public String getColumnName(int column)** , que devolve o nome de cada columna que posúe a táboa.

2.1.2.1 Código fonte Exemplo2

```
package exemplo;
import java.awt.*;
import java.sql.*;
import javax.swing.*;

public class Exemplo2 extends JFrame {

    public Exemplo2() {
        super( "Amosando resultados da consulta nun JTable" );
        try {
            AmosarTaboaResultados modeloTaboa = new AmosarTaboaResultados(
                   "org.apache.derby.jdbc.ClientDriver",
                   "jdbc:derby://localhost:1527/BDExemplo;create=true;user=app;password=abc1234",
                   "SELECT * FROM app.exemplo" );
            JTable taboaResultados = new JTable(modeloTaboa);
            getContentPane().setLayout(new BorderLayout());
            getContentPane().add( new JScrollPane(taboaResultados) , BorderLayout.CENTER );
            setSize( 450, 150 );
            setVisible( true );
        }
        catch (ClassNotFoundException e) {JOptionPane.showMessageDialog(null,
                e.getMessage(), "Non se atopou controlador", JOptionPane.ERROR_MESSAGE );}
        catch (SQLException e) {JOptionPane.showMessageDialog(null,
                e.getMessage(), "Erro na consulta SQL", JOptionPane.ERROR_MESSAGE);}
        catch (Exception e){JOptionPane.showMessageDialog(null,
                e.getMessage(), "", JOptionPane.ERROR_MESSAGE);}
    }


    public static void main( String args[] ) {
        JFrame.setDefaultLookAndFeelDecorated(true);
        new Exemplo2();
    }
}

package exemplo;

import java.sql.*;
import javax.swing.table.*;

public class AmosarTaboaResultados extends AbstractTableModel {
    private Connection conexion;
    private Statement consulta;
    private ResultSet taboa;
    private ResultSetMetaData metaDatos;
    private int numeroDeFilas;

    public AmosarTaboaResultados( String controlador, String url,
            String consulta ) throws SQLException, ClassNotFoundException {
        Class.forName( controlador );
        conexion = DriverManager.getConnection( url );
        this.consulta = conexion.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY );
        establecerConsulta( consulta );
    }

    public int getRowCount() throws IllegalStateException {
        return numeroDeFilas;
    }

    public int getColumnCount() {
        try {
            return metaDatos.getColumnCount();
        }
        catch (SQLException e) {e.printStackTrace();}
        return 0;
    }

    public String getColumnName( int columna ) {
        try {
```

```
            return metaDatos.getColumnName( columna + 1 );
        }
        catch (SQLException e) {e.printStackTrace();}
        return "";
    }

    public Object getValueAt( int fila, int columna ) {
        try {
            taboa.absolute( fila + 1 );
            return taboa.getObject( columna + 1 );
        }
        catch (SQLException e) {e.printStackTrace();}
        return "";
    }

    public void establecerConsulta( String consulta ) {
        try {
        taboa = this.consulta.executeQuery( consulta );
        metaDatos = taboa.getMetaData();
        taboa.last();
        numeroDeFilas = taboa.getRow();
        fireTableStructureChanged();
        }
        catch (SQLException e) {e.printStackTrace();}
    }

}
```

2.1.2.2 Explicación código fonte Exemplo2

2.1.2.2.1 Explicación Clase Exemplo2

```
00  public class Exemplo2 extends JFrame {

01    public Exemplo2() {
02        super( "Amosando resultados da consulta nun JTable" );
03        try {
04            AmosarTaboaResultados modeloTaboa = new AmosarTaboaResultados(
05                    "org.apache.derby.jdbc.ClientDriver",
06                    "jdbc:derby://localhost:1527/BDExemplo;create=true;user=app;password=abc1234",
07                    "SELECT * FROM app.exemplo" );
08            JTable taboaResultados = new JTable(modeloTaboa);
09            getContentPane().setLayout(new BorderLayout());
10            getContentPane().add( new JScrollPane(taboaResultados) , BorderLayout.CENTER );
11            setSize( 450, 150 );
12            setVisible( true );
13        }
14        catch (ClassNotFoundException e) {JOptionPane.showMessageDialog(null,
15                e.getMessage(), "Non se atopou controlador", JOptionPane.ERROR_MESSAGE );}
16        catch (SQLException e) {JOptionPane.showMessageDialog(null,
17                e.getMessage(), "Erro na consulta SQL", JOptionPane.ERROR_MESSAGE);}
18        catch (Exception e){JOptionPane.showMessageDialog(null,
19                e.getMessage(), "", JOptionPane.ERROR_MESSAGE);}
20    }


21    public static void main( String args[] ) {
22        JFrame.setDefaultLookAndFeelDecorated(true);
23        new Exemplo2();
24    }
25 }
```

Liñas 4,5,6,7: Chamada ao constructor da clase **AmosarTaboaResultados** mediante a creación da variable obxecto **modeloTaboa**. Na sinatura do constructor envíanse o driver para recoñecer a base de datos, a conexión á base de datos e a consulta requerida á base de datos, respectivamente.
Liña 8: Creación da variable obxecto **taboaResultados** co modelo **modeloTaboa** como parámetro na sinatura do construtor JTable
Liña 9: Establecer para a formulario JFrame da clase Exemplo2 o xestor de compoñentes (distribuidor de contidos) **BorderLayout**.
Liña 10: Engadir barras de desprazamento á táboa.
Liña 11: Tamaño predeterminado para o formulario.
Liña 12: Facer visible o formulario con todos os compoñentes pertencentes ao mesmo.
Liñas 14,15: Excepción **ClassNotFoundException**, que terá lugar polo xeral cando o programa non atope o Driver.

Liña 16,17: Excepción **SQLException**, que terá lugar cando existan erros de SQL: erros ao insertar datos, erros de sintaxe nas consultas, ...
Liña 18,19: Excepción xenérica **Exception** para calquera tipo de excepción.

2.1.2.2.2 Explicación Clase AmosarTaboaResultados

```
01    public class AmosarTaboaResultados extends AbstractTableModel {
02    private Connection conexion;
03    private Statement consulta;
04    private ResultSet taboa;
05    private ResultSetMetaData metaDatos;
06    private int numeroDeFilas;

07    public AmosarTaboaResultados( String controlador, String url,
08            String consulta ) throws SQLException, ClassNotFoundException {
09        Class.forName( controlador );
10        conexion = DriverManager.getConnection( url );
11       this.consulta = conexion.createStatement(
12                ResultSet.TYPE_SCROLL_INSENSITIVE,
13                ResultSet.CONCUR_READ_ONLY );
14        establecerConsulta( consulta );
15    }

16    public int getRowCount() throws IllegalStateException {
17        return numeroDeFilas;
18    }

19    public int getColumnCount() {
20        try {
21            return metaDatos.getColumnCount();
22        }
23        catch (SQLException e) {e.printStackTrace();}
24        return 0;
25    }

26    public String getColumnName( int columna ) {
27        try {
28            return metaDatos.getColumnName( columna + 1 );
29        }
30        catch (SQLException e) {e.printStackTrace();}
31        return "";
32    }

33    public Object getValueAt( int fila, int columna ) {
34        try {
35            taboa.absolute( fila + 1 );
36            return taboa.getObject( columna + 1 );
37        }
38        catch (SQLException e) {e.printStackTrace();}
38        return "";
39    }

40    public void establecerConsulta( String consulta ) {
41        try {
42        taboa = this.consulta.executeQuery( consulta );
43        metaDatos = taboa.getMetaData();
44        taboa.last();
45        numeroDeFilas = taboa.getRow();
46        fireTableStructureChanged();
47        }
48        catch (SQLException e) {e.printStackTrace();}
49    }

50 }
```

Liña 1: Declaración da clase **AmosarTaboaResultados** que herda da clase AbstractTableModel
Liñas 2,3,4,5,6: Declaración dos campos da clase para poder establecer a conexión coa base de datos e representar os datos nunha táboa creada a través dun modelo personalizado mediante a clase AbstractTableModel
Liñas 7,8,9,10,11,12,13,14,15: Declaración do construtor **AmosarTaboaResultados**. Na súa signatura espera 3 parámetros, respectivamente: o driver para recoñecer a base de datos, como conectar á base de datos, a consulta requerida á base de datos. Nas **liñas 12,13** o conxunto de resultados obtidos son non sensibles ao desprazamento e de só lectura respectivamente.

Liñas 16,17,18: Declaración do método **getRowCount** que permite a obtención do número de filas da táboa resultado da consulta SQL do **ResultSet**. O número de filas que debe ter un JTable debe coincidir co número de rexistros aos que fai referencia a consulta SQL do **ResulSet**. A interface **ResultSet** non dispón de ningún método que devolva esa información, polo que a forma de obtela será mediante o desprazamento do **ResulSet** á última fila e devolvendo a continuación o índice asociado a ésta. Por iso existen as **liñas 44, 45**.
Liñas 19,20,21,22,23,24,25: Declaración do método **getColumnCount** que permite a obtención do número de columnas da táboa resultado da consulta SQL do **ResulSet**. O número de columnas que debe ter un JTable debe coincidir co número de campos aos que fai referencia a consulta SQL do **ResulSet**. Esta información a interface **ResultSet** pode obtela a partir do obxecto **ResulSetMetaData** de nome **metaDatos**.
Liñas 26,27,28,29,30,31,32: Declaración do método **getColumnName** que permite a obtención dos nomes das columnas, o cal tamén pode obterse a partir do obxecto **ResulSetMetaData** de nome **metaDatos**.
Liñas 33,34,35,36,37,38,39: Declaración do método **getValueAt** que devolve á táboa o valor correspondente á celda indicada. Para isto emprega unha variable obxecto **ResultSet** de nome **taboa** desprazable, para obter todos os valores das celdas.
Liñas 44,45: Ver explicación **liñas 16,17,18**.
Liña 46: Actualizar a táboa.

2.1.3 Exemplo 3: Base de datos Derby embebida na aplicación. Conexión mediante JDBC a Java DB Derby Embebida amosando os resultados das consultas por pantalla

NOTA: A base de datos empregada para iste exemplo e máis información sobre Java DB Derby na seguinte ligazón: Traballando con Java DB Derby no IDE NetBeans

2.1.3.1 Código fonte Exemplo3

```
package exemplo;
import java.sql.*;
public class Exemplo1 {
    public static void main(String args[]){
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            Connection conexion = DriverManager.getConnection(
                "jdbc:derby:dist/BDExemplo;create=true;user=app;password=abc1234");
            Statement consulta = conexion.createStatement();
            ResultSet taboa = consulta.executeQuery("SELECT * FROM app.exemplo");
            while(taboa.next())
            System.out.println(taboa.getInt(1)+"\t"+taboa.getString(2)+"\t"+taboa.getInt(3));
        }
        catch(ClassNotFoundException e){ System.out.println(e); }
        catch(SQLException e){ System.out.println(e); }
        catch(Exception e){ System.out.println(e); }
        }
}
```

2.1.3.2 Explicación código fonte Exemplo3

```
try {
01    Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
02    Connection conexion = DriverManager.getConnection(
03        "jdbc:derby:dist/BDExemplo;create=true;user=app;password=abc1234");
04    Statement consulta = conexion.createStatement();
05    ResultSet taboa = consulta.executeQuery("SELECT * FROM app.exemplo");
06    while(taboa.next())
07    System.out.println(taboa.getInt(1)+"\t"+taboa.getString(2)+"\t"+taboa.getInt(3));
    }
```

Liña 01: Cárgase o driver para poderse conectar mediante jdbc a unha base de datos Derby
Liñas 02-03: Establécese a conexión á base de datos Derby mediante DriverManager mediante a variable *conexion*(obxecto Connection). Pechado entre parénteses teremos a **URL** da conexión á base de datos:
jdbc:derby:dist/BDExemplo;create=true;user=app;password=abc1234'"

- **jdbc:derby:**// -->Protocolo de conexión á base de datos Derby mediante jdbc
- **dist**/ -->Ruta onde existe á base de datos Derby á acceder
- **BDExemplo** --> Nome da base de datos Derby
- **create=true** --> Crear a conexión
- **user=app** --> Usuario con permisos de acceso á base de datos, neste caso **app**
- **password=abc1234** --> Contrasinal do usuario con acceso á base de datos, neste caso **abc1234**

Liña 04: Créase a variable *consulta*(obxecto Statement) que permitirá executar sentencias SQL
Liña 05: Contén os resultados das consultas SQL na variable *taboa*(obxecto ResultSet), isto é, a variable *taboa* contén as filas obtidas ao executar a sentencia **SELECT * FROM app.exemplo**
Liñas 06-07: Namentres existan rexistros que ensinar na variable táboa imprímeos por pantalla de forma tabulada.

```
08    catch(ClassNotFoundException e){ System.out.println(e); }
09    catch(SQLException e){ System.out.println(e); }
10    catch(Exception e){ System.out.println(e); }
```

Liña 08: Excepción **ClassNotFoundException**, que terá lugar polo xeral cando o programa non atope o Driver.
Liña 09: Excepción **SQLException**, que terá lugar cando existan erros de SQL: erros ao insertar datos, erros de sintaxe nas consultas, ...
Liña 10: Excepción xenérica **Exception** para calquera tipo de excepción.

2.1.3.3 Empaquetar a aplicación Exemplo3 para que sexa embebida

1. En **GNU/Linux**:
   IMPORTANTE: Para que funcione o exemplo en calquera equipo GNU/Linux onde executemos a aplicación é imprescindible empaquetar a aplicación mediante a estrutura seguinte:

```
instalacion/
|-- dist
|   |-- BDExemplo
|   |   |-- db.lck
|   |   |-- dbex.lck
|   |   |-- log
|   |   |-- seg0
|   |   |-- service.properties
|   |   `-- tmp
|   |-- EmbeberDerby.jar
|   `-- lib
|       |-- derby.jar
|       `-- derbytools.jar
`-- instalacion.sh
```

2. En **Windows** teremos case a mesma estrutura, soamente hai que cambiar o arquivo **instalacion.sh** polo arquivo **instalacion.bat**. Os 2 arquivos básicamente diferencianse en como cargar as variables de contorna, en **GNU/Linux** emprégase o comando **export** mentres que en **Windows** emprégase o comando **set**

NOTA: Para máis información sobre os arquivos instalacion.sh e instalacion.bat ver os apartados seguintes a iste

2.1.3.4 Descargar unha estrutura tipo da aplicación embebida Exemplo3

Podes atopar esta estrutura exemplo válida para **GNU/Linux** e **Windows** na seguinte ligazón: instalacion.zip

NOTA:Lembra que en Windows o arquivo instalacion.sh trocase por instalacion.bat *(ver os apartados anterior e seguinte para máis información ao respecto)*

2.1.3.5 Explicación da estrutura da aplicación embebida Exemplo3

1. Cartafol de nome **instalacion**--> O cartafol raiz da aplicación embebida.
2. Arquivo **instalacion/EmbeberDerby.jar**--> Paquete **jar** da aplicación.
3. Arquivo **instalacion/instalacion.sh** para **GNU/Linux** ou **instalacion/instalacion.bat** para **Windows**--> Script ou ficheiro por lotes que incorpora á variable **CLASSPATH** a ruta onde se atopan os paquetes Derby: *derby.jar* e *derbytools.jar*; o seu contido é o seguinte:
   1. Arquivo **instalacion.sh**:

```
#!/bin/sh
RUTA_ACTUAL=`pwd`
export CLASSPATH=$RUTA_ACTUAL/dist/lib/derby.jar:$RUTA_ACTUAL/dist/lib/derbytools.jar:${CLASSPATH}
java -jar dist/EmbeberDerby.jar
```

   2. Arquivo **instalacion.bat**:

```
@echo off
set CLASSPATH=.\dist\lib\derby.jar;.\dist\lib\derbytools.jar;%CLASSPATH%
java -jar dist/EmbeberDerby.jar
```

4. Cartafol **instalacion/dist** --> Cartafol que contén a BBDD Derby, neste caso, **BDExemplo**

5. Cartafol **instalacion/dist/lib**--> Cartafol que contén os paquetes Derby: **derby.jar** e **derbytools.jar**

2.1.3.6 Execución da aplicación embebida Exemplo3

2.1.3.6.1 Execución en liña de comandos, nun terminal, da aplicación embebida Exemplo3

1. Entrar no directorio raiz da aplicación embebida:

```
cd instalacion
```

2. 1. En **GNU/Linux** executar o script **instalacion.sh**:

```
sh instalacion.sh
```

   2. En **Windows** executar o ficheiro por lotes **instalacion.bat**:

```
instalacion.bat
```

2.1.3.6.2 Execución en contorna gráfica da aplicación embebida Exemplo3

1. Navegar mediante un explorador á ruta onde está situada a aplicación embebida
2. Dobre clic co rato en **instalacion.sh** para sistemas operativos **GNU/Linux** ou en **instalacion.bat** para sistemas operativos **Windows**

2.2 Exemplos MySQL

2.2.1 Exemplo 1: Conexión mediante JDBC a MySQL amosando os resultados das consultas por pantalla

NOTAS:

1. **Estes exemplos son os mesmos que os de Java DB Derby. Soamente cambia nos mesmos as liñas de código correspondentes á carga do driver e a conexión coa base de datos:**

```
Class.forName("com.mysql.jdbc.Driver");
Connection conexion = DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/BDExemplo?" + "user=app&password=abc1234");
```

2. **A base de datos empregada para iste exemplo e máis información sobre Java DB Derby na seguinte ligazón: Traballando con Java DB Derby no IDE NetBeans**

2.2.1.1 Código fonte Exemplo1

```
package exemplo;
import java.sql.*;
public class Exemplo1 {
    public static void main(String args[]){
        try {
             Class.forName("com.mysql.jdbc.Driver");
             Connection conexion = DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/BDExemplo?" + "user=app&password=abc1234");
            Statement consulta = conexion.createStatement();
            ResultSet taboa = consulta.executeQuery("SELECT * FROM BDExemplo.EXEMPLO");
            while(taboa.next())
            System.out.println(taboa.getInt(1)+"\t"+taboa.getString(2)+"\t"+taboa.getInt(3));
        }
        catch(ClassNotFoundException e){ System.out.println(e); }
        catch(SQLException e){ System.out.println(e); }
        catch(Exception e){ System.out.println(e); }
        }
}
```

2.2.1.2 Explicación código fonte Exemplo1

```
try {
01    Class.forName("com.mysql.jdbc.Driver");
02    Connection conexion = DriverManager.getConnection(
03           "jdbc:mysql://localhost:3306/BDExemplo?" + "user=app&password=abc1234");
04    Statement consulta = conexion.createStatement();
```

```
05    ResultSet taboa = consulta.executeQuery("SELECT * FROM BDExemplo.EXEMPLO");
06    while(taboa.next())
07    System.out.println(taboa.getInt(1)+"\t"+taboa.getString(2)+"\t"+taboa.getInt(3));
    }
```

Liña 01: Cárgase o driver para poderse conectar mediante jdbc a unha base de datos MySQL
Liñas 02-03: Establécese a conexión á base de datos MySQL mediante DriverManager mediante a variable *conexion*(obxecto Connection). Pechado entre parénteses teremos a **URL** da conexión á base de datos:

- **jdbc:mysql:**// -->Protocolo de conexión á base de datos MySQL mediante jdbc
- **localhost:3306**/ -->*Máquina:porto* onde existe á base de datos MySQL á acceder
- **BDExemplo** --> Nome da base de datos MySQL
- **create=true** --> Crear a conexión
- **user=app** --> Usuario con permisos de acceso á base de datos, neste caso **app**
- **password=abc1234** --> Contrasinal do usuario con acceso á base de datos, neste caso **abc1234**

Liña 04: Créase a variable *consulta*(obxecto Statement) que permitirá executar sentencias SQL
Liña 05: Contén os resultados das consultas SQL na variable *taboa*(obxecto ResultSet), isto é, a variable *taboa* contén as filas obtidas ao executar a sentencia **SELECT * FROM BDExemplo.EXEMPLO**
Liñas 06-07: Namentres existan rexistros que ensinar na variable táboa imprímeos por pantalla de forma tabulada.

```
08    catch(ClassNotFoundException e){ System.out.println(e); }
09    catch(SQLException e){ System.out.println(e); }
10    catch(Exception e){ System.out.println(e); }
```

Liña 08: Excepción **ClassNotFoundException**, que terá lugar polo xeral cando o programa non atope o Driver.
Liña 09: Excepción **SQLException**, que terá lugar cando existan erros de SQL: erros ao insertar datos, erros de sintaxe nas consultas, ...
Liña 10: Excepción xenérica **Exception** para calquera tipo de excepción.

2.2.2 Exemplo 2: Conexión mediante JDBC a MySQL amosando os resultados das consultas nun JTable

NOTAS:

1. **Estes exemplos son os mesmos que os de Java DB Derby. Soamente cambia nos mesmos as liñas de código correspondentes á carga do driver e a conexión coa base de datos:**

```
Class.forName("com.mysql.jdbc.Driver");
Connection conexion = DriverManager.getConnection(
                    "jdbc:mysql://localhost:3306/BDExemplo?" + "user=app&password=abc1234");
```

2. **A base de datos empregada para iste exemplo e máis información sobre Java DB Derby na seguinte ligazón: Traballando con Java DB Derby no IDE NetBeans**
3. **Para entender iste exemplo é recomendable botarlle unha ollada ao Exemplo 1**

Partindo do código do **Exemplo1** imos modificalo para poder amosar os resultados das consultas nun compoñente Swing **JTable**, para isto crearemos unha clase que herde da clase **AbstractTableModel**. Con ista clase é posible implementar, dunha forma máis completa e eficiente, os métodos necesarios para crear un modelo de táboa. Os modelos de táboa son obxectos que implementan a interface TableModel, a través deles é posible persoalizar moito máis e mellor o comportamento dos compoñentes JTable, permitindo empregar ao máximo as súas potencialidades.
Para crear un **TableModel** como subclase de AbstractTableModel necesitaremos implementar únicamente os seguintes tres métodos:

1. **public int getRowCount()** --> Devolve o número de filas que terá a táboa
2. **public int getColumnCount()** --> Devolve o número de columnas que terá a táboa
3. **public Object getValueAt(int row, int column)** --> Devolve o dato indicado dunha celda da táboa mediante o par (fila, columna), considerando que (0,0) representa a primeira fila e a primeira columna respectivamente.

A maiores implementaremos un cuarto método: **public String getColumnName(int column)** , que devolve o nome de cada columna que posúe a táboa.

2.2.2.1 Código fonte Exemplo2

```
package exemplo;
import java.awt.*;
import java.sql.*;
import javax.swing.*;

public class Exemplo2 extends JFrame {

    public Exemplo2() {
        super( "Amosando resultados da consulta nun JTable" );
        try {
            AmosarTaboaResultados modeloTaboa = new AmosarTaboaResultados(
                   "com.mysql.jdbc.Driver",
                   "jdbc:mysql://localhost:3306/BDExemplo?" + "user=app&password=abc1234",
                   "SELECT * FROM BDExemplo.EXEMPLO" );
            JTable taboaResultados = new JTable(modeloTaboa);
            getContentPane().setLayout(new BorderLayout());
            getContentPane().add( new JScrollPane(taboaResultados) , BorderLayout.CENTER );
            setSize( 450, 150 );
            setVisible( true );
        }
        catch (ClassNotFoundException e) {JOptionPane.showMessageDialog(null,
                e.getMessage(), "Non se atopou controlador", JOptionPane.ERROR_MESSAGE );}
        catch (SQLException e) {JOptionPane.showMessageDialog(null,
                e.getMessage(), "Erro na consulta SQL", JOptionPane.ERROR_MESSAGE);}
        catch (Exception e){JOptionPane.showMessageDialog(null,
                e.getMessage(), "", JOptionPane.ERROR_MESSAGE);}
    }


    public static void main( String args[] ) {
        JFrame.setDefaultLookAndFeelDecorated(true);
        new Exemplo2();
    }
}

package exemplo;

import java.sql.*;
import javax.swing.table.*;

public class AmosarTaboaResultados extends AbstractTableModel {
    private Connection conexion;
    private Statement consulta;
    private ResultSet taboa;
    private ResultSetMetaData metaDatos;
    private int numeroDeFilas;

    public AmosarTaboaResultados( String controlador, String url,
            String consulta ) throws SQLException, ClassNotFoundException {
        Class.forName( controlador );
        conexion = DriverManager.getConnection( url );
        this.consulta = conexion.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY );
        establecerConsulta( consulta );
    }

    public int getRowCount() throws IllegalStateException {
        return numeroDeFilas;
    }

    public int getColumnCount() {
        try {
            return metaDatos.getColumnCount();
        }
        catch (SQLException e) {e.printStackTrace();}
        return 0;
    }

    public String getColumnName( int columna ) {
        try {
```

```
            return metaDatos.getColumnName( columna + 1 );
        }
        catch (SQLException e) {e.printStackTrace();}
        return "";
    }

    public Object getValueAt( int fila, int columna ) {
        try {
            taboa.absolute( fila + 1 );
            return taboa.getObject( columna + 1 );
        }
        catch (SQLException e) {e.printStackTrace();}
        return "";
    }

    public void establecerConsulta( String consulta ) {
        try {
        taboa = this.consulta.executeQuery( consulta );
        metaDatos = taboa.getMetaData();
        taboa.last();
        numeroDeFilas = taboa.getRow();
        fireTableStructureChanged();
        }
        catch (SQLException e) {e.printStackTrace();}
    }

}
```

2.2.2.2 Explicación código fonte Exemplo2

2.2.2.2.1 Explicación Clase Exemplo2

```
00  public class Exemplo2 extends JFrame {

01    public Exemplo2() {
02        super( "Amosando resultados da consulta nun JTable" );
03        try {
04            AmosarTaboaResultados modeloTaboa = new AmosarTaboaResultados(
05                  "com.mysql.jdbc.Driver",
06                  "jdbc:mysql://localhost:3306/BDExemplo?" + "user=app&password=abc1234",
 07                  "SELECT * FROM BDExemplo.EXEMPLO" );
08            JTable taboaResultados = new JTable(modeloTaboa);
09            getContentPane().setLayout(new BorderLayout());
10            getContentPane().add( new JScrollPane(taboaResultados) , BorderLayout.CENTER );
11            setSize( 450, 150 );
12            setVisible( true );
13        }
14        catch (ClassNotFoundException e) {JOptionPane.showMessageDialog(null,
15                e.getMessage(), "Non se atopou controlador", JOptionPane.ERROR_MESSAGE );}
16        catch (SQLException e) {JOptionPane.showMessageDialog(null,
17                e.getMessage(), "Erro na consulta SQL", JOptionPane.ERROR_MESSAGE);}
18        catch (Exception e){JOptionPane.showMessageDialog(null,
19                e.getMessage(), "", JOptionPane.ERROR_MESSAGE);}
20    }


21    public static void main( String args[] ) {
22        JFrame.setDefaultLookAndFeelDecorated(true);
23        new Exemplo2();
24    }
25 }
```

Liñas 4,5,6,7: Chamada ao constructor da clase **AmosarTaboaResultados** mediante a creación da variable obxecto **modeloTaboa**. Na sinatura do constructor envíanse o driver para recoñecer a base de datos, a conexión á base de datos e a consulta requerida á base de datos, respectivamente.
Liña 8: Creación da variable obxecto **taboaResultados** co modelo **modeloTaboa** como parámetro na sinatura do construtor JTable
Liña 9: Establecer para a formulario JFrame da clase Exemplo2 o xestor de compoñentes (distribuidor de contidos) **BorderLayout**.
Liña 10: Engadir barras de desprazamento á táboa.
Liña 11: Tamaño predeterminado para o formulario.
Liña 12: Facer visible o formulario con todos os compoñentes pertencentes ao mesmo.
Liñas 14,15: Excepción **ClassNotFoundException**, que terá lugar polo xeral cando o programa non atope o Driver.

Liña 16,17: Excepción **SQLException**, que terá lugar cando existan erros de SQL: erros ao insertar datos, erros de sintaxe nas consultas, ...
Liña 18,19: Excepción xenérica **Exception** para calquera tipo de excepción.

2.2.2.2.2 Explicación Clase AmosarTaboaResultados

```
01    public class AmosarTaboaResultados extends AbstractTableModel {
02    private Connection conexion;
03    private Statement consulta;
04    private ResultSet taboa;
05    private ResultSetMetaData metaDatos;
06    private int numeroDeFilas;

07    public AmosarTaboaResultados( String controlador, String url,
08            String consulta ) throws SQLException, ClassNotFoundException {
09        Class.forName( controlador );
10        conexion = DriverManager.getConnection( url );
11       this.consulta = conexion.createStatement(
12                ResultSet.TYPE_SCROLL_INSENSITIVE,
13                ResultSet.CONCUR_READ_ONLY );
14        establecerConsulta( consulta );
15    }

16    public int getRowCount() throws IllegalStateException {
17        return numeroDeFilas;
18    }

19    public int getColumnCount() {
20        try {
21            return metaDatos.getColumnCount();
22        }
23        catch (SQLException e) {e.printStackTrace();}
24        return 0;
25    }

26    public String getColumnName( int columna ) {
27        try {
28            return metaDatos.getColumnName( columna + 1 );
29        }
30        catch (SQLException e) {e.printStackTrace();}
31        return "";
32    }

33    public Object getValueAt( int fila, int columna ) {
34        try {
35            taboa.absolute( fila + 1 );
36            return taboa.getObject( columna + 1 );
37        }
38        catch (SQLException e) {e.printStackTrace();}
38        return "";
39    }

40    public void establecerConsulta( String consulta ) {
41        try {
42        taboa = this.consulta.executeQuery( consulta );
43        metaDatos = taboa.getMetaData();
44        taboa.last();
45        numeroDeFilas = taboa.getRow();
46        fireTableStructureChanged();
47        }
48        catch (SQLException e) {e.printStackTrace();}
49    }

50 }
```

Liña 1: Declaración da clase **AmosarTaboaResultados** que herda da clase AbstractTableModel
Liñas 2,3,4,5,6: Declaración dos campos da clase para poder establecer a conexión coa base de datos e representar os datos nunha táboa creada a través dun modelo personalizado mediante a clase AbstractTableModel
Liñas 7,8,9,10,11,12,13,14,15: Declaración do construtor **AmosarTaboaResultados**. Na súa signatura espera 3 parámetros, respectivamente: o driver para recoñecer a base de datos, como conectar á base de datos, a consulta requerida á base de datos. Nas **liñas 12,13** o conxunto de resultados obtidos son non sensibles ao desprazamento e de só lectura respectivamente.

Liñas 16,17,18: Declaración do método **getRowCount** que permite a obtención do número de filas da táboa resultado da consulta SQL do **ResultSet**. O número de filas que debe ter un JTable debe coincidir co número de rexistros aos que fai referencia a consulta SQL do **ResulSet**. A interface **ResultSet** non dispón de ningún método que devolva esa información, polo que a forma de obtela será mediante o desprazamento do **ResulSet** á última fila e devolvendo a continuación o índice asociado a ésta. Por iso existen as **liñas 44, 45**.
Liñas 19,20,21,22,23,24,25: Declaración do método **getColumnCount** que permite a obtención do número de columnas da táboa resultado da consulta SQL do **ResulSet**. O número de columnas que debe ter un JTable debe coincidir co número de campos aos que fai referencia a consulta SQL do **ResulSet**. Esta información a interface **ResultSet** pode obtela a partir do obxecto **ResulSetMetaData** de nome **metaDatos**.
Liñas 26,27,28,29,30,31,32: Declaración do método **getColumnName** que permite a obtención dos nomes das columnas, o cal tamén pode obterse a partir do obxecto **ResulSetMetaData** de nome **metaDatos**.
Liñas 33,34,35,36,37,38,39: Declaración do método **getValueAt** que devolve á táboa o valor correspondente á celda indicada. Para isto emprega unha variable obxecto **ResultSet** de nome **taboa** desprazable, para obter todos os valores das celdas.
Liñas 44,45: Ver explicación **liñas 16,17,18**.
Liña 46: Actualizar a táboa.

--ricardofc [20/05/10]