Introduction to JDBC

Java Database Connectivity

Contents

1. Introduction to JDBC
2. Querying the database
3. Different statements
4. Handling exceptions
5. Transactions
6. Best practices

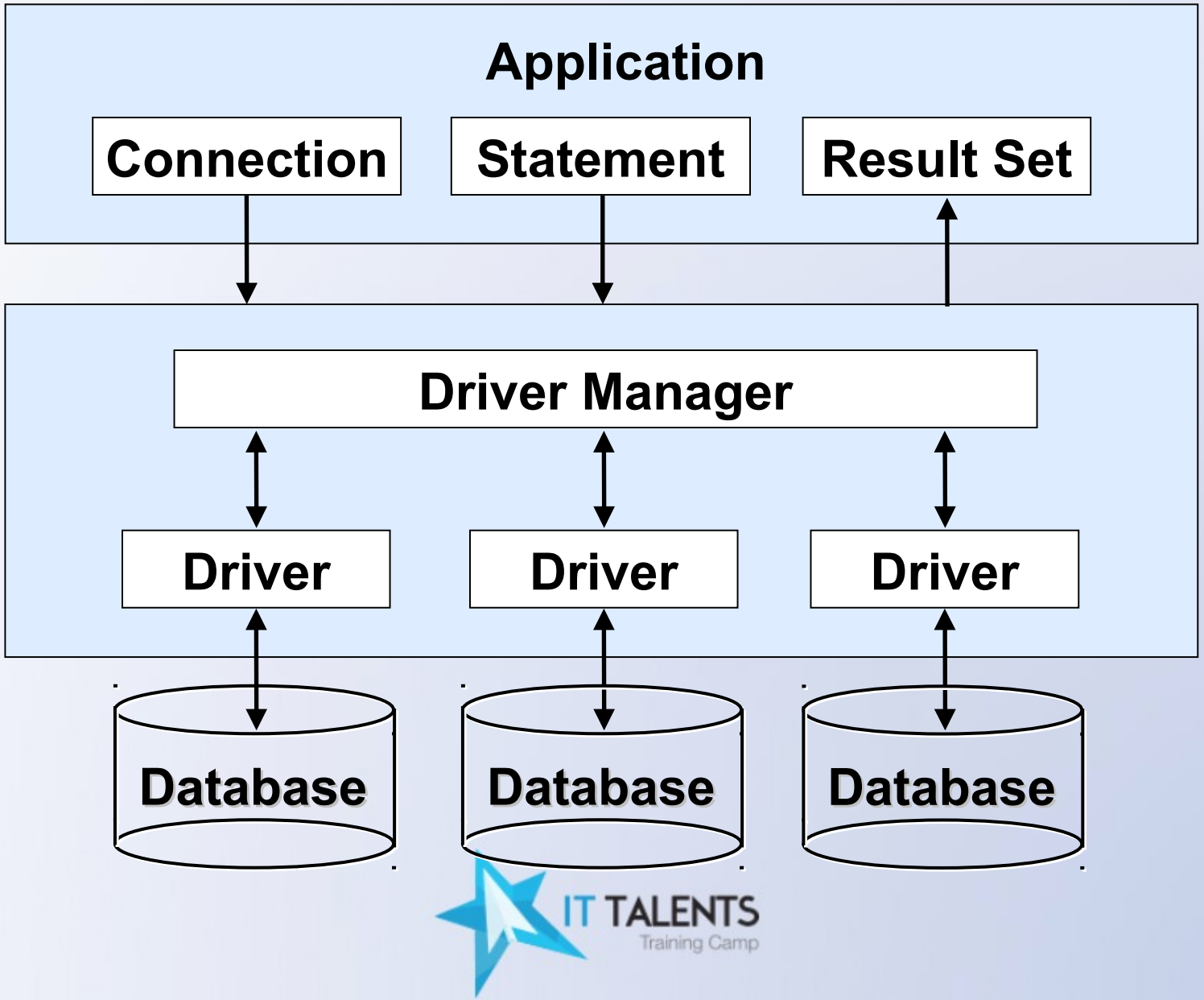
JDBC

Technology Overview

About JDBC

- JDBC is a standard interface for connecting to relational databases from Java
  - The JDBC Core API package in `java.sql`
  - JDBC 2.0 Optional Package API in `javax.sql`
  - JDBC 3.0 API includes the Core API and Optional Package API

JDBC Architecture



JDBC Components

- Driver Manager
  - Loads database drivers, and manages the connection between application & driver
- Driver
  - Translates API calls to operations for a specific data source
- Connection
  - A session between an application and a database driver

JDBC Components (2)

- Statement
  - A SQL statement to perform a query or an update operation
- Metadata
  - Information about the returned data, driver and the database
- Result Set
  - Logical set of columns and rows returned by executing a statement

Data Access with JDBC

Querying the Database

Stage 1: Loading Drivers

- Loading the JDBC driver is done by a single line of code:

```
Class.forName("<jdbc driver class>");
```

- Your driver documentation will give you the class name to use
- Loading MySQL JDBC driver

```
Class.forName("com.mysql.jdbc.Driver");
```

Stage 2: Establishing a Connection

- The following line of code illustrates the process:

```
Connection con = DriverManager.

GetConnection("jdbc:mysql://localhost:3306/MyDat
abase", "root", "root");
```

- If you are using a JDBC driver developed by a third party, the documentation will tell you what subprotocol to use

Stage 3: Creating Statement

- A `Statement` object sends the SQL commands to the DBMS
  - `executeQuery()` is used for `SELECT` statements
  - `executeUpdate()` is used for statements that create/modify tables
- An instance of active `Connection` is used to create a `Statement` object

```
Connection dbCon =
DriverManager.getConnection("...");
Statement stmt = dbCon.createStatement();
```

Stage 4: Executing Query

- `executeQuery()` executes SQL command through a previously created statement
  - Returns the results in a `ResultSet` object

```
Connection dbCon =
  DriverManager.getConnection(...);

Statement stmt = dbCon.createStatement();

ResultSet rs = stmt.executeQuery(
  "SELECT first_name FROM employees");
```

Stage 4: Executing Statement

- `executeUpdate()` is used to submit DML/DDL SQL statements
  - DML is used to manipulate existing data in objects (using `UPDATE`, `INSERT`, `DELETE` statements)
  - DDL is used to manipulate database objects (`CREATE`, `ALTER`, `DROP`)

```
Statement stmt = dbCon.createStatement();
int rowsAffected = stmt.executeUpdate(
  "UPDATE employees SET salary = salary*1.2");
```

Stage 5: Process The Returned Results

- The `ResultSet` object
  - Maintains a cursor pointing to its current row of data
  - Provides methods to retrieve column values

```
ResultSet rs = stmt.executeQuery(
  "SELECT last_name, salary FROM employees");
while (rs.next()) {
  String name = rs.getString("last_name");
  double salary = rs.getDouble("salary");
  System.out.println(name + " " + salary);
}
```

Stage 6: Closing the Connection

- Explicitly close a `Connection`, `Statement`, and `ResultSet` to release resources that are no longer needed

```
try {
  Connection conn = ...;
  Statement stmt = ...;
  ResultSet rset = stmt.executeQuery(...);
  ...
} finally
    // clean up
    rset.close();
    stmt.close();
    conn.close();
}
```

SQLException

- SQL statements can throw `java.sql.SQLException` during their execution

```
try  {
  rset = stmt.executeQuery(
    "SELECT first_name, last_name FROM employee");
  } catch (SQLException sqlex) {
    ... // Handle SQL errors here
  } finally {
    // Clean up all used resources
    try {
      if (rset != null) rset.close();
    } catch (SQLException sqlex)  {
      ... // Ignore closing errors
    }
  ...
}
```

Querying MySQL Database through JDBC

Live Demo

JDBC Statements

`Statement`, `PreparedStatement` and `CallableStatement` Interfaces

Submitting DML Statements That Change the Database

1. Create an empty statement object

```
Statement stmt = conn.createStatement();
```

1. Use `executeUpdate()` to execute the statement

```
int count = stmt.executeUpdate(sql_dml_statement);
```

- Example

```
Statement stmt = conn.createStatement();

int rowsDeleted = stmt.executeUpdate("DELETE FROM
  order_items WHERE order_id = 2354");
```

Submitting DDL Statements

1. Create an empty statement object

```
Statement stmt = conn.createStatement();
```

1. Use `executeUpdate()` to execute the statement

```
int count = stmt.executeUpdate(sql_ddl_statement);
```

- Example

```
Statement stmt = conn.createStatement();

stmt.executeUpdate("CREATE TABLE
  temp(col1 NUMBER(5,2), col2 VARCHAR2(30))");
```

Unknown Statements

1. Create an empty statement object

```
Statement stmt = conn.createStatement();
```

1. Use `execute()` to execute the statement

```
boolean result = stmt.execute(SQLstatement);
```

1. Process the statement accordingly

```
if (result) { // was a query - process results
  ResultSet r = stmt.getResultSet(); ...
}
else { // was an update or DDL - process result
  int count = stmt.getUpdateCount(); ...
}
```

Prepared Statements

- `PreparedStatement` is used to
  - Execute a statement that takes parameters
  - Execute given statement many times

```
String insertSQL = "INSERT INTO employees(" +
  "first_name, last_name, salary) VALUES(?,?,?)";
PreparedStatement stmt =
  con.prepareStatement(insertSQL);
stmt.setString(1, "Krasimir");
stmt.setString(2, "Stoev");
stmt.setDouble(3, 8100.0);
stmt.executeUpdate();
```

Prepared Statements – Best Practices

- Use prepared statements always when you need parameterized query
  - Do not build query by "+" operator
- Prefer named parameters

```
String insertSQL = "INSERT INTO employees(" +
  "first_name, last_name) VALUES(@fn, @ln)";
PreparedStatement stmt =
  con.prepareStatement(insertSQL);
stmt.setString("@fn", "Krasi");
stmt.setString("@ln", "Stoev");
stmt.executeUpdate();
```

Retrieving Auto Generated Primary Key

- Some databases support "auto increment" primary key columns
  - E. g. MS SQL Server, MS Access, MySQL, ...
  - JDBC can retrieve auto generated keys

```
// Insert row and return PK
int rowCount = stmt.executeUpdate(
  "INSERT INTO Messages(Msg) VALUES ('Test')",
  Statement.RETURN_GENERATED_KEYS);

// Get the auto generated PK
ResultSet rs = stmt.getGeneratedKeys();
rs.next();
long primaryKey = rs.getLong(1);
```

Callable Statements

- `CallableStatement` interface
  - Is used for executing stored procedures
  - Can pass input parameters
  - Can retrieve output parameters
- Example

```
CallableStatement callStmt =
  dbCon.prepareCall("call SP_Insert_Msg(?,?)");
callStmt.setString(1, msgText);
callStmt.registerOutParameter(2, Types.BIGINT);
callStmt.executeUpdate();
long id = callStmt.getLong(2);
```

JDBC Statements

Live Demo

Transactions Management in JDBC

Database Transactions

- A database transaction is a set of database operations that must be either entirely completed or aborted
- A simple transaction is usually in this form:
    1. Begin the transaction
    2. Execute several SQL DML statements
    3. Commit the transaction
- If one of the SQL statements fails, rollback the entire transaction

JDBC Transactions

- JDBC transaction mode:
  - Auto-commit by default
  - Can be turned off by calling `setAutoCommit(false)`
- In auto-commit mode each statement is treated as a separate transaction
- If the auto-commit mode is off, no changes will be committed until `commit()` is invoked
- Auto-commit mode can be turned back on by calling `setAutoCommit(true)`

JDBC Transactions – Example

- If we don't want certain changes to be made permanent, we can issue `rollback()`

```
dbCon.setAutoCommit(false);
try {
  Statement stmt = con.createStatement();
  stmt.executeUpdate("INSERT INTO Groups " +
    "VALUES (101, 'Administrators')");
  stmt.executeUpdate("INSERT INTO Users " +
    "VALUES (NULL, 'Mary', 101)");
  dbCon.commit();
} catch (Exception ex) {
  dbCon.rollback();
  throw ex;
}
```

Transactions in JDBC

Live Demo

JDBC Advanced Features and Best Practices

Connection Pool

- Connection pool contains a number of open database connections
- There are a few choices when using connection pool:
  1. Depend on application server
  2. Use JDBC 2.0 interfaces (`ConnectionPoolDataSource` and `PooledConnection`)
  3. Create your own connection pool

Optimal Isolation Level

- You can set the transaction isolation level by calling `setTransactionIsolation(level)`

Transaction Level	Permitted Phenomena			Impact
	Dirty Reads	Non-Repeatable Reads	Phantom Reads	
TRANSACTION_NONE	-	-	-	FASTEST
TRANSACTION_READ_UNCOMMITED	YES	YES	YES	FASTEST
TRANSACTION_READ_COMMITED	NO	YES	YES	FAST
TRANSACTION_REPEATABLE_READ	NO	NO	YES	MEDIUM
TRANSACTION_SERIALIZABLE	NO	NO	NO	SLOW

ResultSet Metadata

- `ResultSetMetaData` class
  - Used to get information about the types and properties of the columns in a `ResultSet` object

```
ResultSet rs = stmt.executeQuery(
  "SELECT * FROM employees");
ResultSetMetaData rsm = rs.getMetaData();
int number = rsm.getColumnCount();
for (int i=0; i<number; i++) {
  System.out.println(rsm.getColumnName(i));
}
```

Close Unneeded Resources

- Closing connections, statements and result sets explicitly allows garbage collector to recollect memory and resources as early as possible
- Close statement object as soon as you finish working with them
- Use `try-finally` statement to guarantee that resources will be freed even in case of exception

Choose the Right Statement

- Use `PreparedStatement` when you execute the same statement more than once
- Use `CallableStatement` when you want result from multiple and complex statements for a single request

Use Batch Updates/Retrieval

- Send multiple queries to reduce the number of JDBC calls and improve performance:

```
statement.addBatch("sql_query1");
statement.addBatch("sql_query2");
statement.addBatch("sql_query3");

statement.executeBatch();
```

- You can improve performance by increasing number of rows to be fetched at a time

```
statement.setFetchSize(30);
```

Optimize the SQL Queries

- **Bad:**

```
Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(
  "SELECT * FROM EMPLOYEE WHERE ID=1");
```

- **Good:**

```
Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(
  "SELECT SALARY FROM EMPLOYEE WHERE ID=1");
```

Problems

1. Write a program that prints the names of all employees and their salaries from a predefined table in MySQL. Don't forget to ensure that all exceptions are handled appropriately and used resources are cleaned.

Write a program that reads a last name and a salary range from the console and prints all matched employees and their salaries from a predefined table in MySQL. Use `PreparedStatement` with parameters. Handle the possible exceptions and close all used resources.

Problems (2)

3. Write a program that creates a table `Countries(country_id, country_name)`. Define a class `Country` with the same fields like the columns in the `Countries` table. Write a method to insert new country. Write a method to list all countries (it should return `List<Country>`). Write a method to find country by `country_id`. Write a method to find country by part of its name. Finally write a method to drop the `Countries` table. Test all methods.

Homework

Write a program that prints the names of all employees, their managers and departments from the standard HR schema in MySQL Handle the possible exceptions and close all used resources.

Write a program that reads a department name from the console and prints all employees in this department and their average salary. Use the standard HR schema in MySQL. Use `PreparedStatement` with parameters. Handle the possible exceptions and close all used resources.

Homework (2)

3. Write a program that creates tables `Users(user_id, user_name, group_id)` and `Groups(group_id, group_name)`. Write classes `User` and `Group` that correspond to these tables. Write methods for adding new users and groups. Write methods for listing all groups, all users and all users by given group. Write methods for updating and deleting users and groups. Finally write a method to drop the tables `Users` and `Groups`. Test all these methods. Handle the exceptions appropriately and close all used resources.