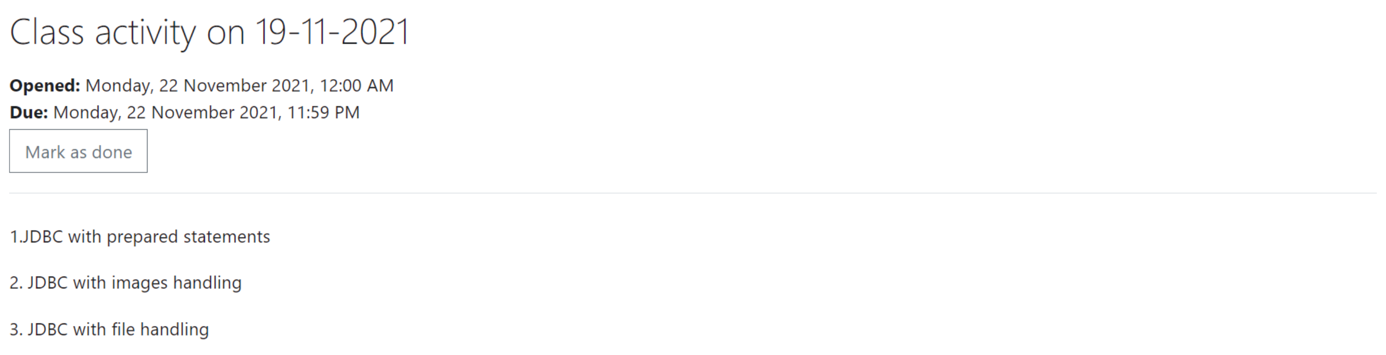
CSE1007 Java Programming Theory A2+TAA2 on November 19, 2021

Sharadindu Adhikari | 19BCE2105



Class activity on 19-11-2021

Opened: Monday, 22 November 2021, 12:00 AM
Due: Monday, 22 November 2021, 11:59 PM

Mark as done

1.JDBC with prepared statements

2. JDBC with images handling

3. JDBC with file handling

1. JDBC PreparedStatement

A Java JDBC PreparedStatement is a special kind of Java JDBC Statement object with some useful additional features. We need a Statement in order to execute either a query or an update. We can use a Java JDBC PreparedStatement instead of a Statement and benefit from the features of the PreparedStatement.

The Java JDBC PreparedStatement primary features are:

- Easy to insert parameters into the SQL statement.
- Easy to reuse the PreparedStatement with new parameter values.
- May increase performance of executed statements.
- Enables easier batch updates.

I will show how to insert parameters into SQL statements in this text, and also how to reuse a PreparedStatement. The batch updates is explained in a separate text.

Here is a quick example, to give a sense of how it looks in code:

```
String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");
preparedStatement.setString(2, "Larson");
preparedStatement.setLong  (3, 123);

int rowsAffected = preparedStatement.executeUpdate();
```

Creating a PreparedStatement:

Before we can use a PreparedStatement we must first create it. We do so using the Connection.prepareStatement(), like this:

```
String sql = "select * from people where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);
```

The PreparedStatement is now ready to have parameters inserted.

Inserting Parameters into a PreparedStatement:

Everywhere we need to insert a parameter into our SQL, we write a question mark (?). For instance:

```
String sql = "select * from people where id=?";
```

Once a PreparedStatement is created (prepared) for the above SQL statement, we can insert parameters at the location of the question mark. This is done using the many setXXX() methods. Here is an example:

```
preparedStatement.setLong(1, 123);
```

The first number (1) is the index of the parameter to insert the value for. The second number (123) is the value to insert into the SQL statement.

Here is the same example with a bit more details:

```
String sql = "select * from people where id=?";

PreparedStatement preparedStatement =
      connection.prepareStatement(sql);

preparedStatement.setLong(123);
```

We can have more than one parameter in an SQL statement. Just insert more than one question mark. Here is a simple example:

```
String sql = "select * from people where firstname=? and lastname=?";

PreparedStatement preparedStatement =
      connection.prepareStatement(sql);

preparedStatement.setString(1, "John");
preparedStatement.setString(2, "Smith");
```

Executing the PreparedStatement:

Executing the PreparedStatement looks like executing a regular Statement. To execute a query, we call the executeQuery() or executeUpdate method. Here is an executeQuery() example:

```
String sql = "select * from people where firstname=? and lastname=?";

PreparedStatement preparedStatement =
      connection.prepareStatement(sql);

preparedStatement.setString(1, "John");
preparedStatement.setString(2, "Smith");

ResultSet result = preparedStatement.executeQuery();
```

As we can see, the executeQuery() method returns a ResultSet. Iterating the ResultSet is described in the Query the Database text.

Here is an executeUpdate() example:

```
String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement =
      connection.prepareStatement(sql);
```

```
preparedStatement.setString(1, "Gary");
preparedStatement.setString(2, "Larson");
preparedStatement.setLong  (3, 123);

int rowsAffected = preparedStatement.executeUpdate();
```

The executeUpdate() method is used when updating the database. It returns an int which tells how many records in the database were affected by the update.

Reusing a PreparedStatement:

Once a PreparedStatement is prepared, it can be reused after execution. We reuse a PreparedStatement by setting new values for the parameters and then execute it again. Here is a simple example:

```
String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");
preparedStatement.setString(2, "Larson");
preparedStatement.setLong  (3, 123);

int rowsAffected = preparedStatement.executeUpdate();

preparedStatement.setString(1, "Stan");
preparedStatement.setString(2, "Lee");
preparedStatement.setLong  (3, 456);

int rowsAffected = preparedStatement.executeUpdate();
```

This works for executing queries too, using the executeQuery() method, which returns a ResultSet.

PreparedStatement Performance:

It takes time for a database to parse an SQL string, and create a query plan for it. A query plan is an analysis of how the database can execute the query in the most efficient way.

If we submit a new, full SQL statement for every query or update to the database, the database has to parse the SQL and for queries create a query plan. By reusing an existing PreparedStatement, we can reuse both the SQL parsing and query plan for subsequent queries. This speeds up query execution, by decreasing the parsing and query planning overhead of each execution.

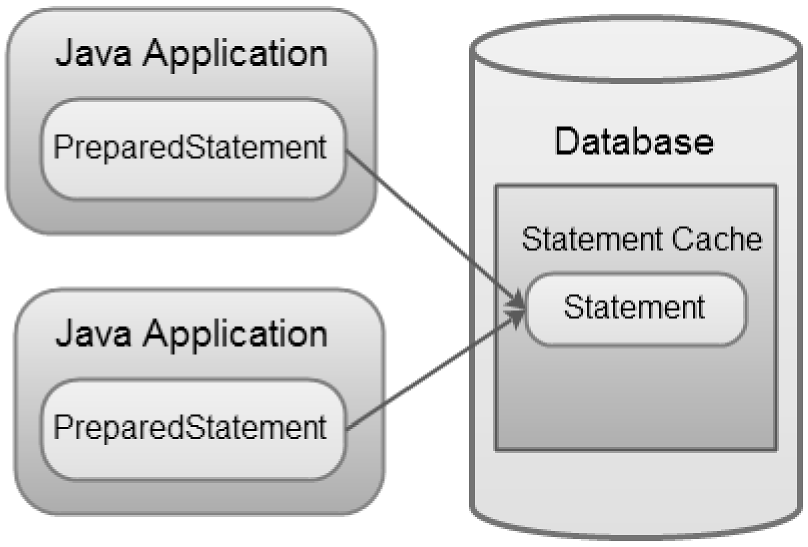
There are two levels of potential reuse for a PreparedStatement:

1. Reuse of PreparedStatement by the JDBC driver.
2. Reuse of PreparedStatement by the database.

First of all, the JDBC driver can cache PreparedStatement objects internally, and thus reuse the PreparedStatement objects. This may save a little of the PreparedStatement creation time.

Second, the cached parsing and query plan could potentially be reused across Java applications, for instance application servers in a cluster, using the same database.

Here is a diagram illustrating the caching of statements in the database:



The diagram does not show the JDBC driver PreparedStatement cache. We will have to imagine that. 😅

2. JDBC with images handling

We cannot Insert a picture in database directly. But we can store binary data of picture file. To insert image in database we need a column of type BLOB (Binary Large Object).

Key points:

- In Jdbc only PreparedStatement support the binary data transfer between a Java application to database.
- Jdbc supports only gif or jpeg or png type of images to insert or read from a database.
- To set binary data into a parameter of PreparedStatement object, you need to call setBinaryStream().

To insert images into a database, the database must support images. Images are stored in binary in a table cell. The data type for the cell is a binary large object (BLOB), which is a new SQL type in SQL3 for storing binary data. Another new SQL3 type is character large object (CLOB), for storing a large text in the character format. JDBC 2 introduced the interfaces java.sql.Blob and java.sql.Clob to support mapping for these new SQL types. JBDC 2 also added new methods, such as getBlob, setBinaryStream, getClob, setBlob, and setClob, in the interfaces ResultSet, PreparedStatement, and CallableStatement, to access SQL BLOB and CLOB values.

To store an image in a cell in a table, the corresponding column for the cell must be of the BLOB type. For example, the following SQL statement creates a table whose type for the flag column is BLOB:

```
create table Country(name varchar(30), flag blob,
description varchar(500));
```

In the preceding statement, the description column is limited to 500 characters. The upper limit for the VARCHAR type is 32,672 bytes. For a large character field, you can use the CLOB type, which can store up to 2GB characters.

To insert a record with images to a table, define a prepared statement like this one:

```
PreparedStatement pstmt = connection.prepareStatement(
"insert into Country values(?, ?, ?)");
```

Images are usually stored in files. We may first get an instance of InputStream for an image file and then use the setBinaryStream method to associate the input stream with cell in the table, as follows:

```
// Store image to the table cell
File file = new File(imageFilenames[i]);
InputStream inputImage = new FileInputStream(file);
pstmt.setBinaryStream(2, inputImage, (int)(file.length()));
```

To retrieve an image from a table, we can use the getBlob method, as shown here:

```
// Store image to the table cell
Blob blob = rs.getBlob(1);
ImageIcon imageIcon = new ImageIcon(
blob.getBytes(1, (int)blob.length()));
```

Example to Insert image in database:

```
import java.sql.*;
import java.util.*;
import java.io.*;
class PhotoInsert
{
Connection con;
public void openCon()throws Exception
{
Class.forName("oracle.jdbc.OracleDriver");
con=DriverManager.getConnection("jdbc:oracle:thin:@rama-pc:1521:xe","system","system");
System.out.println("connection is opened");
}


public void insert()throws Exception
{
Scanner s=new Scanner(System.in);
PreparedStatement pstmt=con.prepareStatement("insert into emp_info values(?,?,?)");
System.out.println("enter emp id");
int empid=s.nextInt();
pstmt.setInt(1,empid);
System.out.println("enter emp name");
String empname=s.next();
pstmt.setString(2,empname);
System.out.println("enter photo file path");
String path=s.next();


File f=new File("c:/pho001.gif");
int size=(int) f.length();
FileInputStream fis=new FileInputStream(f);
pstmt.setBinaryStream(3,fis,size);
int i=pstmt.executeUpdate();
System.out.println(i+"row inserted");
pstmt.close();
fis.close();
}


public void closeCon()throws Exception
{
```

```
con.close();
}

public static void main(String[] args)throws Exception
{
PhotoInsert p1= new PhotoInsert();
 p1.openCon();
 p1.insert();
 p1.closeCon();
 }
}
```

3. JDBC with file handling

In general, the contents of a file are stored under Clob (TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT) datatype in MySQL database. JDBC provides support for the Clob datatype, to store the contents of a file in to a table in a database. The setCharacterStream() method of the PreparedStatement interface accepts an integer representing the index of the parameter and, a Reader object as a parameter. And sets the contents of the given reader object (file) as value to the parameter (place holder) in the specified index. Whenever we need to send very large text value, we can use this method.

Storing text file using JDBC:

If you need to store a file in a database using JDBC program create table with a Clob (TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT) datatype as shown below:

```
CREATE TABLE Articles(Name VARCHAR(255), Article LONGTEXT);
```

Now, using JDBC we connect to the database and prepare a PreparedStatement to insert values into the above created table:

```
String query = "INSERT INTO Tutorial(Name, Article) VALUES (?,?)";
PreparedStatement pstmt = con.prepareStatement(query);
```

Set values to the place holders using the setter methods of the PreparedStatement interface and for the Clob datatype set value using the setCharacterStream() method.

Example:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;




public class StoreFileExample {
   public static void main(String... arg) {
        Connection con =null;
        PreparedStatement prepStmt = null;
        File file= null;
        FileReader fr = null;
        String filePath="c:/myTxt.txt";
        try {
            // registering Oracle driver class
            Class.forName("oracle.jdbc.driver.OracleDriver");


            // getting connection
            con = DriverManager.getConnection(
                    "jdbc:oracle:thin:@localhost:1521:orcl",
                    "sharadindu", "Oracle123");
            System.out.println("Connection established successfully!");


            //create File object
            file=new File(filePath);


            //create a FileReader object which will read from File
            fr=new FileReader(file);


            prepStmt=con.prepareStatement("INSERT into TEST_FILES (ID, FILE_COL) "
                    + "values (1, ?)");
            prepStmt.setCharacterStream(1,fr);


            //execute insert query
            int numberOfRowsInserted = prepStmt.executeUpdate();
            System.out.println("numberOfRowsInserted = " + numberOfRowsInserted);


            System.out.println(filePath+" > File stored in database");




        } catch (Exception e) {
            e.printStackTrace();
        }


        finally{
            try {
                if(fr!=null) fr.close(); //close reader
                if(prepStmt!=null) prepStmt.close(); //close PreparedStatement
                if(con!=null) con.close(); // close connection
            } catch (SQLException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}


/*OUTPUT
Connection established successfully!
numberOfRowsInserted = 1
c:/myTxt.txt > File stored in database */
```