JAVA PROGRAMMING

Week 8: JDBC

Lecturer:

- HO Tuan Thanh, M.Sc.

JDBC

- JDBC = **J**ava **D**ata**B**ase **C**onnectivity
- JDBC library includes APIs for the following tasks:
  - Connect to databases
  - Create SQL or MySQL statements
  - Execute SQL or MySQL queries in the database.
  - View and modify the resulting records.
- Database access is the same for all database vendors
- The JVM uses a JDBC driver to translate generalized JDBC calls into vendor specific database calls.
- There are four general types of JDBC drivers

Implementations of JDBC drivers [1]

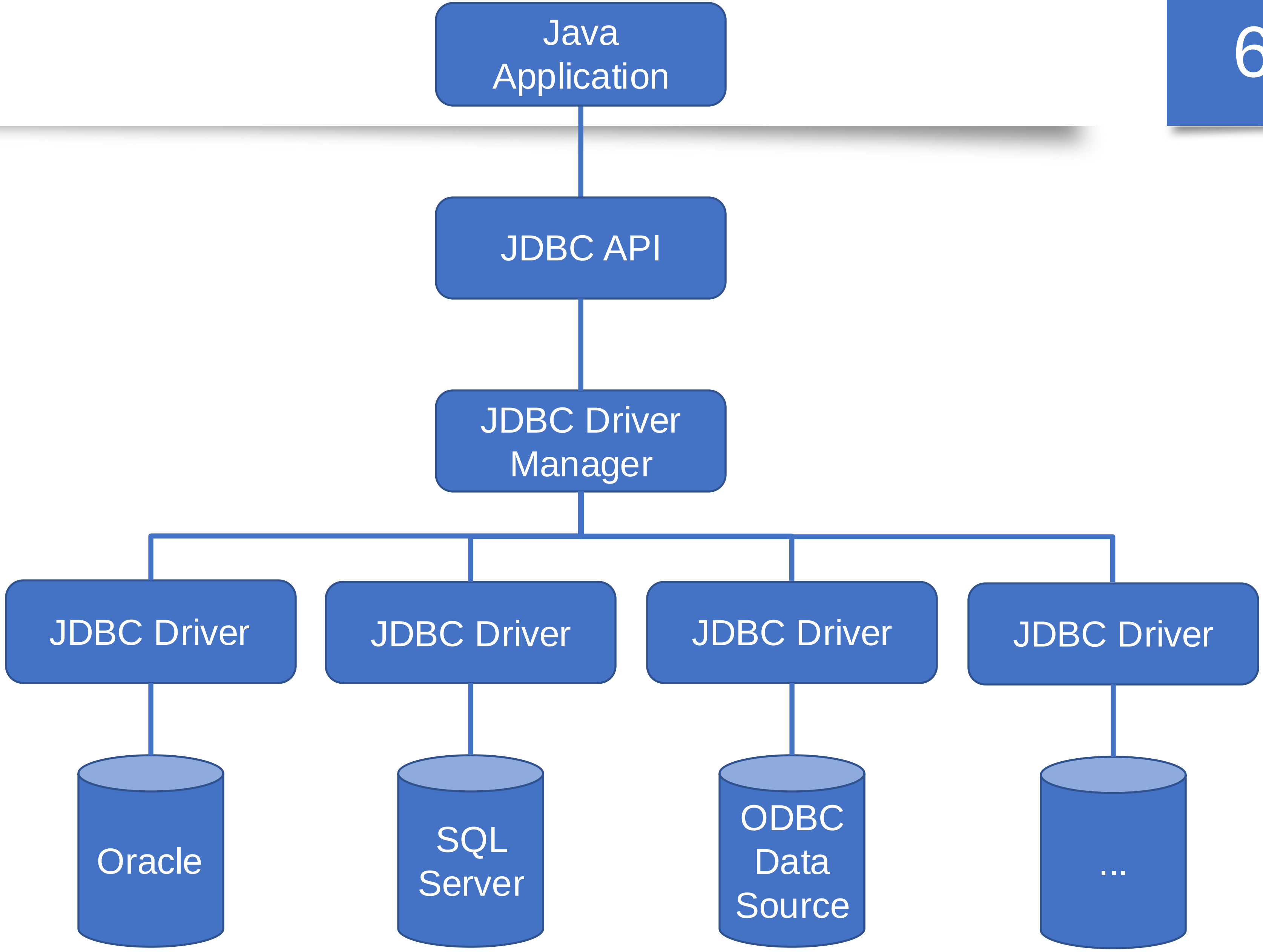
- Type 1: Drivers that implement the JDBC API as a mapping to another data access API, such as ODBC (Open Database Connectivity).
  - Drivers of this type are generally dependent on a native library, which limits their portability.
  - The JDBC-ODBC Bridge is an example of a Type 1 driver.
- Type 2: Drivers that are written partly in the Java programming language and partly in native code.
  - These drivers use a native client library specific to the data source to which they connect.
  - Because of the native code, their portability is limited.
  - Oracle's OCI (Oracle Call Interface) client-side driver is an example of a Type 2 driver.

Implementations of JDBC drivers [2]

- Type 3: Drivers that use a pure Java client and communicate with a middleware server using a database-independent protocol.
  - The middleware server then communicates the client's requests to the data source.
- Type 4: Drivers that are pure Java and implement the network protocol for a specific data source.
  - The client connects directly to the data source.
- Java DB comes with two Type 4 drivers, an Embedded driver and a Network Client Driver.
  - MySQL Connector/J is a Type 4 driver.

Pure Java Driver (Type 4)

- Converts the JDBC API calls to direct network calls using vendor-specific networking protocols by making direct socket connections with the database
- The most efficient method to access database
  - Both in performance and development time
- Simplest to deploy
- All major database vendors provide pure Java JDBC drivers for their databases and they are also available from third party vendors


Java Application
JDBC API
JDBC Driver Manager
JDBC Driver
JDBC Driver
JDBC Driver
JDBC Driver
Oracle
SQL Server
ODBC Data Source
...

Common JDBC Components [1]

- **DriverManager**
  - Manages a list of database drivers.
  - Matches connection requests from the Java application with the proper database driver using communication sub protocol.
- **Driver**
  - Handles the communications with the database server.
- **Connection**
  - Establish the connection with a database.

Common JDBC Components [2]

- **Statement**
  - You use objects created from this interface to submit the SQL statements to the database.
- **ResultSet**
  - These objects hold data retrieved from a database after you execute an SQL query using Statement objects.
  - It acts as an iterator to allow you to move through its data.
- **SQLException**
  - This class handles any errors that occur in a database application.

Setup JDBC environment

- Install Database
- Install Database Drivers
- Set Database Credential

Typical JDBC Programming Procedure

1. Load the database driver
2. Obtain a connection
3. Create and execute statements (SQL queries)
4. Use result sets (tables) to navigate through the results
5. Close the connection

Register JDBC Driver

- Class.forName()

- DriverManager.registerDriver()

Example 1

```
try{
        Class.forName("com.mysql.cj.jdbc.Driver");
}catch(ClassNotFoundException ex){
  System.out.println("Error: unable to load driver class.");
  System.exit(1);
}
```

Example 2

```
try {
	Driver myDriver = new com.mysql.cj.jdbc.Driver();
	DriverManager.registerDriver(myDriver);
} catch (SQLException ex) {
	System.out.println("Error: Unable to load driver class.");
	System.exit(1);
}
```

Database URL Formulation

- To establish a connection:

```
getConnection(String url)

getConnection(String url, Properties prop)

getConnection(String url, String user,
                                        String password)
```

- To close a connection:

```
close();
```

DBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql://**hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:**@hostname: port Number:databaseName
DB2	com.ibm.db2.jcc.DB2Driver	**jdbc:db2://**hostname:port Number/databaseName
SQL Server	com.microsoft.sqlserver.jdbc.SQL ServerDriver	jdbc:sqlserver://hostname:port Number/databaseName

Using a Database URL with a username and password

```java
// Database URL
static final String DB_URL = "jdbc:mysql://localhost/";
// Database credentials
static final String USER = "username";
static final String PASS = "password";

// Open a connection
Connection conn = DriverManager.getConnection(DB_URL,
                                                                                USER, PASS);
```

Using Only a Database URL

```
static final String DB_URL =
"jdbc:mysql://localhost?user=username&password=password";

Connection conn = DriverManager.getConnection(DB_URL);
```

Using a Database URL and a Properties Object

```
static final String DB_URL = "jdbc:mysql://localhost/";

Properties info = new Properties();
info.put("user", ”username");
info.put("password", ”password");

Connection conn = DriverManager.getConnection(DB_URL, info);
```

Interfaces

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

Statement Objects

```
Statement stmt = null;
try {
  stmt = conn.createStatement( );
  //. . .
}
catch (SQLException e) {
  //. . .
}
finally {
  stmt.close();
}
```

Methods

- boolean execute (String SQL)
  - Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false.
  - Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- int executeUpdate (String SQL)
  - Returns the number of rows affected by the execution of the SQL statement.
  - Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- ResultSet executeQuery (String SQL)
  - Returns a ResultSet object.
  - Use this method when you expect to get a result set, as you would with a SELECT statement.

PreparedStatement

- PreparedStatement interface extends the Statement interface
    - Provides added functionality with a couple of advantages over a generic Statement object.

PreparedStatement Objects

```
PreparedStatement pstmt = null;
try {
  String SQL = "Update Employees SET age = ? WHERE id = ?";
  pstmt = conn.prepareStatement(SQL);
  // . . .
} catch (SQLException e) {
  // . . .
} finally {
  pstmt.close();
}
```

Example

```
System.out.println("Creating statement...");
String sql = "UPDATE Employees set age=? WHERE id=?";
PreparedStatement stmt = conn.prepareStatement(sql);

//Bind values into the parameters.
stmt.setInt(1, 35);  // This would set age
stmt.setInt(2, 102); // This would set ID

// Let us update age of the record with ID = 102;
int rows = stmt.executeUpdate();
System.out.println("Rows impacted : " + rows );
```

ResultSet

- SQL statements that read data from a database query, return the data in a result set.
- The SELECT statement is the standard way to select rows from a database and view them in a result set.
- In Java: java.sql.ResultSet interface represents the result set of a database query.
- A ResultSet object maintains a cursor that points to the current row in the result set.
  - The term "result set" refers to the row and column data contained in a ResultSet object.

Three categories of ResultSet methodes

- Navigational methods:
  - Used to move the cursor around.
- Get methods:
  - Used to view the data in the columns of the current row being pointed by the cursor.
- Update methods:
  - Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.
- The cursor is movable based on the properties of the ResultSet.
  - These properties are designated when the corresponding Statement that generates the ResultSet is created.

ResultSet methodes

- These are connection methods to create statements with desired ResultSet:

```
createStatement(int RSType, int RSConcurrency);

prepareStatement(String SQL, int RSType, int RSConcurrency);

prepareCall(String sql, int RSType, int RSConcurrency);
```

  - RSType: the type of a ResultSet object and
  - RSConcurrency: one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Types of ResultSet

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency of ResultSet

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

```
try {
  Statement stmt = conn.createStatement(
                        ResultSet.TYPE_FORWARD_ONLY,
                        ResultSet.CONCUR_READ_ONLY);
}catch(Exception ex) {
  // ....
}finally {
  // ....
}
```

Navigating a Result Set [1]

- public void beforeFirst() throws SQLException
  - Moves the cursor just before the first row.
- public void afterLast() throws SQLException
  - Moves the cursor just after the last row.
- public boolean first() throws SQLException
  - Moves the cursor to the first row.
- public void last() throws SQLException
  - Moves the cursor to the last row.
- public boolean absolute(int row) throws SQLException
  - Moves the cursor to the specified row.
- public boolean relative(int row) throws SQLException
  - Moves the cursor the given number of rows forward or backward, from where it is currently pointing.

Navigating a Result Set [2]

- public boolean previous() throws SQLException
  - Moves the cursor to the previous row.
  - This method returns false if the previous row is off the result set.
- public boolean next() throws SQLException
  - Moves the cursor to the next row.
  - This method returns false if there are no more rows in the result set.
- public int getRow() throws SQLException
  - Returns the row number that the cursor is pointing to.
- public void moveToInsertRow() throws SQLException
  - Moves the cursor to a special row in the result set that can be used to insert a new row into the database.
  - The current cursor location is remembered.
- public void moveToCurrentRow() throws SQLException
  - Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

Data type

- JDBC driver converts the Java data type to the appropriate JDBC type, before sending it to the database.
- It uses a default mapping for most data types.
  - Example: a Java int is converted to an SQL INTEGER.
- Default mappings were created to provide consistency between drivers.

Java types to JDBC data types [1]

SQL	JDBC/Java	setXXX	updateXXX
VARCHAR	java.lang.String	setString	updateString
CHAR	java.lang.String	setString	updateString
LONGVARCHAR	java.lang.String	setString	updateString
BIT	boolean	setBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
TINYINT	byte	setByte	updateByte
SMALLINT	short	setShort	updateShort
INTEGER	int	setInt	updateInt
BIGINT	long	setLong	updateLong
REAL	float	setFloat	updateFloat
FLOAT	float	setFloat	updateFloat

Java types to JDBC data types [1]

SQL	JDBC/Java	setXXX	updateXXX
DOUBLE	double	setDouble	updateDouble
VARBINARY	byte[ ]	setBytes	updateBytes
BINARY	byte[ ]	setBytes	updateBytes
DATE	java.sql.Date	setDate	updateDate
TIME	java.sql.Time	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	updateClob
BLOB	java.sql.Blob	setBlob	updateBlob
ARRAY	java.sql.Array	setARRAY	updateARRAY
REF	java.sql.Ref	SetRef	updateRef
STRUCT	java.sql.Struct	SetStruct	updateStruct

Retrieve column value [1]

SQL	JDBC/Java	setXXX	getXXX
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
TINYINT	byte	setByte	getByte
SMALLINT	short	setShort	getShort
INTEGER	int	setInt	getInt
BIGINT	long	setLong	getLong
REAL	float	setFloat	getFloat
FLOAT	float	setFloat	getFloat

Retrieve column value [2]

SQL	JDBC/Java	setXXX	getXXX
DOUBLE	double	setDouble	getDouble
VARBINARY	byte[ ]	setBytes	getBytes
BINARY	byte[ ]	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

Date and Time Data Types

- java.sql.Date class maps to the SQL DATE type,
- java.sql.Time class maps to the SQL TIME data type.
- java.sql.Timestamp class maps to the SQL TIMESTAMP data type.

Example [1]

```java
public class SqlDateTime {
    public static void main(String[] args) {
        // Get standard date and time
        java.util.Date javaDate = new java.util.Date();
        long javaTime = javaDate.getTime();
        System.out.println("The Java Date is:" +
                                    javaDate.toString());
        // Get and display SQL DATE
        java.sql.Date sqlDate = new java.sql.Date(javaTime);
        System.out.println("The SQL DATE is: " +
                                    sqlDate.toString());
```

Example [2]

```
            // Get and display SQL TIME
            java.sql.Time sqlTime = new java.sql.Time(javaTime);
            System.out.println("The SQL TIME is: " +
                                                    sqlTime.toString());
            // Get and display SQL TIMESTAMP
            java.sql.Timestamp sqlTimestamp = new
                                          java.sql.Timestamp(javaTime);
            System.out.println("The SQL TIMESTAMP is: " +
                                                    sqlTimestamp.toString());
        }// end main
    }// end SqlDateTime
```

Handling NULL values

- SQL's use of NULL values and Java's use of null are different concepts → to handle SQL NULL values in Java:
  - Avoid using getXXX( ) methods that return primitive data types.
  - Use wrapper classes for primitive data types, and use the ResultSet object's wasNull() method to test whether the wrapper class variable that received the value returned by the getXXX() method should be set to null.
  - Use primitive data types and the ResultSet object's wasNull() method to test whether the primitive variable that received the value returned by the getXXX( ) method should be set to an acceptable value that you've chosen to represent a NULL.

Handling NULL Values

```java
Statement stmt = conn.createStatement( );
String sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);

int id = rs.getInt(1);
if(rs.wasNull()){
   id = 0;
}
```

Transactions

- If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.
- Three reasons to turn off the auto-commit and manage your own transactions:
  - To increase performance.
  - To maintain the integrity of business processes.
  - To use distributed transactions.
- To set or turn off auto-commit:

```
conn.setAutoCommit(boolean);
```

Commit & Rollback

- To commit the changes using Connection named conn:

```
conn.commit();
```

- To roll back updates to the database made using the Connection named conn:

```
conn.rollback( );
```

```
try{
  //Assume a valid connection object conn
  conn.setAutoCommit(false);
  Statement stmt = conn.createStatement();
  String SQL = "INSERT INTO Employees  " +
               "VALUES (106, 20, 'Rita', 'Tez')";
  stmt.executeUpdate(SQL);
  //Submit a malformed SQL statement that breaks
  String SQL = "INSERTED IN Employees  " +
               "VALUES (107, 22, 'Sita', 'Singh')";
  stmt.executeUpdate(SQL);
  // If there is no error.
  conn.commit();
}catch(SQLException se){
  // If there is any error.
  conn.rollback();
}
```

Exceptions Handling

Method	Description
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error, handled by the driver or gets the Oracle error number and message for a database error.
getSQLState( )	Gets the XOPEN SQLstate string. For a JDBC driver error, no useful information is returned from this method. For a database error, the five-digit XOPEN SQLstate code is returned. This method can return null.
getNextException( )	Gets the next Exception object in the exception chain.
printStackTrace( )	Prints the current exception, or throwable, and it's backtrace to a standard error stream.
printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the print stream you specify.
printStackTrace(PrintWriter w)	Prints this throwable and it's backtrace to the print writer you specify.

```
import java.sql.*;
public class JDBCExample {
   // JDBC driver name and database URL
   static final String JDBC_DRIVER =
                                                                  "com.mysql.jdbc.Driver";
   static final String DB_URL = "jdbc:mysql://localhost/EMP";
   //  Database credentials
   static final String USER = "username";
   static final String PASS = "password";
   public static void main(String[] args) {
   Connection conn = null;
   try{
      //STEP 2: Register JDBC driver
      Class.forName("com.mysql.jdbc.Driver");
      //STEP 3: Open a connection
      System.out.println("Connecting to database...");
      conn = DriverManager.getConnection(DB_URL,USER,PASS);
```

```
//STEP 4: Execute a query
System.out.println("Creating statement...");
Statement stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);
//STEP 5: Extract data from result set
while(rs.next()){
  //Retrieve by column name
  int id  = rs.getInt("id");
  int age = rs.getInt("age");
  String first = rs.getString("first");
 String last = rs.getString("last");
  //Display values
  System.out.print("ID: " + id);
  System.out.print(", Age: " + age);
  System.out.print(", First: " + first);
  System.out.println(", Last: " + last);
}
```

```
              //STEP 6: Clean-up environment
              rs.close();
           stmt.close();
           conn.close();
         }catch(SQLException se){
           //Handle errors for JDBC
           se.printStackTrace();
         }catch(Exception e){ //Handle errors for Class.forName
           e.printStackTrace();
         }finally{ //finally block used to close resources
           try{  if(conn!=null) conn.close();
           }catch(SQLException se){ se.printStackTrace();
           }//end finally try
         }//end try
         System.out.println("Goodbye!");
        }//end main
   }//end JDBCExample
```

Examples

Example: Create a DB

```
public class JDBCCreateDB {
    // JDBC driver name and database URL
    static final String JDBC_DRIVER =
                                    "com.mysql.cj.jdbc.Driver";
    static final String DB_URL = "jdbc:mysql://localhost/";
    // Database credentials
    static final String USER = ”user";
    static final String PASS = ”password";
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
```

```
try {
        // STEP 2: Register JDBC driver
        Class.forName(JDBC_DRIVER);
        // STEP 3: Open a connection
        System.out.println("Connecting to database...");
        conn = DriverManager.getConnection(DB_URL, USER, PASS);
        // STEP 4: Execute a query
        System.out.println("Creating database...");
        stmt = conn.createStatement();
        String sql = "CREATE DATABASE STUDENTS";
        stmt.executeUpdate(sql);
        System.out.println("Database created successfully...");
} catch (SQLException se) { // Handle errors for JDBC
        se.printStackTrace();
```

```
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                // finally block used to close resources
                try { if (stmt != null) stmt.close();
                } catch (SQLException se2) { }
                try { if (conn != null) conn.close();
                } catch (SQLException se) { se.printStackTrace();
                } // end finally try
            } // end try
            System.out.println("Goodbye!");
        }// end main
    }
```

Drop a DB

```
System.out.println("Deleting database...");
stmt = conn.createStatement();

String sql = "DROP DATABASE STUDENTS";
stmt.executeUpdate(sql);
```

Create a table

```
stmt = conn.createStatement();

String sql = "CREATE TABLE REGISTRATION " +
                        "(id INTEGER not NULL, " +
                        " first VARCHAR(255), ” +
                        " last VARCHAR(255), " +
                        " age INTEGER, " +
                        " PRIMARY KEY ( id ))";

stmt.executeUpdate(sql);
```

Drop a table

```
stmt = conn.createStatement();
String sql = "DROP TABLE REGISTRATION ";
stmt.executeUpdate(sql);
```

Insert a record

```
stmt = conn.createStatement();
String sql = "INSERT INTO Registration "
                                        + "VALUES (100, 'Zara', 'Ali', 18)";
stmt.executeUpdate(sql);
sql = "INSERT INTO Registration "
                                        + "VALUES (101, 'Mahnaz', 'Fatma', 25)";
stmt.executeUpdate(sql);
sql = "INSERT INTO Registration "
                                        + "VALUES (102, 'Zaid', 'Khan', 30)";
stmt.executeUpdate(sql);
sql = "INSERT INTO Registration "
                                        + "VALUES(103, 'Sumit', 'Mittal', 28)";
stmt.executeUpdate(sql);
```

Select a record

```
stmt = conn.createStatement();
String sql = "SELECT id, first, last, age FROM”
                                                            + “ Registration";
ResultSet rs = stmt.executeQuery(sql);
while (rs.next()) { // Retrieve by column name
    int id = rs.getInt("id"); int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");
    // Display values
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
}
rs.close();
```

Update a record

```
stmt = conn.createStatement();
String sql = "UPDATE Registration "
                                    + "SET age = 30 WHERE id in (100, 101)";
stmt.executeUpdate(sql);
```

Delete a record

```
stmt = conn.createStatement();
String sql = "DELETE FROM Registration " + "WHERE id = 101";
stmt.executeUpdate(sql);
```

WHERE clause

```
stmt = conn.createStatement();
sql = "SELECT id, first, last, age FROM Registration"
                                                        + " WHERE id >= 101 ";
rs = stmt.executeQuery(sql);
while (rs.next()) { // Retrieve by column name
        int id = rs.getInt("id"); int age = rs.getInt("age");
        String first = rs.getString("first");
        String last = rs.getString("last");
        // Display values
        System.out.print("ID: " + id);
        System.out.print(", Age: " + age);
        System.out.print(", First: " + first);
        System.out.println(", Last: " + last);
}
```

QUESTION ?