WOLDIA UNIVERSITY
INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTING

DEPARTMENT OF SOFTWARE ENGINEERING
ADVANCED PROGRAMMING

CHAPTER THREE
Java Database Connectivity

Lecture by:
Demeke G.
AY-2017

Outline

- ☞ Introduction to SQL and JDBC
- ☞ Connecting to a Database
- ☞ Manipulating Databases with JDBC
- ☞ PreparedStatements
- ☞ Scrollable and Updateable Result Sets
- ☞ Transaction Processing

Introduction

- JDBC is an API (Application Programming Interface) for Java.
- JDBC provides a standard method for connecting and interacting with relational databases.
- It Enables Java programs to execute SQL queries, update data, and retrieve data from databases.
- A **database** is an organized collection of data.
- A database management system(DBMS) provides mechanisms for storing, organizing, retrieving and modifying data for many users.
- DBMS allow for the access and storage of data With out concern for the internal representation of data.

Cont..

- Some popular **RDBMSs** are: ***Microsoft SQL Server,*** Oracle, Sybase, IBM DB2, **PostgreSQL** and ***MySQL***
- Java programs communicate with databases and manipulate their data using the ***Java Database Connectivity*** (JDBC) API.
- A **JDBC** driver enables Java applications to connect to a database in a particular DBMS and allows to manipulate that database using the **JDBC API**.

SQL Overview

SELECT Query:

- SQL query "**selects**" rows and columns from one or more tables in a database.
- performed by queries with the **SELECT** keyword.
- basic form: ***SELECT * FROM tableName*** asterisk (*) wildcard character indicates that all columns from the **tableName** table should be retrieve.

WHERE Clause:

- it's necessary to locate rows in a database that satisfy certain selection criteria.
- SQL uses the optional **WHERE** clause in a query to specify the selection criteria for the query.
- Basic form: ***SELECT columnName1, columnName2, … FROM tableName WHERE criteria***

cont..

- **ORDER BY Clause:**
  - The rows in the result of a query can be sorted into ascending or descending order by using the optional **ORDER BY** **c**lause.
  - The basic form of a query with an ORDER BY clause is

    SELECT columnName1, columnName2, … FROM tableName ORDER BY column ASC|DESC

- **INNER JOIN:**
  - operator, which merges rows from two tables by matching values in columns that are common to the tables.
  - Basic form: **SELECT columnName1, columnName2, … FROM table1 INNER JOIN table2 ON table1.columnName = table2.columnName**

Cont..

- INSERT Statement:
  - inserts a row into a table.
  - Basic form : **INSERT INTO tableName (columnName1, columnName2, …, columnNameN) VALUES (value1, value2, …, valueN)**
- UPDATE Statement:
  - modifies data in a table
  - Basic form: **UPDATE tableName SET columnName1 = value1, columnName2 = value2,…,columnNameN = valueN WHERE criteria**
- DELETE Statement:
  - removes rows from a table.
  - Its basic form is ***DELETE FROM tableName WHERE criteria***

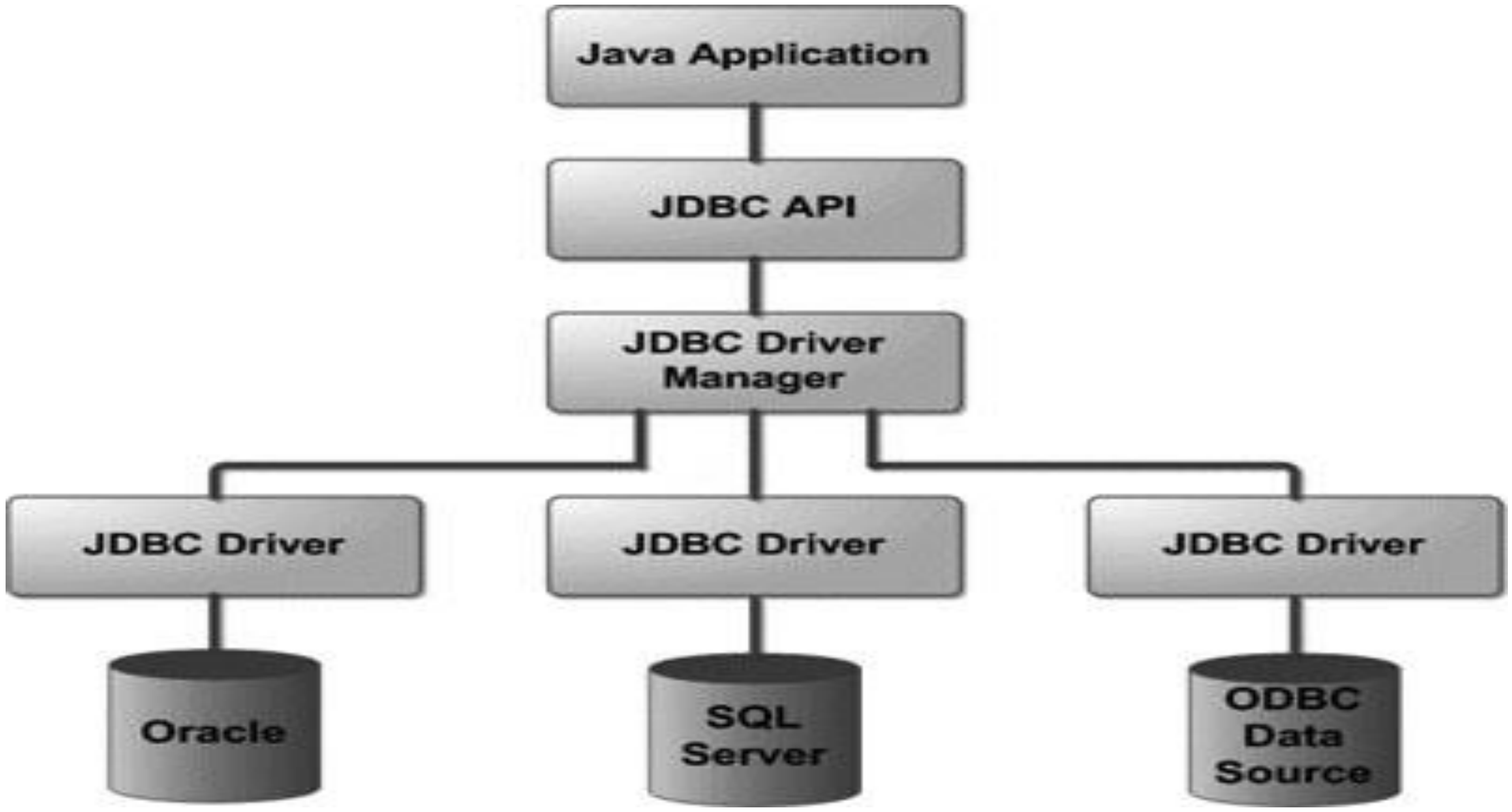
Reading Assignment

- **Between, IN,LIMIT**
- **GROUP BY**
- **Like Clause**
- **Left JOIN and right JOIN**
- **ALTER, DROP, CREATE**
- **DISTINCT statement**
- **AND/OR Clause**
- **IF, CASE and WHILE**
- **COMMIT Statement**
- **ROLLBACK Statement**
- **TRUNCATE TABLE Statement**

Basic JDBC Programming Concepts

- The classes used for JDBC programming are contained in the **java.sql** and **javax.sql** packages.
- JDBC was developed by Sun Microsystems in late 90s
- JDBC provides database independent connectivity between Java Applications and a wide range of relational databases
- Facilitates seamless communication between Java applications and databases.
- Simplifies database operations by providing a set of classes and interfaces
- **In general JDBC Architecture consists of two layers**
  - **JDBC API:** provides the **application-to-JDBC** Manager connection.
  - **JDBC Driver API:** Supports the JDBC Manager-to-Driver Connection.

- The architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application



JDBC API

- A set of interfaces and classes in Java that allow developers to interact with relational databases in a platform-independent manner.
- Provides a high-level abstraction for database operations.
- Contains interfaces like **Connection**, **Statement**, **ResultSet**, **PreparedStatement**, and **CallableStatement**
- uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
- Allows execution of SQL statements and retrieval of query results.
- Manages database transactions (commit/rollback).

JDBC Driver API

- It is the implementation of the JDBC API, provided by database vendors, to handle the low-level communication between a Java application and a specific database.
- Each database (e.g., MySQL, PostgreSQL, Oracle) has its own driver implementation.
- Handles the actual communication with the database
- Implements the JDBC API interfaces, such as Connection and Statement.
- It Handles database-specific communication.
- It is Platform and database-specific.
- Executes SQL queries and returns results to the Java application.

Common JDBC Components

- JDBC API provides the following interfaces and classes

1. DriverManager:

- This class manages a list of database drivers.
- Matches connection requests from the java application with the proper database driver using communication subprotocol.
- The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

Cont..

2. **Connection :** The connection object represents communication context, i.e. all communication with database is through connection object only.

3.Statement : Objects created from this interface is used to submit SQL statements to the database.

- Some derived interfaces accept parameters in addition to executing stored procedures.

4. **ResultSet:** These objects hold data retrieved from a database.

- It acts as an iterator to allow you to move through its data.

5. **SQLException:** handles any errors that occur in a database application

Popular JDBC driver names and database URL

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.cj.jdbc.Driver	jdbc:mysql://hostname/databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName
Postgres	org.postgresql.Driver	jdbc:postgresql://localhost:5432/databaseName
MSSQL Server	com.microsoft.sqlserver.jdbc.SQLServerDriver	jdbc:sqlserver://localhost:1433;databaseName=database

1. Connecting to a Database

- The first step to establishing a connection using **JDBC** involves registering the driver class
- To do that, use the ***forName*** method of the **Class** class, specifying the package and class name of the driver.
    - For example, to register the **MySQL** connector:
      Class.forName(“**com.mysql.cj.jdbc.Driver*****”);***
- Note that the **forName** method throws **ClassNotFoundException**, so you have to enclose this statement in a try/catch block

Cont..

- After you register the driver class, you can call the ***static getConnection*** method of the **DriverManager** class to open the connection.
- This method takes three String parameters: the **database URL**, the **user name**, and a **password**. i.e

```
String url = “jdbc:mysql://localhost/databaseName”;
String user = “root”;
String pw = “pw”;
con = DriverManager.getConnection(url, user, pw);
```

- **java.sql.Connection**
    - Represents a single logical **DB** connection; used for sending SQL statements

2. Creating Statements

- After you connect to a database, you get **a Connection** object.
- The **Connection** class contains the methods for creating SQL statements.
- **Statement** interface represents a SQL statement
- **There are three types of Statement objects**

1. **Simple Statements**

- It represents a simple SQL statement.
- If SQL queries are to be run only once, this Statement is preferred over **PreparedStatement**.

Cont..

- ❖**Connection interface methods for creating** Statement object

public **Statement createStatement**() throws **SQLException**.

- ✓Creates a simple SQL statement.
- ✓The result set of this statement will be **read-only** and **forward scrolling only**.

Statement statement = connection.**createStatement**();

statement.**executeUpdate**("INSERT INTO Employees VALUES (101, 20.00,'Gashaw', 'Alene')");

Cont..

- **public Statement createStatement(int resltSetType, int concurrency) throws SQLException.**
- Creates a simple SQL statement whose result set will have the given properties. The **resultSetType** is either
  - TYPE_FOR-WARD_ONLY,
  - TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE, which are static fields in the java.sql.ResultSet interface.
  - The concurrency type is CONCUR_READ_ONLY or CONCUR_UPDATABLE, for denoting whether the **Resultset** is updatable or not.

Result Set Types

- **TYPE_FORWARD_ONLY:**
  - This result set allows you to move only forward through the data.
  - Once you have retrieved a row of data, you cannot revisit it.
  - The most efficient because it doesn't require caching of data or support for scrolling.
- TYPE_SCROLL_INSENSITIVE:
  - allows to move forward and backward through the data, and it reflects changes made to the data by others after the result set was created.
  - However, it does not reflect changes made by the current application
- TYPE_SCROLL_SENSITIVE:
  - allows to move forward and backward through the data. However, it does reflect changes made by both the current application and others after the result set was created.
  - more resource-intensive and might not be supported by all databases.

Concurrency Modes:

- **CONCUR_READ_ONLY**:
  - the result set is read-only.
  - cannot update the data in the result set using methods like updateRow() or insertRow().
  - This mode is suitable when you only need to fetch data for reading purposes.
- CONCUR_UPDATABLE:
  - the result set is updatable.
  - You can modify the data in the result set using methods like **updateRow**() or **insertRow**().
  - However, not all result sets support updatable concurrency, and it depends on factors such as the database and the SQL query used to generate the result set.

Prepared Statements

- It is an SQL statement that contains **parameters**
- pre-compiled and offer better performance.
- used to execute same SQL statements repeatedly.
- The prepared statement is compiled only once even though it used “n” number of times a prepared SQL statement.
- more secure as they use bind variables to prevent SQL injection attacks.
- Before a prepared statement executed, each parameter needs to be assigned using one of the set methods in the **PreparedStatement** interface.
- A question mark**(?)** is used to denote a parameter

  i.e. **INSERT INTO Employees VALUES (?, ?, ?, ?)**
- **Preparedstatements** are preferred over simple statements for **two good reasons**:
  - execute faster because they are precompiled.
  - easier to code

Cont…

- public **PreparedStatement prepareStater** Preparing statement **l)** throws SQLException. Creates a prepared t.
- public **PreparedStatement prepareStatement(String sql, int resultSetType, int concurrency)** throws SQLException.

PreparedStatement insert = connection.prepareStatement(
"INSERT INTO Employees VALUES (?, ?, ?, ?)");
insert.setDouble(2, 2.50);
insert.setInt(1, 103);
insert.setString(3, "George");
insert.setBoolean(4, true)

Setting the Parameters

int results=insert. **executeUpdate();**

Executing a Prepared Statement

Executing a Prepared Statement

- After the values of all the parameters are set, the prepared statement is executed using one of the following methods in the **PreparedStatement** interface

❖ **public ResultSet executeQuery()** ***throws*** SQLException.
  - Use this method if the SQL statement returns a resultset, like a **SELECT** statement.

❖ **public int executeUpdate()** throws SQLException
  - Use this method for statements like INSERT, UPDATE, or DELETE. The return value is the number of rows affected.

❖ **public boolean execute()** throws SQLException.
  - This method executes any type of SQL statement.
  - Use the **getResultSet**() method to obtain the result set if one is created.

Working with ResultSets

- The SQL statements that read data from a database query return the data in a **result set**.
- The **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.
- Methods of the **ResultSet** interface can be broken down into three categories:
  - **Navigational** methods used to move the cursor around.
  - **Get methods** that are used to view the data in the columns of the current row being pointed to by the cursor.
  - **Update** methods that update the data in the columns of the current row.

Navigating a ResultSet:

❖**The** cursor is movable based on the properties of the **ResultSet**.

❖**Some of methods in the ResultSet interface that involve moving the cursor, including:**

- public void **beforeFirst**() : **Moves the cursor to just before the first row.**
- public void **afterLast**() :Moves the cursor to just after the last row.
- public boolean **first**() .Moves the cursor to the first row.
- public void **last**() . Moves the cursor to the last row.
- public boolean **absolute**(int row) . Moves the cursor to the specified row.
- public boolean **relative**(int row) Moves the cursor the given number of rows forward or backwards from where it currently is pointing.
- public boolean **previous**() . Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
- public boolean **next**() Moves the cursor to the next row. This method returns false if there are no more rows in the result set.

Viewing a ResultSet

- The **ResultSet** interface contains many methods for getting the data of the current row. There is a get method for each of the possible data types.
- Each get method has two versions:
  - that takes in a **column name**, and **column index**.

i.e. if the column you are interested in viewing contains an int, you need to use one of the **getInt()** methods of ResultSet.

public getInt(String columnName) : Returns the **int** in the current row in the column named columnName.

public int getInt(int columnIndex) :Returns the int in the current row in the specified column index.

Metadata

- Data that describes the database or one of its parts is called metadata.
- To find out more about the database, you need to request an object of type **DatabaseMetaData**

```
DatabaseMetaData meta = conn.getMetaData();
```

- The **DatabaseMetaData** class gives data about the database.
- **ResultSetMetaData**, that reports information about a result set. Whenever you have a result set from a query, you can inquire about the number of columns and each column's name, type, and field width.

Cont..

i.e. ResultSet rs = stat.executeQuery("SELECT * FROM " + tableName);

ResultSetMetaData meta = rs.getMetaData();

```
for (int i = 1; i <= meta.getColumnCount(); i++)
{
  String columnName = meta.getColumnLabel(i);
  int columnWidth = meta.getColumnDisplaySize(i);
  Label l = new Label(columnName);
  TextField tf = new TextField(columnWidth);
  }
```

CallableStatement Interface

- To call the **stored procedures and functions**, **CallableStatement** interface is used.
- Stored procedure is a group of SQL queries that are executed as a single logical unit to perform a specific task.
- Name of the procedure should be unique since each procedure is represented by its name.
- We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.
- The **prepareCall**() method of Connection interface returns the instance of **CallableStatement**.

Syntax **public CallableStatement prepareCall**(“

{ call procedurename(?,?...?)}");

i.e. ***CallableStatement*** *stmt=con.prepareCall(*

*"{call **saveStudent**(?,?,?,?)}");*

- It calls the procedure ***saveStudent*** that receives 4 arguments.

Stored Procedure

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `EMP`.`getEmpName`
$$
CREATE PROCEDURE `EMP`.`getEmpName`
 (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END $$
DELIMITER ;
```

Cont..

Three types of parameters exist: IN, OUT, and INOUT.

- The **PreparedStatement** object only uses the **IN** parameter.
- The **CallableStatement** object can use all three.

Parameter	**Description**
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

Cont..

- When you use **OUT** and **INOUT** parameters you must employ an additional CallableStatement method, **registerOutParameter().**
- The **registerOutParameter**() method binds the JDBC data type to the data type the stored procedure is expected to return.
- you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Example

```
import java.sql.*;
public class Proc {
public static void main(String[] args) throws Exception{
Class.forName(" com.mysql.cj.jdbc.Driver ");
Connection con=DriverManager .getConnection
        (“jdbc:mysql://localhost/databaseName“,user,pw);
CallableStatement stmt=con.prepareCall("{call insertR(?,?)}
");
stmt.setInt(1,1011);
stmt.setString(2,"Amit");
stmt.execute();
System.out.println("success");
} }
```

Transaction Processing

- Transaction processing enables a program that interacts with a database to treat a database operation (or set of operations) as a single operation. Such an operation also is known as an **atomic operation** or a **transaction**.
- At the end of a transaction, a decision can be made either to **commit** the transaction or **roll back**.
- **Committing** the transaction finalizes the database operation(s); the transaction cannot be reversed
- **Rolling back** the transaction leaves the database in its state prior to the database operation.

Cont..

- Methods of interface Connection
  - **setAutoCommit** specifies whether each SQL statement commits after it completes (a true argument) or if several SQL statements should be grouped as a transaction (a false argument)
    - If the argument to setAutoCommit is false, the program must follow the last SQL statement in the transaction with a call to Connection method **commit** or **rollback**
  - **getAutoCommi**t determines the autocommit state for the Connection.

Example

```
import java.sql.*;
class FetchRecords{
public static void main(String args[])throws Exception{
Class.forName(" com.mysql.jdbc.Driver ");
Connection con=DriverManager.getConnection
        (“jdbc:mysql://localhost/databaseName“,user,pw);
con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.executeUpdate("insert into users values
(190,'abhi',40000)");
stmt.executeUpdate("insert into users values
(191,'umesh',50000)");
con.commit();
con.close();
}}
```

end

Thank you!!!