Programming with JDBC in Derby

In this chapter, we will create Java programs which will work with the Derby database.

JDBC

JDBC is an API for the Java programming language that defines how a client may access a database. It provides methods for querying and updating data in a database. JDBC is oriented towards relational databases. From a technical point of view, the API is as a set of classes in the `java.sql` package. To use JDBC with a particular database, we need a JDBC driver for that database.

Client/server and embedded Derby applications

Derby can be used in Java applications in two basic ways: client/server and embedded. For client/server applications, we use `org.apache.derby.jdbc.ClientDriver` and for Derby embedded applications, we `org.apache.derby.jdbc.EmbeddedDriver`.

Maven dependencies

There are two Maven dependencies for Derby drivers: `derby` and `derbynet`. The `derby` dependency is used for embedded applications and `derbynet` for client/server applications.

```
<dependency>
    <groupId>org.apache.derby</groupId>
    <artifactId>derby</artifactId>
    <version>10.13.1.1</version>
</dependency>
```

This is the Maven dependency containing the `derby` driver.

```
<dependency>
    <groupId>org.apache.derby</groupId>
    <artifactId>derbyclient</artifactId>
    <version>10.13.1.1</version>
</dependency>
```

This is the Maven dependency containing the `derbyclient` driver.

Connection strings

The connection strings are different for the client/server and embedded applications.

```
jdbc:derby://localhost:1527/dbname
```

This is the connection URL for client/server applications.

```
jdbc:derby:dbname
```

This is the connection URL for embedded applications.

Creating the CARS table

In our examples, we use embedded Derby database. In the first example, we will create a `CARS` table and insert eight rows into it.

```
$ $DERBY_HOME/bin/ij
ij version 10.11
ij> CONNECT 'jdbc:derby:testdb';
ij> DROP TABLE USER12.CARS;
0 rows inserted/updated/deleted
```

We should drop the `CARS` table from the database if it is already created before running the example.

CreateCars.java

```
package com.zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CreateCars {

    public static void main(String[] args) {
```

```
        Connection con = null;
        Statement st = null;

        String url = "jdbc:derby:testdb;user=USER12";

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            con = DriverManager.getConnection(url);
            st = con.createStatement();
            st.executeUpdate("CREATE TABLE CARS(ID INT PRIMARY KEY,"
                    + "NAME VARCHAR(30), PRICE INT)");
            st.executeUpdate("INSERT INTO CARS VALUES(1, 'Audi', 52642)");
            st.executeUpdate("INSERT INTO CARS VALUES(2, 'Mercedes', 57127)");
            st.executeUpdate("INSERT INTO CARS VALUES(3, 'Skoda', 9000)");
            st.executeUpdate("INSERT INTO CARS VALUES(4, 'Volvo', 29000)");
            st.executeUpdate("INSERT INTO CARS VALUES(5, 'Bentley', 350000)");
            st.executeUpdate("INSERT INTO CARS VALUES(6, 'Citroen', 21000)");
            st.executeUpdate("INSERT INTO CARS VALUES(7, 'Hummer', 41400)");
            st.executeUpdate("INSERT INTO CARS VALUES(8, 'Volkswagen', 21600)");
            DriverManager.getConnection("jdbc:derby:;shutdown=true");

        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(CreateCars.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {

                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {

                if (st != null) {
                    st.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(CreateCars.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

The example connects to the Derby in embedded mode. It creates a CARS table and adds 8 rows into it. Finally, it shuts down Derby.

```
String url = "jdbc:derby:testdb;user=USER12";
```

This is the URL to connect to the testdb database, in the embedded mode and with USER12 schema.

```
System.setProperty("derby.system.home", "/home/janbodnar/.derby");
```

We set the system property for the Derby system directory.

```
con = DriverManager.getConnection(url);
```

A connection to the Derby database is created. When the connection is created, the Derby database is booted.

```
st.executeUpdate("CREATE TABLE CARS(ID INT PRIMARY KEY,"
        + "NAME VARCHAR(30), PRICE INT)");
st.executeUpdate("INSERT INTO CARS VALUES(1, 'Audi', 52642)");
...
```

We execute the SQL statements which create the database and fill it with some data. For INSERT, UPDATE, and DELETE statements and DDL statements like CREATE TABLE we use the executeUpdate() method.

```
DriverManager.getConnection("jdbc:derby:;shutdown=true");
```

The Derby database engine is shut down.

```
} catch (SQLException ex) {

    Logger lgr = Logger.getLogger(CreateCars.class.getName());
```

We catch the SQLException. The Logger class is used to log the error message.

```
if (((ex.getErrorCode() == 50000)
        && ("XJ015".equals(ex.getSQLState())))) {

    lgr.log(Level.INFO, "Derby shut down normally", ex);

}
```

When the Derby engine is shut down, an SQLException is thrown. We catch this exception and log an information message.

```
} finally {

    try {

        if (st != null) {
```

```
            st.close();
        }
        if (con != null) {
            con.close();
        }
```

In the finally clause, we release the resources.

```
Mar 22, 2017 12:22:15 PM com.zetcode.CreateCars main
INFO: Derby shut down normally
java.sql.SQLException: Derby system shutdown.
...
```

We compile and run the example. The shut down of Derby will end in an `SQLException`. This is a feature o the Derby database.

Retrieving data

Next we will show, how to retrieve data from a database table. We get all data from the `CARS` table.

SelectAllCars.java

```
package com.zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class SelectAllCars {

    public static void main(String[] args) {

        Connection con = null;
        Statement st = null;
        ResultSet rs = null;

        String url = "jdbc:derby:testdb";

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            con = DriverManager.getConnection(url);
            st = con.createStatement();
            rs = st.executeQuery("SELECT * FROM USER12.CARS");

            while (rs.next()) {
                System.out.print(rs.getInt(1));
                System.out.print(" ");
```

```
                System.out.print(rs.getString(2));
                System.out.print(" ");
                System.out.println(rs.getString(3));
            }

            DriverManager.getConnection("jdbc:derby:;shutdown=true");

        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(SelectAllCars.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {

                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {
                if (rs != null) {
                    rs.close();
                }
                if (st != null) {
                    st.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(SelectAllCars.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

We get all the cars from the `CARS` table and print them to the console.

```
st = con.createStatement();
rs = st.executeQuery("SELECT * FROM USER12.CARS");
```

We execute a query that selects all columns from the `CARS` table. We use the `executeQuery()` method. The method executes the given SQL statement, which returns a single ResultSet object. The ResultSet is the data table returned by the SQL query. Also note that since we have not specified the user name in the URL, we have to explicitly mention the schema name in the SQL statement.

```
while (rs.next()) {
    System.out.print(rs.getInt(1));
```

```
            System.out.print(" ");
            System.out.print(rs.getString(2));
            System.out.print(" ");
            System.out.println(rs.getString(3));
        }
```

The `next()` method advances the cursor to the next record of the result set. It returns false when there are no more rows in the result set. The `getInt()` and `getString()` methods retrieve the value of the designated column in the current row of this `ResultSet` object; an `int` and `String` in the Java programming language.

```
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
7 Hummer 41400
8 Volkswagen 21600
Mar 22, 2017 12:28:36 PM com.zetcode.SelectAllCars main
INFO: Derby shut down normally
java.sql.SQLException: Derby system shutdown.
...
```

We compile and run the example. We have a list of all cars from the `CARS` table of the `testdb` database.

Properties

It is a common practice to put the configuration data outside the program in a separate file. We can change the user, a password, or the connection string without needing to recompile the program. It is especially useful in a dynamic environment, where is a need for a lot of testing, debugging, securing data etc.

In Java, the `Properties` is a class used often for storing basic configuration data. The class is used for easy reading and saving of key/value properties.

db.properties

```
db.url=jdbc:derby:testdb;user=USER12
db.user=USER12
db.passwd=34klq*
db.syshome=/home/janbodnar/.derby
```

We have a `db.roperties` file, in which we have four key/value pairs. These are dynamically loaded during the execution of the program. The file is located in the `src/main/resources` directory.

PropertiesExample.java

```
package com.zetcode;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

public class PropertiesExample {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;
        ResultSet rs = null;

        Properties props = new Properties();
        FileInputStream in = null;

        try {

            in = new FileInputStream("src/main/resources/db.properties");
            props.load(in);

        } catch (FileNotFoundException ex) {

            Logger lgr = Logger.getLogger(PropertiesExample.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } catch (IOException ex) {

            Logger lgr = Logger.getLogger(PropertiesExample.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } finally {

            try {
                if (in != null) {
                    in.close();
                }
            } catch (IOException ex) {
                Logger lgr = Logger.getLogger(PropertiesExample.class.getName());
                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }
        }

        String url = props.getProperty("db.url");
        String user = props.getProperty("db.user");
        String passwd = props.getProperty("db.passwd");
```

```
        try {

            System.setProperty("derby.system.home",
                    props.getProperty("db.syshome"));

            con = DriverManager.getConnection(url, user, passwd);
            pst = con.prepareStatement("SELECT * FROM CARS");
            rs = pst.executeQuery();

            while (rs.next()) {
                System.out.print(rs.getInt(1));
                System.out.print(": ");
                System.out.println(rs.getString(2));
            }

            DriverManager.getConnection("jdbc:derby:;shutdown=true");

        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(PropertiesExample.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {

                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {
                if (rs != null) {
                    rs.close();
                }
                if (pst != null) {
                    pst.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {

                Logger lgr = Logger.getLogger(PropertiesExample.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

We connect to the `testdb` and select all cars from the `CARS` table. The configuration data for the example is read from the `db.properties` file.

```
Properties props = new Properties();
FileInputStream in = null;

try {

    in = new FileInputStream("src/main/resources/db.properties");
    props.load(in);
```

The `Properties` class is created. The data is loaded from the file called `db.properties`, where we have ou configuration data.

```
String url = props.getProperty("db.url");
String user = props.getProperty("db.user");
String passwd = props.getProperty("db.passwd");
```

The values are retrieved with the `getProperty()` method.

```
con = DriverManager.getConnection(url, user, passwd);
```

Note that in the default Derby configuration, the password is ignored.

Prepared statements

Now we will concern ourselves with prepared statements. When we write prepared statements, we use placeholders instead of directly writing the values into the statements. Prepared statements increase securi and performance.

In Java a `PreparedStatement` is an object which represents a precompiled SQL statement.

Prepared.java

```
package com.zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Prepared {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;
        ResultSet rs = null;

        String url = "jdbc:derby:testdb;user=USER12";
```

```
        int price = 58000;
        int id = 2;

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            con = DriverManager.getConnection(url);

            pst = con.prepareStatement("UPDATE CARS SET PRICE = ? WHERE ID = ?");
            pst.setInt(1, price);
            pst.setInt(2, id);
            pst.executeUpdate();

            DriverManager.getConnection("jdbc:derby:;shutdown=true");

        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(Prepared.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {

                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {
                if (rs != null) {
                    rs.close();
                }
                if (pst != null) {
                    pst.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(Prepared.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

We change the price for a car with id equal to 2.

```
int price = 58000;
int id = 2;
```

These are the values that are going to be set to the prepared statement. These values could come from a use and everything coming from users should be considered potentionally dangerous.

```
pst = con.prepareStatement("UPDATE CARS SET PRICE = ? WHERE ID = ?");
```

Here we create a prepared statement. When we write prepared statements, we use placeholders instead of directly writing the values into the statements. Prepared statements are faster and guard against SQL injection attacks. The `?` is a placeholder, which is going to be filled later.

```
pst.setInt(1, price);
pst.setInt(2, id);
```

Values are bound to the placeholders.

```
pst.executeUpdate();
```

The prepared statement is executed. We use the `executeUpdate()` method of the statement object when w do not expect any data to be returned. This is when we create databases or execute `INSERT`, `UPDATE`, and `DELETE` statements.

```
ij> SELECT * FROM CARS WHERE ID=2;
ID         |NAME                          |PRICE
------------------------------------------------------
2          |Mercedes                      |58000

1 row selected
```

After running the example, we check the outcome with the `ij` tool.

Column headers

Next we will show, how to print column headers with the data from the database table. We refer to column names as MetaData. MetaData is data about the core data in the database.

ColumnHeaders.java

```
package com.zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.util.Formatter;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class ColumnHeaders {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;
        ResultSet rs = null;

        String url = "jdbc:derby:testdb;user=USER12";

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            con = DriverManager.getConnection(url);
            String query = "SELECT NAME, TITLE From AUTHORS, "
                    + "Books WHERE AUTHORS.ID=BOOKS.AUTHOR_ID";
            pst = con.prepareStatement(query);

            rs = pst.executeQuery();

            ResultSetMetaData meta = rs.getMetaData();

            String colname1 = meta.getColumnName(1);
            String colname2 = meta.getColumnName(2);

            Formatter fmt1 = new Formatter();
            fmt1.format("%-21s%s", colname1, colname2);
            System.out.println(fmt1);

            while (rs.next()) {
                Formatter fmt2 = new Formatter();
                fmt2.format("%-21s", rs.getString(1));
                System.out.print(fmt2);
                System.out.println(rs.getString(2));
            }

            DriverManager.getConnection("jdbc:derby:;shutdown=true");

        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(ColumnHeaders.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {

                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {
                if (rs != null) {
```

```
                rs.close();
            }
            if (pst != null) {
                pst.close();
            }
            if (con != null) {
                con.close();
            }

        } catch (SQLException ex) {
            Logger lgr = Logger.getLogger(ColumnHeaders.class.getName());
            lgr.log(Level.WARNING, ex.getMessage(), ex);
        }
    }
}
```

In this program, we select authors from the AUTHORS table and their books from the BOOKS table. We print the names of the columns returned in the result set. We format the output. The SQL file to create the tables is located in the first chapter of this tutorial.

```
String query = "SELECT NAME, TITLE From AUTHORS, "
        + "Books WHERE AUTHORS.ID=BOOKS.AUTHOR_ID";
```

This is the SQL statement which joins authors with their books.

```
ResultSetMetaData meta = rs.getMetaData();
```

To get the column names we need to get the `ResultSetMetaData`. It is an object that can be used to get information about the types and properties of the columns in a `ResultSet` object. The `ResultSetMetaDat` is obtained from the `ResultSet` with the `getMetaData()` method.

```
String colname1 = meta.getColumnName(1);
String colname2 = meta.getColumnName(2);
```

From the obtained metadata, we get the column names using the `getColumnName()` method.

```
Formatter fmt1 = new Formatter();
fmt1.format("%-21s%s", colname1, colname2);
System.out.println(fmt1);
```

We print the column names to the console. The `Formatter` object formats the data.

```
while (rs.next()) {
    Formatter fmt2 = new Formatter();
    fmt2.format("%-21s", rs.getString(1));
    System.out.print(fmt2);
    System.out.println(rs.getString(2));
}
```

We print the data to the console. We again use the `Formatter` object to format the data. The first column is 21 characters wide and is aligned to the left.

```
NAME                 TITLE
Jack London          Call of the Wild
Jack London          Martin Eden
Honore de Balzac     Old Goriot
Honore de Balzac     Cousin Bette
Lion Feuchtwanger    Jew Suess
Emile Zola           Nana
Emile Zola           The Belly of Paris
Truman Capote        In Cold blood
Truman Capote        Breakfast at Tiffany
Mar 22, 2017 12:52:56 PM com.zetcode.ColumnHeaders main
INFO: Derby shut down normally
java.sql.SQLException: Derby system shutdown.
...
```

This is the output of the example.

Writing images

Some people prefer to put their images into the database, some prefer to keep them on the file system for their applications. Technical difficulties arise when we work with lots of images. Images are binary data. Derby has a special data type to store binary data called `BLOB` (Binary Large Object).

We create a new table called `IMAGES` for this and the following example.

```
ij> CREATE TABLE IMAGES(ID INT PRIMARY KEY, DATA BLOB);
0 rows inserted/updated/deleted
```

The `DATA` column has the `BLOB` type. There we will insert the encoded binary data.

WriteImage.java

```
package com.zetcode;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class WriteImage {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;

        String url = "jdbc:derby:testdb;user=USER12";

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            con = DriverManager.getConnection(url);

            File imgFile = new File("woman.jpg");

            try (FileInputStream fin = new FileInputStream(imgFile)) {
                con = DriverManager.getConnection(url);

                pst = con.prepareStatement("INSERT INTO IMAGES(ID, DATA) VALUES(1, ?)");
                pst.setBinaryStream(1, fin, (int) imgFile.length());
                pst.executeUpdate();
            }

            DriverManager.getConnection("jdbc:derby:;shutdown=true");

        } catch (FileNotFoundException ex) {

            Logger lgr = Logger.getLogger(WriteImage.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(WriteImage.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {

                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } catch (IOException ex) {
            Logger.getLogger(WriteImage.class.getName()).log(Level.SEVERE, null, ex);
        } finally {

            try {

                if (pst != null) {
                    pst.close();
                }
                if (con != null) {
```

```
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(WriteImage.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

In this example, we read a JPG image from the current working directory and insert in into the `IMAGES` table.

```
File imgFile = new File("woman.jpg");

try (FileInputStream fin = new FileInputStream(imgFile)) {
```

We create a `File` object for the image file. To read bytes from this file, we create a `FileInputStream` objec

```
pst = con.prepareStatement("INSERT INTO IMAGES(ID, DATA) VALUES(1, ?)");
```

This SQL statement inserts the image into the `Images` table.

```
pst.setBinaryStream(1, fin, (int) img.length());
```

The binary stream is set to the prepared statement. The parameters of the `setBinaryStream()` method ar the parameter index to bind, the input stream and the number of bytes in the stream.

```
pst.executeUpdate();
```

We execute the statement with the `executeUpdate()` method.

Reading images

In the previous example, we have inserted an image into the database table. Now we are going to read the image back from the table.

ReadImage.java

```
package com.zetcode;

import java.io.FileOutputStream;
import java.io.IOException;
import java.sql.Blob;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class ReadImage {

    public static void main(String[] args) {

        Connection con = null;
        PreparedStatement pst = null;
        ResultSet rs = null;

        String url = "jdbc:derby:testdb;user=USER12";

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            System.out.println(System.getProperty("user.dir"));

            con = DriverManager.getConnection(url);

            String query = "SELECT DATA FROM IMAGES WHERE ID = 1";
            pst = con.prepareStatement(query);

            rs = pst.executeQuery();
            rs.next();

            String fileName = "src/main/resources/woman.jpg";

            try (FileOutputStream fos = new FileOutputStream(fileName)) {

                Blob blob = rs.getBlob("DATA");
                int len = (int) blob.length();

                byte[] buf = blob.getBytes(1, len);

                fos.write(buf, 0, len);
            }

            DriverManager.getConnection("jdbc:derby:;shutdown=true");

        } catch (IOException ex) {

            Logger lgr = Logger.getLogger(ReadImage.class.getName());
            lgr.log(Level.SEVERE, ex.getMessage(), ex);

        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(ReadImage.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {
```

```
                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {
                if (rs != null) {
                    rs.close();
                }
                if (pst != null) {
                    pst.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(ReadImage.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

We read one image from the `IMAGES` table.

```
String query = "SELECT DATA FROM IMAGES WHERE ID = 1";
```

One record is selected.

```
try (FileOutputStream fos = new FileOutputStream(fileName)) {
```

The `FileOutputStream` object is created to write to a file. It is meant for writing streams of raw bytes such as image data.

```
Blob blob = result.getBlob("DATA");
```

We get the image data from the `DATA` column by calling the `getBlob()` method.

```
int len = (int) blob.length();
```

We find out the length of the blob data. In other words, we get the number of bytes.

```
byte[] buf = blob.getBytes(1, len);
```

The `getBytes()` method retrieves all bytes of the `BLOB` object, as an array of bytes.

```
fos.write(buf, 0, len);
```

The bytes are written to the output stream. The image is created on the filesystem.

Transaction support

A transaction is an atomic unit of database operations against the data in one or more databases. The effec of all the SQL statements in a transaction can be either all committed to the database or all rolled back.

When a connection is created, it is in autocommit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed. This is true for all JDBC drivers, including the Derby's one. To start a new transaction, we turn the autocommit off.

In direct SQL, a transaction is started with `BEGIN TRANSACTION` statement and ended with `END TRANSACTION/COMMIT` statement. In Derby these statements are `BEGIN` and `COMMIT`. However, when worki with drivers these statements are omitted. They are handled by the driver. Exact details are specific to the driver. For example `psycopg2` Python driver starts a transaction after the first SQL statement. If we want the autocommit mode, we must be set the autocommit property to True. In constrast, JDBC driver is by default in the autocommit mode. And to start a new transaction, the autocommit must be turned off.

Transaction.java

```
package com.zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Transaction {

    public static void main(String[] args) {

        Connection con = null;
        Statement st = null;

        String url = "jdbc:derby:testdb;user=USER12";

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            con = DriverManager.getConnection(url);

            st = con.createStatement();

            con.setAutoCommit(false);

            st.executeUpdate("UPDATE AUTHORS SET NAME = 'Leo Tolstoy' "
                    + "WHERE Id = 1");
            st.executeUpdate("UPDATE BOOKS SET TITLE = 'War and Peace' "
                    + "WHERE Id = 1");
            st.executeUpdate("UPDATE BOOKS SET TITL = 'Anna Karenina' "
                    + "WHERE Id = 2");
```

```
            con.commit();

            DriverManager.getConnection("jdbc:derby:;shutdown=true");


        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(Transaction.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {

                if (con != null) {
                    try {
                        con.rollback();
                    } catch (SQLException ex1) {
                        lgr.log(Level.WARNING, ex1.getMessage(), ex1);
                    }
                }

                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {
                if (st != null) {
                    st.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(Transaction.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

In this program, we want to change the name of the author in the first row of the `AUTHORS` table. We must also change the books associated with this author. If we change the author and do not change the author's books, the data is corrupted.

```
con.setAutoCommit(false);
```

To work with transactions, we must set the autocommit to false. By default, a database connection is in autocommit mode. In this mode each statement is committed to the database, as soon as it is executed. A

statement cannot be undone. When the autocommit is turned off, we commit the changes by calling the `commit()` or roll it back by calling the `rollback()` method.

```
st.executeUpdate("UPDATE BOOKS SET TITL = 'Anna Karenina' "
                    + "WHERE Id = 2");
```

The third SQL statement has an error. There is no TITL column in the `BOOKS` table.

```
con.commit();
```

If there is no exception, the transaction is committed. If the autocommit is turned off, we must explicitly ca the `commit()` method.

```
if (con != null) {

    try {
        con.rollback();
    } catch (SQLException ex1) {
        lgr.log(Level.WARNING, ex1.getMessage(), ex1);
    }
}
```

In case of an exception other than the Derby system shutdown, the transaction is rolled back. No changes are committed to the database.

```
Mar 22, 2017 2:00:40 PM com.zetcode.Transaction main
SEVERE: 'TITL' is not a column in table or VTI 'USER12.BOOKS'.
java.sql.SQLSyntaxErrorException: 'TITL' is not a column in table or VTI 'USER12.BOOKS'.
```

The execution fails with the "'TITL' is not a column in table" message. An exception was thrown. The transaction was rolled back and no changes took place.

```
ij> CONNECT 'jdbc:derby:testdb';
ij> SET CURRENT SCHEMA = USER12;
ij> SELECT NAME, TITLE FROM AUTHORS, BOOKS WHERE AUTHORS.ID = BOOKS.AUTHOR_ID;
NAME                     |TITLE
-------------------------------------------------------------
Jack London              |Call of the Wild
Jack London              |Martin Eden
Honore de Balzac         |Old Goriot
Honore de Balzac         |Cousin Bette
Lion Feuchtwanger        |Jew Suess
Emile Zola               |Nana
Emile Zola               |The Belly of Paris
Truman Capote            |In Cold blood
Truman Capote            |Breakfast at Tiffany
```

```
9 rows selected
```

The data is not corrupted.

However, without a transaction, the data is not safe.

NonTransaction.java

```
package com.zetcode;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class NonTransaction {

    public static void main(String[] args) {

        Connection con = null;
        Statement st = null;

        String url = "jdbc:derby:testdb;user=USER12";

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            con = DriverManager.getConnection(url);

            st = con.createStatement();

            st.executeUpdate("UPDATE AUTHORS SET NAME = 'Leo Tolstoy' "
                    + "WHERE Id = 1");
            st.executeUpdate("UPDATE BOOKS SET TITLE = 'War and Peace' "
                    + "WHERE Id = 1");
            st.executeUpdate("UPDATE BOOKS SET TITL = 'Anna Karenina' "
                    + "WHERE Id = 2");

            DriverManager.getConnection("jdbc:derby:;shutdown=true");


        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(NonTransaction.class.getName());

            if (((ex.getErrorCode() == 50000)
                    && ("XJ015".equals(ex.getSQLState())))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {
```

```
                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {
                if (st != null) {
                    st.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(NonTransaction.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

We have the same example; this time, without the transaction support.

```
Mar 22, 2017 2:08:40 PM com.zetcode.NonTransaction main
SEVERE: 'TITL' is not a column in table or VTI 'USER12.BOOKS'.
java.sql.SQLSyntaxErrorException: 'TITL' is not a column in table or VTI 'USER12.BOOKS'.
...


ij> CONNECT 'jdbc:derby:testdb';
ij> SET CURRENT SCHEMA = USER12;
ij> SELECT NAME, TITLE FROM AUTHORS, BOOKS WHERE AUTHORS.ID = BOOKS.AUTHOR_ID;
NAME                    |TITLE
------------------------------------------------------------------
Leo Tolstoy             |War and Peace
Leo Tolstoy             |Martin Eden
Honore de Balzac        |Old Goriot
Honore de Balzac        |Cousin Bette
Lion Feuchtwanger       |Jew Suess
Emile Zola              |Nana
Emile Zola              |The Belly of Paris
Truman Capote           |In Cold blood
Truman Capote           |Breakfast at Tiffany

9 rows selected
```

An exception is thrown again. Leo Tolstoy did not write Martin Eden: the data is corrupted.

Batch updates

When we need to update data with multiple statements, we can use batch updates. Batch updates are available for `INSERT`, `UPDATE`, `DELETE` statements as well as for `CREATE TABLE` and `DROP TABLE` statements

BatchUpdates.java

```
package com.zetcode;

import java.sql.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class BatchUpdates {

    public static void main(String[] args) {

        Connection con = null;
        Statement st = null;
        ResultSet rs = null;

        String url = "jdbc:derby:testdb;user=USER12";

        try {

            System.setProperty("derby.system.home", "/home/janbodnar/.derby");

            con = DriverManager.getConnection(url);

            con.setAutoCommit(false);
            st = con.createStatement();

            st.addBatch("DELETE FROM CARS");
            st.addBatch("INSERT INTO CARS VALUES(1, 'Audi', 52642)");
            st.addBatch("INSERT INTO CARS VALUES(2, 'Mercedes', 57127)");
            st.addBatch("INSERT INTO CARS VALUES(3, 'Skoda', 9000)");
            st.addBatch("INSERT INTO CARS VALUES(4, 'Volvo', 29000)");
            st.addBatch("INSERT INTO CARS VALUES(5, 'Bentley', 350000)");
            st.addBatch("INSERT INTO CARS VALUES(6, 'Citroen', 21000)");
            st.addBatch("INSERT INTO CARS VALUES(7, 'Hummer', 41400)");
            st.addBatch("INSERT INTO CARS VALUES(8, 'Volkswagen', 21600)");
            st.addBatch("INSERT INTO CARS VALUES(9, 'Jaguar', 95000)");

            int counts[] = st.executeBatch();

            con.commit();

            System.out.println("Committed " + counts.length + " updates");

            DriverManager.getConnection("jdbc:derby:;shutdown=true");

        } catch (SQLException ex) {

            Logger lgr = Logger.getLogger(BatchUpdates.class.getName());

            if (((ex.getErrorCode() == 50000)
```

```
                    && ("XJ015".equals(ex.getSQLState()))) {

                lgr.log(Level.INFO, "Derby shut down normally", ex);

            } else {

                if (con != null) {
                    try {
                        con.rollback();
                    } catch (SQLException ex1) {
                        lgr.log(Level.WARNING, ex1.getMessage(), ex1);
                    }
                }

                lgr.log(Level.SEVERE, ex.getMessage(), ex);
            }

        } finally {

            try {
                if (rs != null) {
                    rs.close();
                }
                if (st != null) {
                    st.close();
                }
                if (con != null) {
                    con.close();
                }

            } catch (SQLException ex) {
                Logger lgr = Logger.getLogger(BatchUpdates.class.getName());
                lgr.log(Level.WARNING, ex.getMessage(), ex);
            }
        }
    }
}
```

This is an example program for a batch update. We delete all rows from the `CARS` table and insert nine row into it.

```
con.setAutoCommit(false);
```

Autocommit should always be turned off when doing batch updates.

```
st.addBatch("DELETE FROM CARS");
st.addBatch("INSERT INTO CARS VALUES(1, 'Audi', 52642)");
st.addBatch("INSERT INTO CARS VALUES(2, 'Mercedes', 57127)");
st.addBatch("INSERT INTO CARS VALUES(3, 'Skoda', 9000)");
...
```

We use teh `addBatch()` method to add a new command to the statement.

```
int counts[] = st.executeBatch();
```

After adding all commands, we call the `executeBatch()` to perform a batch update. The method returns a array of committed changes.

```
con.commit();
```

Batch updates are committed in a transaction.

```
ij> SELECT * FROM CARS;
ID          |NAME                           |PRICE
-----------------------------------------------------
1           |Audi                           |52642
2           |Mercedes                       |57127
3           |Skoda                          |9000
4           |Volvo                          |29000
5           |Bentley                        |350000
6           |Citroen                        |21000
7           |Hummer                         |41400
8           |Volkswagen                     |21600
9           |Jaguar                         |95000
```

We have successfully recreated the `CARS` table.

In this chapter, we did some JDBC programming with Java and Derby.

Home Contents Top of Page Previous Nex