Lecture 1: JDBC Introduction

What is JDBC?

JDBC (Java Database Connectivity) is a Java API that enables Java applications to interact with databases. It provides a standard interface for connecting to and working with relational databases.

What is Data?

Data is information that can be stored, processed, and retrieved. In software applications, data represents the core information that needs to be persisted beyond the application's runtime.

Where Should Data Be Stored?

Text Files - Problems:

- **No Structure**: Difficult to organize and search
- **No Data Integrity:** No validation or constraints
- **Poor Performance**: Linear search required
- **Concurrency Issues**: Multiple users can't access simultaneously
- **No Security**: No access control mechanisms

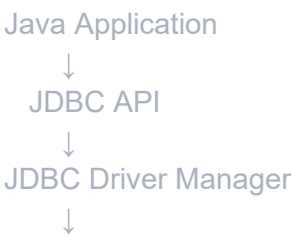
RDBMS (Relational Database Management System) - Solution:

- **Structured Storage**: Tables, rows, columns
- **Data Integrity**: Constraints, validation rules
- **Efficient Queries**: SQL for complex operations
- **Concurrency Control**: Multiple users can access safely
- **Security Features**: User authentication and authorization
- **ACID Properties**: Atomicity, Consistency, Isolation, Durability

Why JDBC?

- **Database Independence**: Same code works with different databases
- **Standardized API**: Consistent interface across database vendors
- **Flexibility**: Easy to switch between databases
- **Supported Databases**: PostgreSQL, MySQL, Oracle, H2, SQL Server, etc.

JDBC Architecture:



TELUSKO

Database-Specific Driver

↓

Database

TELUSKO

Lecture 2: PostgreSQL Setup

Complete PostgreSQL Installation Steps

Step 1: Download PostgreSQL

1. Visit postgresql.org
2. Click "Download" → Select your operating system
3. Download the installer for your platform

Step 2: Installation Process

Windows:

1. Run the downloaded .exe file as administrator
2. Follow the installation wizard:
    - Choose installation directory (default: `C:\Program Files\PostgreSQL\15`)
    - Select components (PostgreSQL Server, pgAdmin 4, Command Line Tools)
    - Choose data directory (default: `C:\Program Files\PostgreSQL\15\data`)
    - Set password for postgres superuser (remember this!)
    - Set port (default: 5432)
    - Choose local (default is fine)
3. Complete installation

macOS:

1. Run the downloaded .dmg file
2. Follow similar steps as Windows
3. Or use Homebrew: `brew install postgresql`

Linux (Ubuntu/Debian):

```
sudo apt update
sudo apt install postgresql postgresql-contrib
sudo systemctl start postgresql
sudo systemctl enable postgresql
```

Step 3: Verify Installation

1. Open Command Prompt/Terminal
2. Test connection: `psql -U postgres`
3. Enter the password you set during installation

Step 4: Using pgAdmin

TELUSKO

1. Launch pgAdmin 4 from Start Menu/Applications
2. Create master password for pgAdmin
3. Connect to the PostgreSQL server:
    - Host: localhost
    - Port: 5432
    - Username: postgres
    - Password: (your set password)

Step 5: Create Database and Table

1. In pgAdmin, right-click "Databases" → "Create" → "Database"
2. Name: `demo`
3. Click "Save"

Create Table using SQL:

```
-- Connect to demo database
-- Create student table
CREATE TABLE student (
    sid INTEGER PRIMARY KEY,
    sname TEXT NOT NULL,
    marks INTEGER
);

-- Insert sample data
INSERT INTO student VALUES (1, 'Navin', 80);
```

TELUSKO

Lecture 3: JDBC Setup – The 7 Steps

The Seven Essential Steps to Connect with JDBC:

1. **Import Package**: Import necessary JDBC classes
2. **Load Driver**: Load the database driver class
3. **Register Driver**: Register driver with DriverManager
4. **Create Connection**: Establish connection to database
5. **Create Statement**: Create statement object for SQL execution
6. **Execute Statement**: Execute SQL queries/updates
7. **Close Connection**: Release database resources

Detailed Explanation:

Step 1: Import Package

```
import java.sql.*;
```

This imports all JDBC-related classes including Connection, Statement, ResultSet, etc.

Steps 2 & 3: Load and Register Driver

```
Class.forName("org.postgresql.Driver");
```

This loads the PostgreSQL driver class and automatically registers it with DriverManager.

Step 4: Create Connection

```
Connection conn = DriverManager.getConnection(url, username, password);
```

Establishes actual connection to the database.

Step 5: Create Statement

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

Creates a statement object to execute SQL commands.

Step 6: Execute Statement

- `executeQuery()`: For SELECT statements (returns ResultSet)
- `executeUpdate()`: For INSERT, UPDATE, DELETE (returns int)
- `execute()`: For any SQL statement (returns boolean)

Step 7: Close Connection

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

Always close connections to free up database resources.

TELUSKO

TELUSKO

Lecture 4: PostgreSQL JDBC Driver Setup

Complete IntelliJ IDEA Setup:

Step 1: Create New Project

1. Open IntelliJ IDEA
2. File → New → Project
3. Choose "Java" → Next
4. Project name: `JDBCCourse`
5. Choose project location and JDK version
6. Finish

Step 2: Download PostgreSQL JDBC Driver

1. Visit jdbc.postgresql.org
2. Download latest JDBC driver (.jar file)
3. Save to a known location (e.g., `libs` folder in the project).

Step 3: Add Driver to Project

Method 1: IntelliJ IDEA

1. File → Project Structure (Ctrl+Alt+Shift+S)
2. Select "Modules" → Dependencies tab
3. Click "+" → "JARs or directories"
4. Navigate to downloaded postgresql-xx.xx.jar file
5. Select and click OK
6. Apply and OK

Method 2: Manual Library Addition

1. Create `libs` folder in project root
2. Copy `postgresql-xx.xx.jar` to libs folder
3. Right-click jar file → "Add as Library"

Step 4: Verify Setup

Create a test class to verify the driver is accessible:

```
public class DriverTest {
    public static void main(String[] args) {
        try {
            Class.forName("org.postgresql.Driver");
            System.out.println("PostgreSQL Driver loaded successfully!");
        } catch (ClassNotFoundException e) {
            System.out.println("Driver not found: " + e.getMessage());
        }
```

TELUSKO

```
    }
}
```

TELUSKO

Lecture 5: Connecting Java and Database

Complete Connection Code:

```
import java.sql.*;

public class DemoJdbc {
    public static void main(String[] args) {
        // Database connection parameters
        String url = "jdbc:postgresql://localhost:5432/demo";
        String username = "postgres";
        String password = "0000"; // Your PostgreSQL password

        Connection conn = null;

        try {
            // Step 1: Load and register driver
            Class.forName("org.postgresql.Driver");
            System.out.println("Driver loaded successfully");

            // Step 2: Create connection
            conn = DriverManager.getConnection(url, username, password);
            System.out.println("Connection established successfully");

        } catch (ClassNotFoundException e) {
            System.out.println("Driver not found: " + e.getMessage());
        } catch (SQLException e) {
            System.out.println("Connection failed: " + e.getMessage());
        } finally {
            // Step 3: Close connection
            try {
                if (conn != null && !conn.isClosed()) {
                    conn.close();
                    System.out.println("Connection closed");
                }
            } catch (SQLException e) {
                System.out.println("Error closing connection: " + e.getMessage());
            }
        }
    }
}
```

Connection URL Format:

```
jdbc:postgresql://hostname:port/database_name
```

- **hostname**: Server address (localhost for local machine)

TELUSKO

- **port**: PostgreSQL port (default: 5432)
- **database_name**: Name of your database

Common Connection Issues:

1. **Driver not found:**Ensure JDBC driver is in the classpath.
2. **Connection refused:**Check if PostgreSQL service is running
3. **Authentication failed:**Verify username/password
4. **Database not found:**Ensure database exists

TELUSKO

Lecture 6: Execute and Process Queries

Complete Query Execution Code:

```
import java.sql.*;

public class DemoJdbc {
    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost:5432/demo";
        String username = "postgres";
        String password = "0000";

        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        try {
            // Load driver and create connection
            Class.forName("org.postgresql.Driver");
            conn = DriverManager.getConnection(url, username, password);
            System.out.println("Connection established");

            // SQL query
            String query = "SELECT sname FROM student WHERE sid = 1";

            // Create statement
            stmt = conn.createStatement();

            // Execute query
            rs = stmt.executeQuery(query);

            // Process result
            if (rs.next()) {
                String studentName = rs.getString("sname");
                System.out.println("Student name: " + studentName);
            } else {
                System.out.println("No student found with sid = 1");
            }

        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Close resources in reverse order
            try {
                if (rs != null) rs.close();
                if (stmt != null) stmt.close();
```

TELUSKO

```
            if (conn != null) conn.close();
            System.out.println("Connection closed");
        } catch (SQLException e) {
            System.out.println("Error closing resources: " + e.getMessage());
        }
    }
}
}
```

Understanding ResultSet:

- **rs.next()**: Moves cursor to next row, returns true if row exists
- **rs.getString("column_name")**: Gets string value from named column
- **rs.getInt("column_name")**: Gets integer value from named column
- **rs.getString(1)**: Gets string value from first column (1-indexed)

TELUSKO

Lecture 7: Fetching All Records

Setup Additional Test Data:

```
INSERT INTO student VALUES (2, 'Kiran', 50);
INSERT INTO student VALUES (3, 'Harsh', 60);
INSERT INTO student VALUES (4, 'Sushil', 40);
```

Complete Code to Fetch All Records:

```
import java.sql.*;

public class FetchAllRecords {
    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost:5432/demo";
        String username = "postgres";
        String password = "0000";

        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;

        try {
            // Establish connection
            Class.forName("org.postgresql.Driver");
            conn = DriverManager.getConnection(url, username, password);
            System.out.println("Connection established");

            // Query to fetch all records
            String query = "SELECT * FROM student";

            // Create statement and execute query
            stmt = conn.createStatement();
            rs = stmt.executeQuery(query);

            // Display headers
            System.out.println("SID\tName\t\tMarks");
            System.out.println("---\t----\t\t-----");

            // Process all rows
            while (rs.next()) {
                int sid = rs.getInt(1);          // or rs.getInt("sid")
                String sname = rs.getString(2);   // or rs.getString("sname")
                int marks = rs.getInt(3);         // or rs.getInt("marks")

                System.out.println(sid + "\t" + sname + "\t\t" + marks);
            }
```

TELUSKO

```
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Close resources
            try {
                if (rs != null) rs.close();
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
                System.out.println("Connection closed");
            } catch (SQLException e) {
                System.out.println("Error closing resources: " + e.getMessage());
            }
        }
    }
}
```

ResultSet Navigation Methods:

- **rs.next()**: Move to next row
- **rs.previous()**: Move to previous row
- **rs.first()**: Move to first row
- **rs.last()**: Move to last row
- **rs.absolute(n)**: Move to nth row

TELUSKO

Lecture 8: CRUD Operations

CRUD Overview:

- **C**reate: INSERT operations
- **R**ead: SELECT operations
- **U**pdate: UPDATE operations
- **D**elete: DELETE operations

INSERT Operation:

```
import java.sql.*;

public class InsertOperation {
    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost:5432/demo";
        String username = "postgres";
        String password = "0000";

        Connection conn = null;
        Statement stmt = null;

        try {
            Class.forName("org.postgresql.Driver");
            conn = DriverManager.getConnection(url, username, password);

            // INSERT query
            String sql = "INSERT INTO student VALUES (5, 'Shiva', 70)";

            stmt = conn.createStatement();

            // Execute insert - returns boolean
            boolean status = stmt.execute(sql);
            System.out.println("Execute status: " + status); // false for
INSERT/UPDATE/DELETE

            // Alternative: Use executeUpdate() for INSERT/UPDATE/DELETE
            // int rowsAffected = stmt.executeUpdate(sql);
            // System.out.println("Rows affected: " + rowsAffected);

            System.out.println("Record inserted successfully");

        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            try {
                if (stmt != null) stmt.close();
```

TELUSKO

```
            if (conn != null) conn.close();
        } catch (SQLException e) {
            System.out.println("Error closing resources: " + e.getMessage());
        }
    }
  }
}
```

UPDATE Operation:

```
// UPDATE query
String sql = "UPDATE student SET sname = 'Arjun' WHERE sid = 5";
int rowsUpdated = stmt.executeUpdate(sql);

System.out.println("Rows updated: " + rowsUpdated);
```

DELETE Operation:

```
// DELETE query
String sql = "DELETE FROM student WHERE sid = 5";
int rowsDeleted = stmt.executeUpdate(sql);

System.out.println("Rows deleted: " + rowsDeleted);
```

Understanding the execute() Method:

- Returns **true** if result is a ResultSet (SELECT queries)
- Returns **false** if result is an update count (INSERT/UPDATE/DELETE)
- Use **executeQuery()** for SELECT statements
- Use **executeUpdate()** for INSERT/UPDATE/DELETE statements

TELUSKO

Lecture 9: Problems with Statement

Dynamic Values Problem:

```
// Variables with dynamic values
int sid = 101;
String sname = "Max";
int marks = 48;

// This WON'T work - variables not substituted
String sql = "INSERT INTO student VALUES (sid, sname, marks)";

// This works but is problematic
String sql = "INSERT INTO student VALUES (" + sid + ", '" + sname + "', " + marks + ")";
```

Major Problems with Statement:

1. SQL Injection Vulnerability:

```
// If sname comes from user input: "Max'; DROP TABLE student; --"
String sname = "Max'; DROP TABLE student; --";
String sql = "INSERT INTO student VALUES (" + sid + ", '" + sname + "', " + marks + ")";
// Results in: INSERT INTO student VALUES (101, 'Max'; DROP TABLE student; --', 48)

// This could delete your entire table!
```

2. String Concatenation Complexity:

```
// Complex and error-prone
String sql = "INSERT INTO student VALUES (" + sid + ", '" + sname + "', " + marks + ")";

// Must handle quotes, escaping, data types manually
```

3. Performance Issues:

- SQL is compiled every time
- No query plan reuse
- Database has to parse SQL each execution

4. Data Type Handling:

```
// Date/Time values are complex to handle
Date joinDate = new Date();
String sql = "INSERT INTO employee VALUES (" + id + ", '" + name + "', '" + joinDate + "')";

// Date formatting issues across databases
```

TELUSKO

Lecture 10: Prepared Statement Solution

What is PreparedStatement?

PreparedStatement is a precompiled SQL statement that:

- Prevents SQL injection attacks
- Improves performance through query plan reuse
- Provides cleaner, more readable code
- Handles data type conversions automatically

Complete PreparedStatement Example:

```
import java.sql.*;

public class PreparedStatementDemo {
    public static void main(String[] args) {
        String url = "jdbc:postgresql://localhost:5432/demo";
        String username = "postgres";
        String password = "0000";

        Connection conn = null;
        PreparedStatement pstmt = null;

        try {
            Class.forName("org.postgresql.Driver");
            conn = DriverManager.getConnection(url, username, password);

            // SQL with placeholders (?)
            String sql = "INSERT INTO student VALUES (?, ?, ?)";

            // Create PreparedStatement
            pstmt = conn.prepareStatement(sql);

            // Set parameters (1-indexed)
            pstmt.setInt(1, 10);          // sid
            pstmt.setString(2, "Ananya");  // sname
            pstmt.setInt(3, 90);          // marks

            // Execute the statement
            int rowsAffected = pstmt.executeUpdate();
            System.out.println("Rows inserted: " + rowsAffected);

        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            e.printStackTrace();
        } finally {
```

TELUSKO

```
            try {
                if (pstmt != null) pstmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
                System.out.println("Error closing resources: " + e.getMessage());
            }
        }
    }
}
```

Understanding Placeholders (?):

- **?** represents a parameter placeholder
- Parameters are 1-indexed (first ? is index 1)
- Use appropriate setter methods based on data type:
    - `setInt(index, value)` for integers
    - `setString(index, value)` for strings
    - `setDouble(index, value)` for doubles
    - `setDate(index, value)` for dates
    - `setBoolean(index, value)` for booleans

PreparedStatement vs Statement Comparison:

Aspect	Statement	PreparedStatement
SQL Injection	Vulnerable	Safe
Performance	Slower (compiled each time)	Faster (precompiled)
Code Readability	Complex concatenation	Clean and simple
Parameter Handling	Manual string manipulation	Automatic type handling
Reusability	Limited	High (can reuse with different parameters)

Complete CRUD with PreparedStatement:

```
public class PreparedStatementCRUD {
    private static final String URL = "jdbc:postgresql://localhost:5432/demo";
    private static final String USERNAME = "postgres";
    private static final String PASSWORD = "0000";
```

TELUSKO

```
    // CREATE
    public static void insertStudent(int sid, String sname, int marks) {
        String sql = "INSERT INTO student VALUES (?, ?, ?)";
        try (Connection conn = DriverManager.getConnection(URL, USERNAME,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, sid);
            pstmt.setString(2, sname);
            pstmt.setInt(3, marks);

            int rows = pstmt.executeUpdate();
            System.out.println("Inserted " + rows + " record(s)");

        } catch (SQLException e) {
            System.out.println("Insert error: " + e.getMessage());
        }
    }

    // READ
    public static void getStudent(int sid) {
        String sql = "SELECT * FROM student WHERE sid = ?";
        try (Connection conn = DriverManager.getConnection(URL, USERNAME,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, sid);
            ResultSet rs = pstmt.executeQuery();

            if (rs.next()) {
                System.out.println("ID: " + rs.getInt("sid") +
                                   ", Name: " + rs.getString("sname") +
                                   ", Marks: " + rs.getInt("marks"));
            } else {
                System.out.println("Student not found");
            }

        } catch (SQLException e) {
            System.out.println("Select error: " + e.getMessage());
        }
    }

    // UPDATE
    public static void updateStudent(int sid, String newName, int newMarks) {
        String sql = "UPDATE student SET sname = ?, marks = ? WHERE sid = ?";
        try (Connection conn = DriverManager.getConnection(URL, USERNAME,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
```

TELUSKO

```
            pstmt.setString(1, newName);
            pstmt.setInt(2, newMarks);
            pstmt.setInt(3, sid);

            int rows = pstmt.executeUpdate();
            System.out.println("Updated " + rows + " record(s)");

        } catch (SQLException e) {
            System.out.println("Update error: " + e.getMessage());
        }
    }

    // DELETE
    public static void deleteStudent(int sid) {
        String sql = "DELETE FROM student WHERE sid = ?";
        try (Connection conn = DriverManager.getConnection(URL, USERNAME,
PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, sid);

            int rows = pstmt.executeUpdate();
            System.out.println("Deleted " + rows + " record(s)");

        } catch (SQLException e) {
            System.out.println("Delete error: " + e.getMessage());
        }
    }
}
```

TELUSKO

Summary

This course covered the fundamentals of JDBC programming:

1. **Introduction**: Understanding the need for JDBC and database connectivity
2. **Setup**: Installing PostgreSQL and configuring the development environment
3. **Connection**: The 7-step process to connect Java with databases
4. **Basic Queries**: Executing SELECT statements and processing results
5. **CRUD Operations**: Complete Create, Read, Update, Delete operations
6. **Security**: Understanding SQL injection vulnerabilities
7. **Best Practices**: Using PreparedStatement for safe and efficient database operations

Key Takeaways:

- Always use PreparedStatement for user input
- Properly close database resources
- Handle exceptions appropriately
- Use try-with-resources for automatic resource management
- Follow the 7-step JDBC process consistently

Next Steps:

- Learn about Connection Pooling
- Explore Transaction Management
- Study Database Metadata
- Practice with different database systems
- Learn about ORM frameworks (Hibernate, JPA)