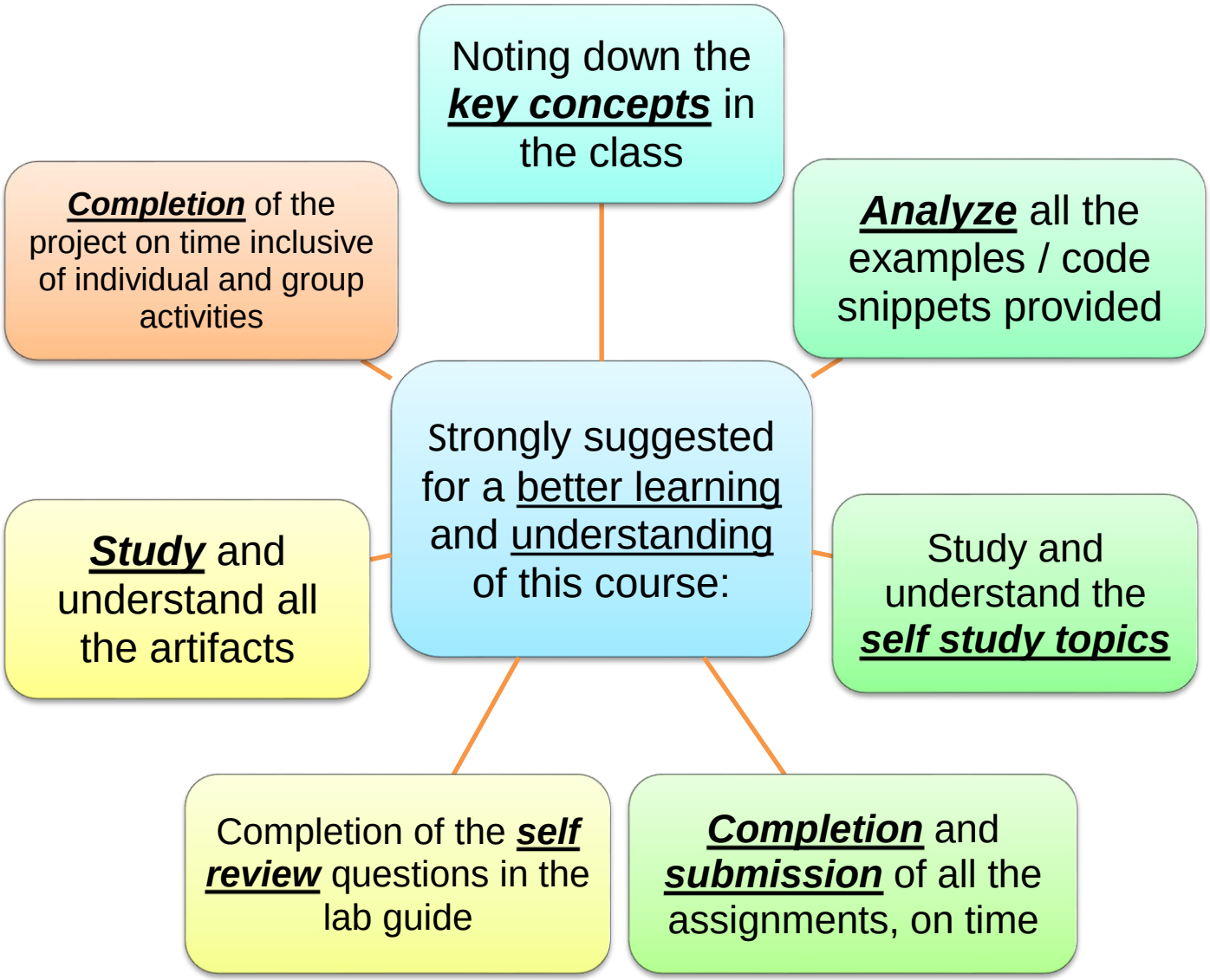
DATABASE PROGRAMMING WITH JDBC

Instructor:

Table of contents

Learning Approach

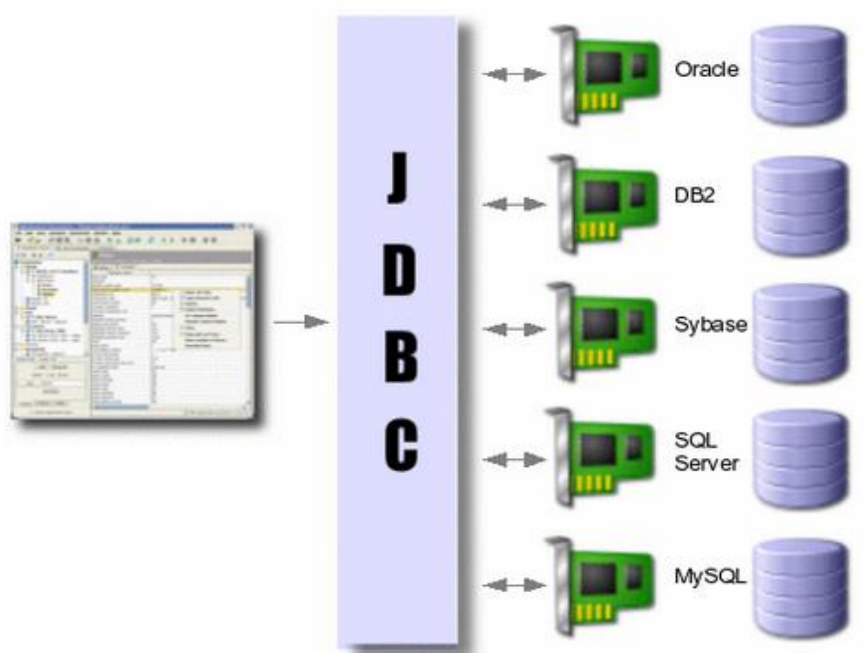


Section 1

JDBC DRIVERS TYPES

JDBC OVERVIEW

- JDBC (Java Database Connectivity) API allows Java programs to connect to databases
- 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.



JDBC Driver

- A **JDBC driver** is a **set of Java classes** that implement the JDBC interfaces,
  - ✓ targeting a specific database.
- The JDBC interfaces comes with **standard Java,**
  - ✓ but the implementation of these interfaces is specific **to the database** you need to connect to. Such an implementation is called a JDBC driver.
- There are **4 different types of JDBC drivers:**
  - ✓ Type 1: JDBC-ODBC bridge driver
  - ✓ Type 2: Java + Native code driver
  - ✓ Type 3: All Java + Middleware translation driver
  - ✓ Type 4: All Java driver.

Section 2

WORKING STEPS

Working steps

1. Create connection
2. Create access statement
3. Run access statement
4. Retrieve data
5. Close connection

Create Connection

- **Load the database driver**

```
Class.forName
("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

- **Obtain a connection:**

```
String connectionUrl = "jdbc:sqlserver://1FWADIEUNT1-
LT:1433;databaseName=Fsoft_Training";

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

Create Access Statement

- Use for general-purpose access to your database.
- Useful when you are using static SQL statements at runtime.
- The **Statement** interface cannot accept parameters.
- **Syntax:**

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

Create Access Statement (2)

```
// for use with ResultSet only
// No "previous" method using, no update
connection.createStatement(
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_READ_ONLY);

// with "previous" method using, update
connection.createStatement(
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
```

Run Access Statement

- Statement's **methods**:
  - ✓ **boolean execute(String SQL)** : may be any kind of SQL statement. Returns a boolean value of true if a ResultSet object can be retrieved; false if the first result is an update count or there is no result.
  - ✓ **int executeUpdate(String SQL)** : Returns the numbers 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.

Examples

- **Example 1:**

```
// Create and execute an SQL statement that returns some data.
String SQL1 = "SELECT TOP 10 * FROM Person";
stmt=conn.createStatement();//ResultSet.TYPE_SCROLL_SENSITIVE,Re
  sultSet.CONCUR_UPDATABLE
rs = stmt.executeQuery(SQL);
```

- **Example 2:**

```
// Create and execute an SQL statement that returns some data.
String SQL2 = “INSERT INTO STOCK(STOCK_CODE, STOCK_NAME)
          VALUES(‘11’, ‘STOCK1’)";
stmt = conn.createStatement();
int no_of_row = stmt.executeUpdate(SQL);
```

Retrieve Data & Close Connection

- **Retrieve data**

```
// Iterate through the data in the result set and display it.

while (rs.next()) {

   System.out.println(rs.getInt(1) + " " +
    rs.getString(2)+" "+rs.getInt(3));

}
```

- **Close connection**

```
conn.close();
```

Section 3

JDBC RESULTSET

JDBC Resultset

- **Type of ResultSet:** The possible Type are given below, If you do not specify any ResultSet type, you will automatically get one that is **TYPE_FORWARD_ONLY.**

Type	Description
ResultSet. TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet. TYPE_SCROLL_INSENSITIVE	The cursor can scroll forwards and backwards, 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 forwards and backwards, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

JDBC Resultset (cont.)

- **Concurrency of ResultSet:** The possible RSConcurrency are given below, If you do not specify any Concurrency type, you will automatically get one that is **CONCUR_READ_ONLY**.

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

JDBC Resultset (cont.)

- **ResultSet methods:**

.N.	Methods & Description
1	**public void beforeFirst() throws SQLException** Moves the cursor to just before the first row
2	**public void afterLast() throws SQLException** Moves the cursor to just after the last row
3	**public boolean first() throws SQLException** Moves the cursor to the first row
4	**public void last() throws SQLException** Moves the cursor to the last row.
5	**public boolean absolute(int row) throws SQLException** Moves the cursor to the specified row
6	**public boolean relative(int row) throws SQLException** Moves the cursor the given number of rows forward or backwards from where it currently is pointing.

JDBC Resultset (cont.)

- **ResultSet methods:**

.N.	Methods & Description
7	**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
8	**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
9	**public int getRow() throws SQLException** Returns the row number that the cursor is pointing to.
10	**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.
11	**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

JDBC Resultset (cont.)

- **Viewing a Result Set:**

S. N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.
3	**public XXX getXXX(int columnIndex) throws SQLException**

- **Example:** reference **InsertData.java**

JDBC Resultset (cont.)

- The ResultSet interface contains a collection of update methods for updating the data of a result set.
- As with the get methods, there are two update methods for each data type:
  - ✓ One that takes in a column name.
  - ✓ One that takes in a column index.
- **For example:**

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of s.
2	**public void updateString(String columnName, String s) throws SQLException** Similar to the previous method, except that the column is specified by its name instead of its index.

JDBC Update using ResultSet

```
ResultSet rs = statement.executeQuery(query);
…
// for update
rs.updateBoolean(1, false); // change the first column
rs.updateInt("Age", 25); // change the column named "Age"
rs.updateRow();

// to delete
rs.deleteRow();
```

Section 4

JDBC WITH PARAMETER

JDBC With Parameter

- **PreparedStatement** Objects:
  - ✓ The PreparedStatement interface extends the Statement interface which gives you added functionality with a couple of advantages over a generic Statement object.
- This statement gives you the flexibility of supplying **arguments dynamically.**

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

}
finally {}
```

JDBC With Parameter (cont.)

- The **setXXX()** methods bind values to the parameters.

```
pstmt.setInt(1,23);
pstmt.setString(2,"Roshan");
pstmt.setString(3,"CEO");
pstmt.executeUpdate();
```

JDBC With Parameter

```
// in string query
String query = "INSERT INTO Person " +
               "VALUES (" + <name> + ", "
                          <age> + … + ")";
// using statementPrepare
String query = "INSERT INTO Person " +
               "VALUES (?, ?)"
PreparedStatement statement = connect.prepareStatement(query);
connect.setAutoCommit(false);
statement.setString(1, "Titi");
statement.setInt(2, 25);
statement.executeQuery();          // insert 1
statement.setString(1, "Tata");
statement.setInt(2, 28);
statement.executeQuery();          // Insert 2
connect.commit();
connect.setAutoCommit(true);
```

Section 5

JDBC “BATCH” STATEMENT

JDBC “batch” processing

```
connect.setAutoCommit(false);
// replace executeQuery by addBatch
…
statement.setString(1, “Titi”);
statement.setInt(2, 25);
statement.addBatch();          // Insert 1
statement.setString(1, “Tata”);
statement.setInt(2, 28);
statement.addBatch();          // Insert 2
// then call batch processing statement
statement.executeBatch();
// also applied for normal statement (not prepared one)
connect.commit();
connect.setAutoCommit(true);
```

JDBC Batch with String Query

- **Step 1:**

```
connect.setAutoCommit(false);
```

- **Step 2:**

```
Statement statement = connect.createStatement();
statement.addBatch(<Insert query>);
statement.addBatch(<Insert query>);
statement.addBatch(<Update query>);
statement.addBatch(<Delete query>);
```

- **Step 3:**

```
int[] updateCounts = statement.executeBatch();
connect.commit();
statement.close();
connect.setAutoCommit(true);
```

Call Stored Procedure

```
public static void executeStoredProcedure(Connection con) {
   try {
          CallableStatement cstmt =
          con.prepareCall("{call dbo.GetImmediateManager(?, ?)}");

          cstmt.setInt(1, 5);
          cstmt.registerOutParameter(2, java.sql.Types.INTEGER);
          cstmt.execute();
      System.out.println("MANAGER ID: " + cstmt.getInt(2));
      } catch (Exception e) {
           e.printStackTrace();
      }
      }
```

Lesson Summary

- JDBC Drivers
- Working steps
- JDBC resultset
- JDBC with parameter
- JDBC “Batch” statement

Thank you