Introduction to Java and Core OOP Concepts

Dr. Ratnesh Prasad Srivastava

Department of CSIT, GGV, Bilaspur (C.G.)

Academic Year: 2026-27

Course Information

Course Code	CIUDMJT1
Course Title	Object-Oriented Programming with Java
Credit Hours	3-0-3 (3 Lecture, 0 Tutorial, 3 Practical)
Prerequisites	Programming Fundamentals
Textbook	”Java: The Complete Reference” by Herbert Schildt
Reference	”Head First Java” by Kathy Sierra and Bert Bates

Contents

1 Unit V: Java Database Connectivity (JDBC) and Java 8+ Features

1.1 Learning Objectives

- Understand JDBC architecture and components
- Master the steps to connect to a database using JDBC
- Work with DriverManager, Connection, Statement, and ResultSet
- Understand and implement CRUD operations
- Learn PreparedStatement for parameterized queries
- Understand Lambda Expressions and Functional Interfaces
- Master Stream API for data processing
- Apply Java 8+ features in practical scenarios

2 Java Database Connectivity (JDBC) Basics

2.1 JDBC Architecture

JDBC Definition

JDBC (Java Database Connectivity) is a Java API that enables Java applications to interact with databases. It provides a standard interface for connecting to relational databases and executing SQL statements, retrieving results, and managing transactions.

```
/* *******************************************************************************
 * PROGRAM: JDBCArchitecture.java
 *
 * PURPOSE: This program explains the complete JDBC architecture and components.
 *          It provides a conceptual understanding of how Java applications
 *          communicate with databases through different layers.
 *
 * PROBLEM SOLVED: Students often struggle to understand how JDBC works behind
 *                 the scenes. This program visualizes the layered architecture
 *                 and explains each component's role in database connectivity.
 *
 * WHY NECESSARY: Understanding JDBC architecture is fundamental before writing
 *                database code. It helps developers:
 *                1. Troubleshoot connection issues
```

```java
 *                 2. Choose appropriate driver types
 *                 3. Understand performance implications
 *                 4. Make informed decisions about database connectivity
 *
 * KEY CONCEPTS DEMONSTRATED:
 * - 4-layer JDBC architecture
 * - Different types of JDBC drivers
 * - Role of each JDBC component
 * - Java application to database communication flow
 ******************************************************************************/

public class JDBCArchitecture {

    // ==================== JDBC ARCHITECTURE EXPLANATION ====================
    public static void explainArchitecture() {
        System.out.println("=== JDBC ARCHITECTURE ===");
        System.out.println();
        System.out.println("1. Java Application Layer");
        System.out.println("   - Your Java program using JDBC API");
        System.out.println("   - Uses interfaces: Connection, Statement, ResultSet");

        System.out.println("\n2. JDBC API Layer");
        System.out.println("   - java.sql and javax.sql packages");
        System.out.println("   - Provides standard interfaces");
        System.out.println("   - DriverManager: Manages database drivers");

        System.out.println("\n3. JDBC Driver Layer");
        System.out.println("   - Database-specific implementations");
        System.out.println("   - Types: Type 1, 2, 3, 4");
        System.out.println("   - Converts JDBC calls to database-specific calls");

        System.out.println("\n4. Database Layer");
        System.out.println("   - Actual RDBMS (MySQL, Oracle, PostgreSQL, etc.)");
        System.out.println("   - Stores and manages data");

        System.out.println("\n=== JDBC DRIVER TYPES ===");
        System.out.println("Type 1: JDBC-ODBC Bridge Driver (Deprecated)");
        System.out.println("Type 2: Native-API Driver (Part Java, Part Native)");
        System.out.println("Type 3: Network Protocol Driver (Pure Java)");
        System.out.println("Type 4: Thin Driver (Pure Java, Direct) - Most Common");
    }

    // ==================== JDBC COMPONENTS ====================
    public static void explainComponents() {
        System.out.println("\n=== JDBC CORE COMPONENTS ===");

        System.out.println("\n1. DriverManager:");
        System.out.println("   - Manages database drivers");
        System.out.println("   - Establishes database connections");
        System.out.println("   - Methods: getConnection(), registerDriver()");

        System.out.println("\n2. Connection:");
        System.out.println("   - Represents a connection to database");
        System.out.println("   - Creates Statement objects");
        System.out.println("   - Manages transactions");
        System.out.println("   - Methods: createStatement(), prepareStatement()");

        System.out.println("\n3. Statement:");
        System.out.println("   - Executes SQL queries");
        System.out.println("   - Types: Statement, PreparedStatement, CallableStatement");
        System.out.println("   - Methods: executeQuery(), executeUpdate()");

        System.out.println("\n4. ResultSet:");
        System.out.println("   - Contains query results");
        System.out.println("   - Navigable cursor through rows");
        System.out.println("   - Methods: next(), getString(), getInt()");

        System.out.println("\n5. SQLException:");
        System.out.println("   - Checked exception for database errors");
        System.out.println("   - Provides error codes and messages");
    }

    // ==================== JDBC WORKFLOW ====================
    public static void explainWorkflow() {
        System.out.println("\n=== JDBC WORKFLOW ===");

        System.out.println("Step 1: Load and Register Driver");
        System.out.println("   Class.forName(\"com.mysql.cj.jdbc.Driver\");");

        System.out.println("\nStep 2: Establish Connection");
        System.out.println("   Connection conn = DriverManager.getConnection(url, user, pass);");

        System.out.println("\nStep 3: Create Statement");
        System.out.println("   Statement stmt = conn.createStatement();");

        System.out.println("\nStep 4: Execute Query");
        System.out.println("   ResultSet rs = stmt.executeQuery(\"SELECT * FROM table\");");

        System.out.println("\nStep 5: Process Results");
        System.out.println("   while(rs.next()) { /* process each row */ }");

        System.out.println("\nStep 6: Close Resources");
        System.out.println("   rs.close(); stmt.close(); conn.close();");
    }

    public static void main(String[] args) {
        System.out.println("=== JDBC ARCHITECTURE AND COMPONENTS ===\n");

        explainArchitecture();
        explainComponents();
        explainWorkflow();
    }
}
```

Listing 1: JDBC Architecture Overview Program

2.2 Steps to Connect to a Database

```java
/* ********************************************************************************
 * PROGRAM: JDBCConnectionSteps.java
 *
 * PURPOSE: This program demonstrates all 6 steps required to connect Java
 *          applications to databases using JDBC. It provides hands-on examples
 *          of each step with multiple implementation approaches.
 *
 * PROBLEM SOLVED: Beginners often get confused about the proper sequence and
 *                 implementation of database connectivity. This program solves:
 *                 1. Clear step-by-step breakdown
 *                 2. Multiple ways to accomplish each step
 *                 3. Proper resource management
 *                 4. Error handling best practices
 *
 * WHY NECESSARY: Database connectivity is a core skill in enterprise Java
 *                development. This program is necessary because:
 *                1. Most applications need database persistence
 *                2. Proper connection management prevents resource leaks
 *                3. Understanding different approaches helps in different scenarios
 *                4. Industry requires standardized database access patterns
 *
 * KEY CONCEPTS DEMONSTRATED:
 * - 6 essential JDBC steps
 * - Driver loading (manual vs auto)
 * - Multiple connection establishment methods
 * - Different Statement types and their use cases
 * - Proper ResultSet processing
 * - Resource cleanup (traditional vs try-with-resources)
 * - Complete CRUD operations
 * - Transaction management
 *********************************************************************************/

import java.sql.*;
import java.util.Properties;

public class JDBCConnectionSteps {

    // Database configuration - Replace with your database details
    private static final String URL = "jdbc:mysql://localhost:3306/university";
    private static final String USER = "root";
    private static final String PASSWORD = "password";

    // ==================== STEP 1: LOAD DATABASE DRIVER ====================
    public static void step1_LoadDriver() {
        System.out.println("=== STEP 1: LOAD DATABASE DRIVER ===\n");

        try {
            // Method 1: Using Class.forName() (Legacy but clear)
            System.out.println("Method 1: Using Class.forName()");
            Class.forName("com.mysql.cj.jdbc.Driver");
            System.out.println("MySQL JDBC Driver loaded successfully");

        } catch (ClassNotFoundException e) {
            System.out.println("Error: MySQL JDBC Driver not found!");
            System.out.println("Make sure you have mysql-connector-java in classpath");
            System.out.println("Maven dependency: mysql:mysql-connector-java:8.0.33");
            e.printStackTrace();
        }

        System.out.println("\nAlternative: Modern JDBC 4.0+ auto-loads drivers");
        System.out.println("from META-INF/services/java.sql.Driver");
    }

    // ==================== STEP 2: ESTABLISH CONNECTION ====================
    public static Connection step2_EstablishConnection() {
        System.out.println("\n=== STEP 2: ESTABLISH DATABASE CONNECTION ===\n");

        Connection connection = null;

        try {
            // Method 1: Basic connection with parameters
            System.out.println("Method 1: Basic connection");
            connection = DriverManager.getConnection(URL, USER, PASSWORD);

            // Method 2: Connection with Properties
            System.out.println("\nMethod 2: Connection with Properties");
            Properties props = new Properties();
            props.setProperty("user", USER);
            props.setProperty("password", PASSWORD);
            props.setProperty("useSSL", "false");
            props.setProperty("serverTimezone", "UTC");

            Connection conn2 = DriverManager.getConnection(URL, props);
            conn2.close();

            // Method 3: Connection with URL parameters
            System.out.println("\nMethod 3: Connection with URL parameters");
            String urlWithParams = URL + "?user=" + USER +
                                 "&password=" + PASSWORD +
                                 "&useSSL=false&serverTimezone=UTC";
            Connection conn3 = DriverManager.getConnection(urlWithParams);
            conn3.close();

            if (connection != null) {
                System.out.println("\ n   Connection established successfully!");
                System.out.println("Connection URL: " + URL);
                System.out.println("Database: " + connection.getCatalog());
                System.out.println("Auto Commit: " + connection.getAutoCommit());
                System.out.println("Transaction Isolation: " +
                                 connection.getTransactionIsolation());
            }

        } catch (SQLException e) {
            System.out.println("\ n   Failed to establish connection!");
            System.out.println("SQL State: " + e.getSQLState());
            System.out.println("Error Code: " + e.getErrorCode());
            System.out.println("Message: " + e.getMessage());
            e.printStackTrace();
        }

        return connection;
    }

    // ==================== STEP 3: CREATE STATEMENT ====================
    public static void step3_CreateStatement(Connection connection) {
        System.out.println("\n=== STEP 3: CREATE STATEMENT OBJECTS ===\n");

        if (connection == null) {
            System.out.println("No connection available. Skipping statement creation.");
            return;
        }

        try {
            // 1. Regular Statement (for static SQL)
            System.out.println("1. Regular Statement:");
            Statement statement = connection.createStatement();
            System.out.println("   Created: Statement for static SQL queries");

            // 2. PreparedStatement (for parameterized SQL - RECOMMENDED)
            System.out.println("\n2. PreparedStatement:");
            String sql = "SELECT * FROM students WHERE age > ? AND department = ?";
            PreparedStatement preparedStatement = connection.prepareStatement(sql);
            System.out.println("   Created: PreparedStatement for parameterized queries");
            System.out.println("   SQL: " + sql);
            System.out.println("   Benefits: Precompiled, prevents SQL injection");

            // 3. CallableStatement (for stored procedures)
            System.out.println("\n3. CallableStatement:");
            CallableStatement callableStatement =
                connection.prepareCall("{call get_student_by_id(?)}");
            System.out.println("   Created: CallableStatement for stored procedures");

            // 4. Statement with result set type
            System.out.println("\n4. Statement with ResultSet type:");
            Statement scrollableStmt = connection.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY
            );
            System.out.println("   Created: Scrollable, read-only ResultSet");

            // Clean up
            statement.close();
            preparedStatement.close();
            callableStatement.close();
            scrollableStmt.close();

        } catch (SQLException e) {
            System.out.println("Error creating statement: " + e.getMessage());
        }
    }

    // ==================== STEP 4: EXECUTE QUERIES ====================
    public static void step4_ExecuteQueries(Connection connection) {
        System.out.println("\n=== STEP 4: EXECUTE SQL QUERIES ===\n");

        if (connection == null) {
            System.out.println("No connection available. Skipping query execution.");
            return;
        }

        try {
            // Create a test table if it doesn't exist
            createTestTable(connection);

            // 1. Execute SELECT query (executeQuery)
            System.out.println("1. SELECT Query (executeQuery):");
            Statement selectStmt = connection.createStatement();
            String selectSQL = "SELECT * FROM employees";
            ResultSet resultSet = selectStmt.executeQuery(selectSQL);
            System.out.println("   Executed: " + selectSQL);

            // Process results
            System.out.println("\n   Results:");
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                double salary = resultSet.getDouble("salary");
                System.out.println("    ID: " + id + ", Name: " + name + ", Salary: " + salary);
            }
            resultSet.close();
            selectStmt.close();

            // 2. Execute INSERT query (executeUpdate)
            System.out.println("\n2. INSERT Query (executeUpdate):");
            Statement insertStmt = connection.createStatement();
            String insertSQL = "INSERT INTO employees (name, salary, department) " +
                             "VALUES ('John Doe', 50000.00, 'IT')";
            int rowsInserted = insertStmt.executeUpdate(insertSQL);
            System.out.println("   Executed: " + insertSQL);
            System.out.println("   Rows inserted: " + rowsInserted);
            insertStmt.close();

            // 3. Execute UPDATE query
            System.out.println("\n3. UPDATE Query:");
            Statement updateStmt = connection.createStatement();
            String updateSQL = "UPDATE employees SET salary = salary * 1.1 WHERE department = 'IT'";
            int rowsUpdated = updateStmt.executeUpdate(updateSQL);
            System.out.println("   Executed: " + updateSQL);
            System.out.println("   Rows updated: " + rowsUpdated);
            updateStmt.close();

            // 4. Execute DELETE query
            System.out.println("\n4. DELETE Query:");
            Statement deleteStmt = connection.createStatement();
            String deleteSQL = "DELETE FROM employees WHERE name = 'John Doe'";
            int rowsDeleted = deleteStmt.executeUpdate(deleteSQL);
            System.out.println("   Executed: " + deleteSQL);
            System.out.println("   Rows deleted: " + rowsDeleted);
            deleteStmt.close();

            // 5. Execute with PreparedStatement
            System.out.println("\n5. PreparedStatement Example:");
            String prepSQL = "INSERT INTO employees (name, salary, department) VALUES (?, ?, ?)";
            PreparedStatement pstmt = connection.prepareStatement(prepSQL);

            // Insert multiple records
            Object[][] employees = {
                {"Alice Smith", 60000.0, "HR"},
                {"Bob Johnson", 75000.0, "Engineering"},
                {"Carol Williams", 55000.0, "Marketing"}
            };

            for (Object[] emp : employees) {
                pstmt.setString(1, (String) emp[0]);
                pstmt.setDouble(2, (Double) emp[1]);
                pstmt.setString(3, (String) emp[2]);
                int affected = pstmt.executeUpdate();
                System.out.println("   Inserted: " + emp[0] + " (" + affected + " row)");
            }
            pstmt.close();

        } catch (SQLException e) {
            System.out.println("Error executing queries: " + e.getMessage());
        }
    }

    // ==================== STEP 5: PROCESS RESULTSET ====================
    public static void step5_ProcessResultSet(Connection connection) {
        System.out.println("\n=== STEP 5: PROCESS RESULTSET ===\n");

        if (connection == null) {
            System.out.println("No connection available. Skipping ResultSet processing.");
            return;
        }

        try {
            Statement stmt = connection.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY
            );

            ResultSet rs = stmt.executeQuery("SELECT * FROM employees ORDER BY salary DESC");

            System.out.println("ResultSet Metadata:");
            ResultSetMetaData metaData = rs.getMetaData();
            int columnCount = metaData.getColumnCount();
            System.out.println("Number of columns: " + columnCount);

            for (int i = 1; i <= columnCount; i++) {
                System.out.println("  Column " + i + ": " +
                                 metaData.getColumnName(i) + " (" +
                                 metaData.getColumnTypeName(i) + ")");
            }

            System.out.println("\nProcessing ResultSet:");

            // Method 1: Using column names (Recommended)
            System.out.println("\n1. Using column names:");
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                double salary = rs.getDouble("salary");
                String dept = rs.getString("department");
                System.out.println("   ID: " + id + ", Name: " + name +
                                 ", Salary: $" + salary + ", Dept: " + dept);
            }

            // Method 2: Using column indexes (Faster but less readable)
            System.out.println("\n2. Using column indexes:");
            rs.beforeFirst(); // Reset cursor
            while (rs.next()) {
                System.out.println("   ID: " + rs.getInt(1) +
                                 ", Name: " + rs.getString(2) +
                                 ", Salary: $" + rs.getDouble(3) +
                                 ", Dept: " + rs.getString(4));
            }

            // Method 3: Scrollable ResultSet navigation
            System.out.println("\n3. Scrollable ResultSet Navigation:");
            if (rs.last()) {
                System.out.println("   Last row - ID: " + rs.getInt("id"));
            }

            if (rs.first()) {
                System.out.println("   First row - ID: " + rs.getInt("id"));
            }

            if (rs.absolute(2)) {
                System.out.println("   Row 2 - Name: " + rs.getString("name"));
            }

            // Method 4: Getting different data types
            System.out.println("\n4. Different data type getters:");
            rs.beforeFirst();
            if (rs.next()) {
                System.out.println("   getObject(): " + rs.getObject("name"));
                System.out.println("   getString(): " + rs.getString("name"));
                System.out.println("   getInt(): " + rs.getInt("id"));
                System.out.println("   getDouble(): " + rs.getDouble("salary"));
                System.out.println("   getDate(): " + rs.getDate("hire_date"));
            }

            rs.close();
            stmt.close();

        } catch (SQLException e) {
            System.out.println("Error processing ResultSet: " + e.getMessage());
        }
    }

    // ==================== STEP 6: CLOSE RESOURCES ====================
    public static void step6_CloseResources(Connection connection) {
        System.out.println("\n=== STEP 6: CLOSE RESOURCES PROPERLY ===\n");

        // Method 1: Traditional try-catch-finally
        System.out.println("Method 1: Traditional try-catch-finally");

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

        try {
            conn = DriverManager.getConnection(URL, USER, PASSWORD);
            stmt = conn.createStatement();
            rs = stmt.executeQuery("SELECT 1");

            // Process results...

        } catch (SQLException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            // Close in reverse order: ResultSet -> Statement -> Connection
            try {
                if (rs != null) rs.close();
            } catch (SQLException e) {
                System.out.println("Error closing ResultSet: " + e.getMessage());
            }

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

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

        // Method 2: Try-with-resources (Java 7+)
        System.out.println("\nMethod 2: Try-with-resources (Recommended)");

        try (Connection conn2 = DriverManager.getConnection(URL, USER, PASSWORD);
             Statement stmt2 = conn2.createStatement();
             ResultSet rs2 = stmt2.executeQuery("SELECT 1")) {

            System.out.println("   Resources automatically closed");

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

        if (connection != null) {
            try {
                connection.close();
                System.out.println("\n   Main connection closed successfully");
            } catch (SQLException e) {
                System.out.println("\n   Error closing main connection: " + e.getMessage());
            }
        }
    }

    // ==================== HELPER METHOD: CREATE TEST TABLE ====================
    private static void createTestTable(Connection connection) throws SQLException {
        Statement stmt = connection.createStatement();

        // Drop table if exists (for clean testing)
        try {
            stmt.execute("DROP TABLE IF EXISTS employees");
        } catch (SQLException e) {
            // Ignore if table doesn't exist
        }

        // Create employees table
        String createTableSQL = "CREATE TABLE employees (" +
                               "id INT PRIMARY KEY AUTO_INCREMENT, " +
                               "name VARCHAR(100) NOT NULL, " +
                               "salary DECIMAL(10,2), " +
                               "department VARCHAR(50), " +
                               "hire_date DATE DEFAULT CURRENT_DATE)";

        stmt.execute(createTableSQL);

        // Insert sample data
        String insertDataSQL = "INSERT INTO employees (name, salary, department) VALUES " +
                              "('John Smith', 50000.00, 'IT'), " +
                              "('Jane Doe', 60000.00, 'HR'), " +
                              "('Mike Johnson', 75000.00, 'Engineering')";

        stmt.execute(insertDataSQL);

        stmt.close();
        System.out.println("Test table 'employees' created with sample data");
    }

    // ==================== COMPLETE EXAMPLE ====================
    public static void completeExample() {
        System.out.println("\n=== COMPLETE JDBC EXAMPLE ===");

        // Using try-with-resources for automatic resource management
        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD)) {

            System.out.println("1. Connection established");

            // Create a table for demonstration
            try (Statement createStmt = conn.createStatement()) {
                createStmt.execute("CREATE TABLE IF NOT EXISTS products (" +
                                 "id INT PRIMARY KEY AUTO_INCREMENT, " +
                                 "name VARCHAR(100), " +
                                 "price DECIMAL(10,2), " +
                                 "quantity INT)");
                System.out.println("2. Table created/verified");
            }

            // Insert data using PreparedStatement
            String insertSQL = "INSERT INTO products (name, price, quantity) VALUES (?, ?, ?)";
            try (PreparedStatement pstmt = conn.prepareStatement(insertSQL)) {

                Object[][] products = {
                    {"Laptop", 999.99, 10},
                    {"Mouse", 25.50, 100},
                    {"Keyboard", 75.00, 50},
                    {"Monitor", 299.99, 20}
                };

                for (Object[] product : products) {
                    pstmt.setString(1, (String) product[0]);
                    pstmt.setDouble(2, (Double) product[1]);
                    pstmt.setInt(3, (Integer) product[2]);
                    pstmt.executeUpdate();
                }
                System.out.println("3. Data inserted");
            }

            // Query data
            try (Statement stmt = conn.createStatement();
                 ResultSet rs = stmt.executeQuery("SELECT * FROM products")) {

                System.out.println("\n4. Query Results:");
                System.out.println("ID\tName\t\tPrice\tQuantity");
                System.out.println("----------------------------------------");

                double totalValue = 0;
                while (rs.next()) {
                    int id = rs.getInt("id");
                    String name = rs.getString("name");
                    double price = rs.getDouble("price");
                    int quantity = rs.getInt("quantity");

                System.out.printf("%d\t%-10s\t$%.2f\t%d%n", id, name, price, quantity);

                   totalValue += price * quantity;
            }

            System.out.println("----------------------------------------");
            System.out.printf("Total inventory value: $%.2f%n", totalValue);
        }

            // Update data
            try (Statement updateStmt = conn.createStatement()) {
                int rowsUpdated = updateStmt.executeUpdate(
                    "UPDATE products SET price = price * 0.9 WHERE quantity > 30"
                );
                System.out.println("\n5. Prices updated for " + rowsUpdated + " products");
            }

            // Transaction example
            System.out.println("\n6. Transaction Example:");
            conn.setAutoCommit(false); // Start transaction

            try {
                try (Statement transStmt = conn.createStatement()) {
                    // Multiple operations
                    transStmt.executeUpdate("UPDATE products SET quantity = quantity - 5 WHERE name = 'Laptop'");
                    transStmt.executeUpdate("UPDATE products SET quantity = quantity + 5 WHERE name = 'Mouse'");

                    // Simulate an error condition
                    boolean errorCondition = false; // Change to true to test rollback
                    if (errorCondition) {
                        throw new SQLException("Simulated error during transaction");
                    }

                    conn.commit(); // Commit transaction
                    System.out.println("   Transaction committed successfully");
                }
            } catch (SQLException e) {
                conn.rollback(); // Rollback on error
                System.out.println("   Transaction rolled back due to: " + e.getMessage());
            } finally {
                conn.setAutoCommit(true); // Restore auto-commit
            }

```

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

    // ==================== MAIN METHOD ====================
    public static void main(String[] args) {
        System.out.println("=== JDBC CONNECTION STEPS - COMPLETE GUIDE ===\n");

        System.out.println("This example demonstrates all 6 steps of JDBC:");
        System.out.println("1. Load Database Driver");
        System.out.println("2. Establish Connection");
        System.out.println("3. Create Statement");
        System.out.println("4. Execute Queries");
        System.out.println("5. Process ResultSet");
        System.out.println("6. Close Resources\n");

        // Step 1: Load Driver
        step1_LoadDriver();

        // Step 2: Establish Connection
        Connection connection = step2_EstablishConnection();

        // Step 3: Create Statement
        step3_CreateStatement(connection);

        // Step 4: Execute Queries
        step4_ExecuteQueries(connection);

        // Step 5: Process ResultSet
        step5_ProcessResultSet(connection);

        // Step 6: Close Resources
        step6_CloseResources(connection);

        // Complete Example
        System.out.println("\n".repeat(3));
        System.out.println("=".repeat(60));
        System.out.println("DEMONSTRATING COMPLETE JDBC WORKFLOW");
        System.out.println("=".repeat(60));
        completeExample();

        System.out.println("\n=== JDBC BEST PRACTICES ===");
        System.out.println("1. Use PreparedStatement to prevent SQL injection");
        System.out.println("2. Always close resources in finally block or use try-with-resources");
        System.out.println("3. Use connection pooling for production applications");
        System.out.println("4. Handle SQLException properly with specific error messages");
        System.out.println("5. Use transactions for multiple related operations");
        System.out.println("6. Validate and sanitize user input before database operations");
        System.out.println("7. Use appropriate data types (getInt for INT, getString for VARCHAR)");
        System.out.println("8. Limit ResultSet size for large queries (use LIMIT clause)");
        System.out.println("9. Use batch updates for multiple insert/update operations");
        System.out.println("10. Test with different database configurations");
    }
}
```

Listing 2: Complete JDBC Connection Example Program

3 Introduction to Lambda Expressions (Java 8+)

3.1 Lambda Expressions Fundamentals

```java
/*******************************************************************************
 * PROGRAM: LambdaExpressionsGuide.java
 *
 * PURPOSE: This comprehensive program introduces Lambda Expressions in Java 8+,
 *          showing how they enable functional programming and reduce boilerplate
 *          code. It covers syntax, functional interfaces, method references,
 *          and real-world applications.
 *
 * PROBLEM SOLVED: Traditional Java code often involves verbose anonymous inner
 *                 classes. This program solves:
 *                 1. Verbosity reduction in event handlers, threads, comparators
 *                 2. Readability improvement through concise syntax
 *                 3. Functional programming adoption in Java
 *                 4. Stream API compatibility requirement
 *
 * WHY NECESSARY: Lambda expressions are essential for modern Java development:
 *                1. Required for using Stream API effectively
 *                2. Industry standard for concise, readable code
 *                3. Foundation for reactive programming
 *                4. Enables parallel processing patterns
 *                5. Reduces boilerplate code by 70-80%
 *
 * KEY CONCEPTS DEMONSTRATED:
 * - Lambda syntax (parameters, arrow, body)
 * - Functional interfaces (built-in and custom)
 * - Method references (4 types)
 * - Variable capture rules
 * - Real-world use cases
 * - Best practices and common pitfalls
 ******************************************************************************/
```

```java

import java.util.*;
import java.util.function.*;

public class LambdaExpressionsGuide {

    // ==================== FUNCTIONAL INTERFACES ====================
    @FunctionalInterface
    interface SimpleCalculator {
        int calculate(int a, int b);
    }

    @FunctionalInterface
    interface StringTransformer {
        String transform(String input);
    }

    @FunctionalInterface
    interface ConditionChecker {
        boolean check(int number);
    }

    // ==================== 1. BASIC LAMBDA SYNTAX ====================
    public static void basicLambdaSyntax() {
        System.out.println("=== 1. BASIC LAMBDA SYNTAX ===\n");

        // Before Java 8: Anonymous class
        System.out.println("Before Java 8 - Anonymous Class:");
        Runnable oldRunnable = new Runnable() {
            @Override
            public void run() {
                System.out.println("   Running with anonymous class");
            }
        };
        oldRunnable.run();

        // Java 8+: Lambda expression
        System.out.println("\nJava 8+ - Lambda Expression:");
        Runnable newRunnable = () -> System.out.println("   Running with lambda");
        newRunnable.run();

        // Lambda with parameters
        System.out.println("\nLambda with Parameters:");
        SimpleCalculator adder = (a, b) -> a + b;
        SimpleCalculator multiplier = (x, y) -> x * y;

        System.out.println("   Addition: 5 + 3 = " + adder.calculate(5, 3));
        System.out.println("   Multiplication: 5 * 3 = " + multiplier.calculate(5, 3));

        // Lambda with explicit types
        System.out.println("\nLambda with Explicit Types:");
        SimpleCalculator subtractor = (int a, int b) -> a - b;
        System.out.println("   Subtraction: 10 - 4 = " + subtractor.calculate(10, 4));

        // Lambda with multiple statements
        System.out.println("\nLambda with Multiple Statements:");
        SimpleCalculator complexCalc = (a, b) -> {
            int sum = a + b;
            int product = a * b;
            return sum + product;
        };
        System.out.println("   Complex calculation (5,3): " + complexCalc.calculate(5, 3));
    }

    // ==================== 2. BUILT-IN FUNCTIONAL INTERFACES ====================
    public static void builtInFunctionalInterfaces() {
        System.out.println("\n=== 2. BUILT-IN FUNCTIONAL INTERFACES ===\n");

        // 1. Predicate<T> - Tests a condition
        System.out.println("1. Predicate<T> - Tests a condition:");
        Predicate<Integer> isEven = n -> n % 2 == 0;
        Predicate<Integer> isPositive = n -> n > 0;
        Predicate<String> isEmpty = String::isEmpty;

        System.out.println("   Is 10 even? " + isEven.test(10));
        System.out.println("   Is -5 positive? " + isPositive.test(-5));
        System.out.println("   Is empty string? " + isEmpty.test(""));

        // Predicate chaining
        Predicate<Integer> isEvenAndPositive = isEven.and(isPositive);
        System.out.println("   Is 6 even AND positive? " + isEvenAndPositive.test(6));

        // 2. Function<T, R> - Transforms input to output
        System.out.println("\n2. Function<T, R> - Transforms input to output:");
        Function<String, Integer> stringLength = String::length;
        Function<Integer, String> intToString = Object::toString;
        Function<String, String> toUpperCase = String::toUpperCase;
        Function<String, String> addExclamation = s -> s + "!";

        System.out.println("   Length of 'Hello': " + stringLength.apply("Hello"));
        System.out.println("   123 as string: " + intToString.apply(123));

        // Function composition
        Function<String, String> shout = toUpperCase.andThen(addExclamation);
        System.out.println("   Shout 'hello': " + shout.apply("hello"));

        // 3. Consumer<T> - Consumes input, returns nothing
        System.out.println("\n3. Consumer<T> - Consumes input, returns nothing:");
        Consumer<String> printer = System.out::println;
        Consumer<Integer> squarePrinter = n -> System.out.println(n * n);

        System.out.print("   Printing with consumer: ");
        printer.accept("Hello Consumer!");
        System.out.print("   Square of 5: ");
        squarePrinter.accept(5);

        // Consumer chaining
        Consumer<String> printAndUpperCase = printer.andThen(s ->
            System.out.println("Uppercase: " + s.toUpperCase()));
        System.out.print("   Chained consumer: ");
        printAndUpperCase.accept("test");

        // 4. Supplier<T> - Provides values
        System.out.println("\n4. Supplier<T> - Provides values:");
        Supplier<Double> randomSupplier = Math::random;
        Supplier<String> greetingSupplier = () -> "Hello World!";
        Supplier<List<String>> listSupplier = ArrayList::new;

        System.out.println("   Random number: " + randomSupplier.get());
        System.out.println("   Greeting: " + greetingSupplier.get());
        System.out.println("   New list: " + listSupplier.get());

        // 5. UnaryOperator<T> - Function where input and output are same type
        System.out.println("\n5. UnaryOperator<T> - Function with same input/output type:");
        UnaryOperator<Integer> square = n -> n * n;
        UnaryOperator<String> reverse = s -> new StringBuilder(s).reverse().toString();

        System.out.println("   Square of 7: " + square.apply(7));
        System.out.println("   Reverse ’lambda’: " + reverse.apply("lambda"));

        // 6. BinaryOperator<T> - Takes two inputs, returns same type
        System.out.println("\n6. BinaryOperator<T> - Two inputs, returns same type:");
        BinaryOperator<Integer> max = Math::max;
        BinaryOperator<String> concatenator = (s1, s2) -> s1 + " " + s2;

        System.out.println("   Max of 10 and 20: " + max.apply(10, 20));
        System.out.println("   Concatenate: " + concatenator.apply("Hello", "World"));
    }

    // =================== 3. LAMBDA WITH COLLECTIONS ====================
    public static void lambdaWithCollections() {
        System.out.println("\n=== 3. LAMBDA WITH COLLECTIONS ===\n");

        List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry", "Date", "Elderberry");

        // Before Java 8: External iteration
        System.out.println("Before Java 8 - External iteration:");
```

```java
        for (String fruit : fruits) {
            if (fruit.startsWith("A")) {
                System.out.println("   " + fruit);
            }
        }

        // Java 8: Internal iteration with forEach
        System.out.println("\nJava 8 - Internal iteration with forEach:");
        fruits.forEach(fruit -> System.out.println("   " + fruit));

        // Method reference
        System.out.println("\nUsing Method Reference:");
        fruits.forEach(System.out::println);

        // Filtering with Predicate
        System.out.println("\nFiltering fruits starting with 'C':");
        Predicate<String> startsWithC = s -> s.startsWith("C");
        fruits.stream()
              .filter(startsWithC)
              .forEach(System.out::println);

        // Transforming with Function
        System.out.println("\nTransforming to uppercase:");
        Function<String, String> toUpper = String::toUpperCase;
        fruits.stream()
              .map(toUpper)
              .forEach(System.out::println);

        // Sorting with Comparator
        System.out.println("\nSorted by length:");
        Comparator<String> byLength = (s1, s2) -> s1.length() - s2.length();
        fruits.stream()
              .sorted(byLength)
              .forEach(f -> System.out.println("   " + f));

        // Custom sorting
        System.out.println("\nSorted by length then alphabetically:");
        Comparator<String> byLengthThenAlpha =
            Comparator.comparingInt(String::length)
                     .thenComparing(Comparator.naturalOrder());

        fruits.stream()
              .sorted(byLengthThenAlpha)
              .forEach(f -> System.out.println("   " + f));
    }

    // ==================== 4. METHOD REFERENCES ====================
    public static void methodReferences() {
        System.out.println("\n=== 4. METHOD REFERENCES ===\n");

        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");

        // 1. Static method reference
        System.out.println("1. Static Method Reference:");
        names.forEach(System.out::println); // Equivalent to: s -> System.out.println(s)

        // Using custom static method
        System.out.println("\nUsing custom static method:");
        names.forEach(LambdaExpressionsGuide::printWithPrefix);

        // 2. Instance method reference on specific instance
        System.out.println("\n2. Instance Method Reference (specific instance):");
        String prefix = "Name: ";
        names.forEach(prefix::concat); // Equivalent to: s -> prefix.concat(s)

        // 3. Instance method reference on arbitrary instance
        System.out.println("\n3. Instance Method Reference (arbitrary instance):");
        names.forEach(String::toUpperCase); // Equivalent to: s -> s.toUpperCase()

        // 4. Constructor reference
        System.out.println("\n4. Constructor Reference:");
        Supplier<List<String>> listSupplier = ArrayList::new; // Equivalent to: () -> new ArrayList<>()
        List<String> newList = listSupplier.get();
        newList.add("New Element");
        System.out.println("   New list: " + newList);

        // Constructor reference with parameters
        System.out.println("\nConstructor reference with parameters:");
        Function<String, Integer> parseInt = Integer::new; // Equivalent to: s -> new Integer(s)
        System.out.println("   Parsed '123': " + parseInt.apply("123"));
    }

    private static void printWithPrefix(String s) {
        System.out.println("   >>> " + s);
    }

    // ==================== 5. REAL-WORLD EXAMPLES ====================
    public static void realWorldExamples() {
        System.out.println("\n=== 5. REAL-WORLD LAMBDA EXAMPLES ===\n");

        // Example 1: Event Handlers (GUI)
        System.out.println("Example 1: Event Handlers in GUI");
        System.out.println("// Old way:");
        System.out.println("button.addActionListener(new ActionListener() {");
        System.out.println("    public void actionPerformed(ActionEvent e) {");
        System.out.println("        System.out.println(\"Button clicked!\");");
        System.out.println("    }");
        System.out.println("});");

        System.out.println("\n// New way with lambda:");
        System.out.println("button.addActionListener(e -> System.out.println(\"Button clicked!\"));");

        // Example 2: Thread Creation
        System.out.println("\nExample 2: Thread Creation");
        System.out.println("// Old way:");
        System.out.println("new Thread(new Runnable() {");
        System.out.println("    public void run() {");
        System.out.println("        System.out.println(\"Thread running\");");
        System.out.println("    }");
        System.out.println("}).start();");

        System.out.println("\n// New way with lambda:");
        System.out.println("new Thread(() -> System.out.println(\"Thread running\")).start();");

        // Example 3: Sorting Collections
        System.out.println("\nExample 3: Sorting Employees");
        List<Employee> employees = Arrays.asList(
            new Employee("Alice", "Engineering", 75000),
            new Employee("Bob", "Sales", 60000),
            new Employee("Charlie", "Engineering", 80000),
            new Employee("Diana", "Marketing", 55000)
        );

        System.out.println("\nEmployees sorted by salary (descending):");
        employees.sort((e1, e2) -> e2.getSalary() - e1.getSalary());
        employees.forEach(e -> System.out.println("   " + e));

        // Example 4: Filtering and Mapping
        System.out.println("\nExample 4: Filtering high-salary Engineering employees:");
        employees.stream()
                 .filter(e -> e.getDepartment().equals("Engineering"))
                 .filter(e -> e.getSalary() > 70000)
                 .map(Employee::getName)
                 .forEach(name -> System.out.println("   " + name));

        // Example 5: Custom Functional Interface
        System.out.println("\nExample 5: Custom Validator");
        Validator<String> emailValidator = email -> email.contains("@") && email.contains(".");
        Validator<Integer> ageValidator = age -> age >= 18;

        System.out.println("   Valid email 'test@example.com'? " +
                         emailValidator.validate("test@example.com"));
        System.out.println("   Valid age 25? " + ageValidator.validate(25));
    }

    // ==================== 6. VARIABLE CAPTURE ====================
    public static void variableCapture() {
        System.out.println("\n=== 6. VARIABLE CAPTURE IN LAMBDAS ===\n");

        // Effectively final local variable
        final String fixedPrefix = "Item: ";
        String variablePrefix = "Product: ";

        // variablePrefix must be effectively final
        // variablePrefix = "Changed: "; // This would cause error

        List<String> items = Arrays.asList("Book", "Pen", "Notebook");

        System.out.println("Using effectively final variables:");
        items.forEach(item -> {
            // Can access final/effectively final variables
            System.out.println("   " + fixedPrefix + item);
            System.out.println("   " + variablePrefix + item);

            // Cannot modify captured variables
            // variablePrefix = "New: "; // Compilation error
        });

        // Instance and static variable capture
        System.out.println("\nInstance and static variable capture:");
        LambdaDemo demo = new LambdaDemo();
        demo.instanceVariable = 100;

        items.forEach(item -> {
            // Can modify instance variables
            demo.instanceVariable++;
            // Can modify static variables
            LambdaDemo.staticVariable++;

            System.out.println("   Instance: " + demo.instanceVariable +
                             ", Static: " + LambdaDemo.staticVariable);
        });
    }

    // ==================== 7. LAMBDA BEST PRACTICES ====================
    public static void lambdaBestPractices() {
        System.out.println("\n=== 7. LAMBDA BEST PRACTICES ===\n");

        System.out.println("1. Keep Lambdas Short and Simple:");
        System.out.println("   Good: names.stream().filter(n -> n.length() > 3)");
        System.out.println("   Bad: Complex logic in lambda - extract to method");

        System.out.println("\n2. Use Method References When Possible:");
        System.out.println("   Instead of: s -> s.toUpperCase()");
        System.out.println("   Use: String::toUpperCase");

        System.out.println("\n3. Avoid Side Effects:");
        System.out.println("   Pure functions are better than mutating external state");

        System.out.println("\n4. Use Descriptive Parameter Names:");
        System.out.println("   Good: (person, department) -> ...");
        System.out.println("   Bad: (p, d) -> ...");

        System.out.println("\n5. Consider Type Inference:");
        System.out.println("   Let compiler infer types when clear");
        System.out.println("   (a, b) -> a + b  instead of  (int a, int b) -> a + b");

        System.out.println("\n6. Chain Operations Readably:");
        System.out.println("   list.stream()");
        System.out.println("       .filter(...)");
        System.out.println("       .map(...)");
        System.out.println("       .collect(...);");
    }

    // ==================== MAIN METHOD ====================
    public static void main(String[] args) {
        System.out.println("=== LAMBDA EXPRESSIONS - COMPLETE GUIDE ===\n");

        System.out.println("Lambda Expressions introduce functional programming");
        System.out.println("features to Java, enabling concise, readable code.\n");

        // 1. Basic Syntax
        basicLambdaSyntax();

        // 2. Built-in Functional Interfaces
        builtInFunctionalInterfaces();

        // 3. Lambda with Collections
        lambdaWithCollections();

        // 4. Method References
        methodReferences();

        // 5. Real-world Examples
        realWorldExamples();

        // 6. Variable Capture
        variableCapture();

        // 7. Best Practices
        lambdaBestPractices();

        System.out.println("\n=== KEY BENEFITS OF LAMBDA EXPRESSIONS ===");
        System.out.println("1. Conciseness: Less boilerplate code");
        System.out.println("2. Readability: More expressive code");
        System.out.println("3. Functional Programming: Support for FP paradigms");
        System.out.println("4. Parallelism: Easier parallel processing");
        System.out.println("5. API Design: Enables fluent APIs");

        System.out.println("\n=== COMMON PITFALLS ===");
        System.out.println("1. Overusing lambdas for complex logic");
        System.out.println("2. Not understanding variable capture rules");
        System.out.println("3. Ignoring exception handling in lambdas");
        System.out.println("4. Performance overhead in some cases");
        System.out.println("5. Debugging can be more challenging");
    }

    // ==================== SUPPORTING CLASSES ====================
    static class Employee {
        private String name;
        private String department;
        private int salary;

        public Employee(String name, String department, int salary) {
            this.name = name;
            this.department = department;
            this.salary = salary;
        }

        public String getName() { return name; }
        public String getDepartment() { return department; }
        public int getSalary() { return salary; }

        @Override
        public String toString() {
            return name + " (" + department + "): $" + salary;
        }
    }

    @FunctionalInterface
    interface Validator<T> {
        boolean validate(T value);
    }

    static class LambdaDemo {
        int instanceVariable;
        static int staticVariable = 0;
    }
}
```

Listing 3: Lambda Expressions - Complete Guide Program

4 Stream API (Java 8+)

```java
/* *******************************************************************************
 * PROGRAM: StreamAPIGuide.java
 *
 * PURPOSE: This comprehensive guide demonstrates Java 8+ Stream API for
 *          declarative data processing. It shows how to process collections
 *          efficiently using functional programming patterns.
 *
 * PROBLEM SOLVED: Traditional Java collection processing involves:
 *                 1. Verbose iteration code
 *                 2. Complex filtering and transformation logic
 *                 3. Difficulty in parallel processing
 *                 4. Code duplication for common operations
 *
 *                 Stream API solves these by providing:
 *                 1. Declarative syntax (what, not how)
 *                 2. Composable operations (pipeline)
 *                 3. Automatic parallelization
 *                 4. Built-in common operations
 *
 * WHY NECESSARY: Stream API is essential for modern Java because:
 *                1. Industry standard for data processing
 *                2. Enables functional programming in Java
 *                3. Simplifies complex data transformations
 *                4. Improves code readability and maintainability
 *                5. Built-in support for parallel processing
 *                6. Required for big data and analytics applications
 *
 * KEY CONCEPTS DEMONSTRATED:
 * - Stream creation (from collections, arrays, generators)
 * - Intermediate operations (filter, map, flatMap, sorted, distinct)
 * - Terminal operations (collect, reduce, forEach, count)
 * - Primitive streams (IntStream, LongStream, DoubleStream)
 * - Parallel streams and performance considerations
 * - Collectors for advanced data aggregation
 * - Real-world data processing scenarios
 ******************************************************************************/

import java.util.*;
import java.util.stream.*;
import java.util.function.*;
import java.time.LocalDate;

public class StreamAPIGuide {

    // ==================== 1. STREAM BASICS ====================
    public static void streamBasics() {
        System.out.println("=== 1. STREAM API BASICS ===\n");

        List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry", "Date", "Elderberry");

        System.out.println("Source Collection: " + fruits);

        // Creating streams
        System.out.println("\n1. Different ways to create streams:");

        // From Collection
        Stream<String> stream1 = fruits.stream();
        System.out.println("   From Collection: fruits.stream()");

        // From Array
        String[] array = {"One", "Two", "Three"};
        Stream<String> stream2 = Arrays.stream(array);
        System.out.println("   From Array: Arrays.stream(array)");

        // Static factory methods
        Stream<String> stream3 = Stream.of("A", "B", "C");
        System.out.println("   Using Stream.of(): Stream.of(\"A\", \"B\", \"C\")");

        Stream<Integer> stream4 = Stream.iterate(1, n -> n + 1).limit(5);
        System.out.println("   Infinite stream: Stream.iterate(1, n -> n + 1).limit(5)");

        Stream<Double> stream5 = Stream.generate(Math::random).limit(3);
        System.out.println("   Generated stream: Stream.generate(Math::random).limit(3)");

        // Stream operations pipeline
        System.out.println("\n2. Stream Operations Pipeline:");
        System.out.println("   Source     Intermediate Operations     Terminal Operation");

        long count = fruits.stream()            // Source
                           .filter(f -> f.length() > 5)  // Intermediate
                           .map(String::toUpperCase)     // Intermediate
                           .count();                     // Terminal

        System.out.println("   Example: Count fruits with length > 5 = " + count);

        // Characteristics of streams
        System.out.println("\n3. Stream Characteristics:");
        System.out.println("   - Not a data structure, carries values from source");
        System.out.println("   - Functional in nature (doesn't modify source)");
        System.out.println("   - Lazily evaluated (only when terminal operation called)");
        System.out.println("   - Possibly unbounded");
        System.out.println("   - Consumable (can be traversed only once)");
    }

    // ==================== 2. INTERMEDIATE OPERATIONS ====================
    public static void intermediateOperations() {
        System.out.println("\n=== 2. INTERMEDIATE OPERATIONS ===\n");

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        System.out.println("Original numbers: " + numbers);

        // 1. filter() - Selects elements based on predicate
        System.out.println("\n1. filter() - Selects elements:");
        List<Integer> evenNumbers = numbers.stream()
                                           .filter(n -> n % 2 == 0)
                                           .collect(Collectors.toList());
        System.out.println("   Even numbers: " + evenNumbers);

        // 2. map() - Transforms each element
        System.out.println("\n2. map() - Transforms elements:");
        List<Integer> squares = numbers.stream()
                                      .map(n -> n * n)
                                      .collect(Collectors.toList());
        System.out.println("   Squares: " + squares);

        List<String> numberStrings = numbers.stream()
                                           .map(n -> "Number: " + n)
                                           .collect(Collectors.toList());
        System.out.println("   Number strings: " + numberStrings);

        // 3. flatMap() - Flattens nested structures
        System.out.println("\n3. flatMap() - Flattens nested structures:");
        List<List<String>> nestedLists = Arrays.asList(
            Arrays.asList("A", "B", "C"),
            Arrays.asList("D", "E", "F"),
            Arrays.asList("G", "H", "I")
        );

        List<String> flatList = nestedLists.stream()
                                          .flatMap(List::stream)
                                          .collect(Collectors.toList());
        System.out.println("   Nested: " + nestedLists);
        System.out.println("   Flattened: " + flatList);

        // 4. distinct() - Removes duplicates
        System.out.println("\n4. distinct() - Removes duplicates:");
        List<Integer> withDuplicates = Arrays.asList(1, 2, 2, 3, 3, 3, 4, 4, 4, 4);
        List<Integer> distinct = withDuplicates.stream()
                                              .distinct()
                                              .collect(Collectors.toList());
        System.out.println("   With duplicates: " + withDuplicates);
        System.out.println("   Distinct: " + distinct);

        // 5. sorted() - Sorts elements
        System.out.println("\n5. sorted() - Sorts elements:");
        List<Integer> shuffled = Arrays.asList(5, 3, 8, 1, 9, 2);
        List<Integer> sortedAsc = shuffled.stream()
                                         .sorted()
                                         .collect(Collectors.toList());
        List<Integer> sortedDesc = shuffled.stream()
                                          .sorted(Comparator.reverseOrder())
                                          .collect(Collectors.toList());
        System.out.println("   Original: " + shuffled);
        System.out.println("   Sorted ascending: " + sortedAsc);
        System.out.println("   Sorted descending: " + sortedDesc);

        // 6. peek() - Debugging operation
        System.out.println("\n6. peek() - For debugging:");
        List<Integer> processed = numbers.stream()
```

```java
                                         .peek(n -> System.out.print("   Before filter: " + n + " "))
                                         .filter(n -> n > 5)
                                         .peek(n -> System.out.println("   After filter: " + n))
                                         .collect(Collectors.toList());
        System.out.println("   Result: " + processed);

        // 7. limit() and skip()
        System.out.println("\n7. limit() and skip():");
        List<Integer> limited = numbers.stream()
                                       .skip(3)        // Skip first 3
                                       .limit(4)       // Take next 4
                                       .collect(Collectors.toList());
        System.out.println("   Skip 3, limit 4: " + limited);

        // 8. Multiple operations chained
        System.out.println("\n8. Chained Intermediate Operations:");
        List<String> result = numbers.stream()
                                     .filter(n -> n % 2 != 0)      // Odd numbers
                                     .map(n -> n * 2)              // Double them
                                     .filter(n -> n > 5)           // Greater than 5
                                     .map(n -> "#" + n)            // Add prefix
                                     .collect(Collectors.toList());
        System.out.println("   Chained result: " + result);
    }

    // ==================== 3. TERMINAL OPERATIONS ====================
    public static void terminalOperations() {
        System.out.println("\n=== 3. TERMINAL OPERATIONS ===\n");

        List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);
        System.out.println("Numbers: " + numbers);

        // 1. forEach() - Performs action on each element
        System.out.println("\n1. forEach() - Performs action:");
        System.out.print("   Printing numbers: ");
        numbers.stream().forEach(n -> System.out.print(n + " "));

        // 2. collect() - Accumulates elements into collection
        System.out.println("\n\n2. collect() - Accumulates elements:");
        Set<Integer> numberSet = numbers.stream().collect(Collectors.toSet());
        System.out.println("   As Set: " + numberSet);

        // 3. toArray() - Converts to array
        System.out.println("\n3. toArray() - Converts to array:");
        Integer[] array = numbers.stream().toArray(Integer[]::new);
        System.out.println("   Array: " + Arrays.toString(array));

        // 4. reduce() - Combines elements
        System.out.println("\n4. reduce() - Combines elements:");
        Optional<Integer> sum = numbers.stream().reduce((a, b) -> a + b);
        Optional<Integer> product = numbers.stream().reduce((a, b) -> a * b);
        Integer sumWithIdentity = numbers.stream().reduce(0, (a, b) -> a + b);

        System.out.println("   Sum: " + sum.orElse(0));
        System.out.println("   Product: " + product.orElse(0));
        System.out.println("   Sum with identity: " + sumWithIdentity);

        // 5. min() and max() - Finds min/max element
        System.out.println("\n5. min() and max():");
        Optional<Integer> min = numbers.stream().min(Integer::compare);
        Optional<Integer> max = numbers.stream().max(Integer::compare);

        System.out.println("   Min: " + min.orElse(null));
        System.out.println("   Max: " + max.orElse(null));

        // 6. count() - Counts elements
        System.out.println("\n6. count():");
        long count = numbers.stream().count();
        System.out.println("   Count: " + count);

        // 7. anyMatch(), allMatch(), noneMatch() - Boolean checks
        System.out.println("\n7. anyMatch(), allMatch(), noneMatch():");
        boolean anyGreaterThan25 = numbers.stream().anyMatch(n -> n > 25);
        boolean allGreaterThan5 = numbers.stream().allMatch(n -> n > 5);
        boolean noneGreaterThan100 = numbers.stream().noneMatch(n -> n > 100);

        System.out.println("   Any > 25: " + anyGreaterThan25);
        System.out.println("   All > 5: " + allGreaterThan5);
        System.out.println("   None > 100: " + noneGreaterThan100);

        // 8. findFirst() and findAny() - Finding elements
        System.out.println("\n8. findFirst() and findAny():");
        Optional<Integer> first = numbers.stream().findFirst();
        Optional<Integer> any = numbers.stream().findAny();

        System.out.println("   First: " + first.orElse(null));
        System.out.println("   Any: " + any.orElse(null));

        // 9. Collectors utility methods
        System.out.println("\n9. Advanced Collectors:");
        Double average = numbers.stream()
                               .collect(Collectors.averagingInt(n -> n));
        Integer summing = numbers.stream()
                                .collect(Collectors.summingInt(n -> n));
        IntSummaryStatistics stats = numbers.stream()
                                           .collect(Collectors.summarizingInt(n -> n));

```

```java
        System.out.println("   Average: " + average);
        System.out.println("   Sum: " + summing);
        System.out.println("   Statistics: " + stats);

        // 10. Joining strings
        System.out.println("\n10. Joining strings:");
        String joined = numbers.stream()
                              .map(String::valueOf)
                              .collect(Collectors.joining(", ", "[", "]"));
        System.out.println("   Joined: " + joined);
    }

    // ==================== 4. PRIMITIVE STREAMS ====================
    public static void primitiveStreams() {
        System.out.println("\n=== 4. PRIMITIVE STREAMS ===\n");

        // IntStream
        System.out.println("IntStream examples:");
        IntStream.range(1, 6).forEach(n -> System.out.print(n + " "));

        System.out.println("\n\nIntStream operations:");
        int sum = IntStream.rangeClosed(1, 100).sum();
        double avg = IntStream.rangeClosed(1, 100).average().orElse(0);
        OptionalInt max = IntStream.rangeClosed(1, 100).max();

        System.out.println("   Sum 1-100: " + sum);
        System.out.println("   Average 1-100: " + avg);
        System.out.println("   Max 1-100: " + max.orElse(0));

        // LongStream
        System.out.println("\nLongStream examples:");
        long factorial = LongStream.rangeClosed(1, 10)
                                  .reduce(1, (a, b) -> a * b);
        System.out.println("   10! = " + factorial);

        // DoubleStream
        System.out.println("\nDoubleStream examples:");
        double randomAvg = DoubleStream.generate(Math::random)
                                      .limit(1000)
                                      .average()
                                      .orElse(0);
        System.out.println("   Average of 1000 random numbers: " + randomAvg);

        // Converting between stream types
        System.out.println("\nConverting between stream types:");
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Convert to IntStream
        int intSum = numbers.stream()
                           .mapToInt(Integer::intValue)
                           .sum();
        System.out.println("   Sum using mapToInt: " + intSum);

        // Boxed stream (primitive to object)
        Stream<Integer> boxedStream = IntStream.range(1, 6).boxed();
        List<Integer> boxedList = boxedStream.collect(Collectors.toList());
        System.out.println("   Boxed list: " + boxedList);
    }

    // ==================== 5. PARALLEL STREAMS ====================
    public static void parallelStreams() {
        System.out.println("\n=== 5. PARALLEL STREAMS ===\n");

        List<Integer> numbers = IntStream.rangeClosed(1, 1000)
                                        .boxed()
                                        .collect(Collectors.toList());

        System.out.println("Processing " + numbers.size() + " numbers");

        // Sequential stream
        long startTime = System.currentTimeMillis();
        long seqCount = numbers.stream()
                              .filter(n -> isPrime(n))
                              .count();
        long seqTime = System.currentTimeMillis() - startTime;

        // Parallel stream
        startTime = System.currentTimeMillis();
        long parallelCount = numbers.parallelStream()
                                   .filter(n -> isPrime(n))
                                   .count();
        long parallelTime = System.currentTimeMillis() - startTime;

        System.out.println("\nPrime numbers between 1 and 1000:");
        System.out.println("   Sequential: " + seqCount + " primes in " + seqTime + "ms");
        System.out.println("   Parallel: " + parallelCount + " primes in " + parallelTime + "ms");
        System.out.println("   Speedup: " + (seqTime / (double) parallelTime) + "x");

        // When to use parallel streams
        System.out.println("\nWhen to use parallel streams:");
        System.out.println("       Large datasets");
        System.out.println("       Computationally intensive operations");
        System.out.println("       Stateless, independent operations");
        System.out.println("       Small datasets");
        System.out.println("       Stateful operations");
        System.out.println("       I/O bound operations");

        // Parallel stream considerations
        System.out.println("\nParallel stream considerations:");
        System.out.println("   - Order may not be preserved");
        System.out.println("   - Thread safety is important");
        System.out.println("   - Overhead for small tasks");
        System.out.println("   - Use parallel() judiciously");
    }

    private static boolean isPrime(int n) {
        if (n <= 1) return false;
```

```java
        for (int i = 2; i <= Math.sqrt(n); i++) {
            if (n % i == 0) return false;
        }
        return true;
    }

    // ==================== 6. REAL-WORLD EXAMPLES ====================
    public static void realWorldStreamExamples() {
        System.out.println("\n=== 6. REAL-WORLD STREAM EXAMPLES ===\n");

        List<Employee> employees = Arrays.asList(
            new Employee("Alice", "Engineering", 75000, 28),
            new Employee("Bob", "Sales", 60000, 35),
            new Employee("Charlie", "Engineering", 80000, 32),
            new Employee("Diana", "Marketing", 55000, 29),
            new Employee("Eve", "Engineering", 90000, 40),
            new Employee("Frank", "Sales", 65000, 45),
            new Employee("Grace", "HR", 50000, 30)
        );

        System.out.println("Employee Data:");
        employees.forEach(e -> System.out.println("   " + e));

        // Example 1: Find average salary by department
        System.out.println("\n1. Average salary by department:");
        Map<String, Double> avgSalaryByDept = employees.stream()
            .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.averagingDouble(Employee::getSalary)
            ));
        avgSalaryByDept.forEach((dept, avg) ->
            System.out.printf("   %s: $%.2f%n", dept, avg));

        // Example 2: Highest paid employee in each department
        System.out.println("\n2. Highest paid in each department:");
        Map<String, Optional<Employee>> topByDept = employees.stream()
            .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.maxBy(Comparator.comparing(Employee::getSalary))
            ));
        topByDept.forEach((dept, emp) ->
            System.out.printf("   %s: %s ($%.0f)%n",
                dept, emp.map(Employee::getName).orElse("None"),
                emp.map(Employee::getSalary).orElse(0.0)));

        // Example 3: Employees grouped by age range
        System.out.println("\n3. Employees by age range:");
        Map<String, List<Employee>> byAgeRange = employees.stream()
            .collect(Collectors.groupingBy(e -> {
                if (e.getAge() < 30) return "Under 30";
                else if (e.getAge() < 40) return "30-39";
                else return "40+";
            }));
        byAgeRange.forEach((range, empList) -> {
            System.out.println("   " + range + ": " +
                emplist.stream().map(Employee::getName).collect(Collectors.joining(", ")));
        });

        // Example 4: Total salary budget by department
        System.out.println("\n4. Total salary budget by department:");
        Map<String, Double> budgetByDept = employees.stream()
            .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.summingDouble(Employee::getSalary)
            ));
        budgetByDept.forEach((dept, total) ->
            System.out.printf("   %s: $%.0f%n", dept, total));

        // Example 5: Find engineering employees with salary > 70000
        System.out.println("\n5. Engineering employees earning > $70,000:");
        List<String> highEarners = employees.stream()
            .filter(e -> e.getDepartment().equals("Engineering"))
            .filter(e -> e.getSalary() > 70000)
            .map(Employee::getName)
            .sorted()
            .collect(Collectors.toList());
        System.out.println("   " + String.join(", ", highEarners));

        // Example 6: Statistics for engineering department
        System.out.println("\n6. Engineering department statistics:");
        DoubleSummaryStatistics engStats = employees.stream()
            .filter(e -> e.getDepartment().equals("Engineering"))
            .mapToDouble(Employee::getSalary)
            .summaryStatistics();
        System.out.printf("   Count: %d%n", engStats.getCount());
        System.out.printf("   Average: $%.2f%n", engStats.getAverage());
        System.out.printf("   Max: $%.2f%n", engStats.getMax());
        System.out.printf("   Min: $%.2f%n", engStats.getMin());
        System.out.printf("   Sum: $%.2f%n", engStats.getSum());
    }

    // ==================== 7. STREAM BEST PRACTICES ====================
    public static void streamBestPractices() {
        System.out.println("\n=== 7. STREAM BEST PRACTICES ===\n");

        System.out.println("1. Use Method References:");
        System.out.println("   Prefer: .map(String::toUpperCase)");
        System.out.println("   Over:   .map(s -> s.toUpperCase())");

        System.out.println("\n2. Avoid Side Effects:");
        System.out.println("   Don't modify external state in streams");
        System.out.println("   Use pure functions where possible");

        System.out.println("\n3. Choose Right Data Structure:");
        System.out.println("   ArrayList     .stream()");
        System.out.println("   Large datasets     .parallelStream()");

        System.out.println("\n4. Order Operations Efficiently:");
        System.out.println("   Filter early to reduce elements");
        System.out.println("   Expensive operations after filtering");

        System.out.println("\n5. Use Primitive Streams for Performance:");
        System.out.println("   Use IntStream, LongStream, DoubleStream");
        System.out.println("   Avoid boxing/unboxing overhead");

        System.out.println("\n6. Handle Optional Properly:");
        System.out.println("   Don't call .get() without checking .isPresent()");
        System.out.println("   Use .orElse(), .orElseGet(), .orElseThrow()");

        System.out.println("\n7. Limit Infinite Streams:");
        System.out.println("   Always use limit() with generate() or iterate()");

        System.out.println("\n8. Collect Once:");
        System.out.println("   Don't create multiple terminal operations");
        System.out.println("   Collect once and reuse");
    }

    // ==================== 8. COMPLETE EXAMPLE ====================
    public static void completeStreamExample() {
        System.out.println("\n=== 8. COMPLETE STREAM API EXAMPLE ===\n");

        // Create sample data
        List<Transaction> transactions = Arrays.asList(
            new Transaction(1001, "GROCERY", 150.50),
            new Transaction(1002, "ELECTRONICS", 1200.00),
            new Transaction(1003, "GROCERY", 75.25),
            new Transaction(1004, "CLOTHING", 200.00),
            new Transaction(1005, "ELECTRONICS", 850.00),
            new Transaction(1006, "GROCERY", 45.75),
            new Transaction(1007, "CLOTHING", 120.00),
            new Transaction(1008, "GROCERY", 300.00)
        );

        System.out.println("All Transactions:");
        transactions.forEach(t -> System.out.println("   " + t));

        // Business Logic using Stream API
        System.out.println("\n--- Analysis Results ---");

        // 1. Total amount of all transactions
        double totalAmount = transactions.stream()
            .mapToDouble(Transaction::getAmount)
            .sum();
        System.out.printf("1. Total amount: $%.2f%n", totalAmount);

        // 2. Average transaction amount
        double avgAmount = transactions.stream()
            .mapToDouble(Transaction::getAmount)
            .average()
            .orElse(0);
        System.out.printf("2. Average transaction: $%.2f%n", avgAmount);

        // 3. Highest transaction
        Optional<Transaction> highest = transactions.stream()
            .max(Comparator.comparing(Transaction::getAmount));
        highest.ifPresent(t ->
            System.out.printf("3. Highest transaction: %s - $%.2f%n",
                t.getType(), t.getAmount()));

        // 4. Group by type with totals
        System.out.println("\n4. Transactions by type:");
        Map<String, DoubleSummaryStatistics> byType = transactions.stream()
            .collect(Collectors.groupingBy(
                Transaction::getType,
                Collectors.summarizingDouble(Transaction::getAmount)
            ));

        byType.forEach((type, stats) -> {
            System.out.printf("  %s:%n", type);
            System.out.printf("    Count: %d%n", stats.getCount());
            System.out.printf("    Total: $%.2f%n", stats.getSum());
            System.out.printf("    Average: $%.2f%n", stats.getAverage());
            System.out.printf("    Max: $%.2f%n", stats.getMax());
        });

        // 5. Find all expensive transactions (> $500)
        System.out.println("\n5. Expensive transactions (> $500):");
        List<Transaction> expensive = transactions.stream()
            .filter(t -> t.getAmount() > 500)
            .sorted(Comparator.comparing(Transaction::getAmount).reversed())
            .collect(Collectors.toList());

        expensive.forEach(t ->
            System.out.printf("  ID %d: %s - $%.2f%n",
                t.getId(), t.getType(), t.getAmount()));

        // 6. Transaction IDs as comma-separated string
        String transactionIds = transactions.stream()
            .map(t -> String.valueOf(t.getId()))
            .collect(Collectors.joining(", ", "[", "]"));
        System.out.println("\n6. Transaction IDs: " + transactionIds);
    }

    // ==================== MAIN METHOD ====================
    public static void main(String[] args) {
        System.out.println("=== STREAM API - COMPLETE GUIDE ===\n");

        System.out.println("Stream API provides a functional approach to");
        System.out.println("processing collections of data in Java.\n");

        // 1. Stream Basics
        streamBasics();

        // 2. Intermediate Operations
        intermediateOperations();

        // 3. Terminal Operations
        terminalOperations();

        // 4. Primitive Streams
        primitiveStreams();

        // 5. Parallel Streams
        parallelStreams();

        // 6. Real-world Examples
        realWorldStreamExamples();

        // 7. Best Practices
        streamBestPractices();

        // 8. Complete Example
        completeStreamExample();

        System.out.println("\n=== STREAM API BENEFITS ===");
        System.out.println("1. Declarative: Say what you want, not how");
        System.out.println("2. Composable: Chain operations easily");
        System.out.println("3. Parallelizable: Easy parallel processing");
        System.out.println("4. Lazy Evaluation: Efficient execution");
        System.out.println("5. Functional: Encourages pure functions");

        System.out.println("\n=== COMMON MISTAKES ===");
        System.out.println("1. Reusing streams (they're one-time use)");
        System.out.println("2. Forgetting terminal operations (nothing happens)");
        System.out.println("3. Modifying source collection during stream ops");
        System.out.println("4. Ignoring ordering in parallel streams");
        System.out.println("5. Not handling Optional properly");

        System.out.println("\n=== WHEN TO USE STREAMS ===");
        System.out.println("   Processing collections of data");
        System.out.println("   Transformations and filtering");
        System.out.println("   Aggregations and summaries");
        System.out.println("   Parallel processing needs");
        System.out.println("   Functional programming style");

        System.out.println("\n=== WHEN NOT TO USE STREAMS ===");
        System.out.println("   Simple loops (traditional for-loop might be clearer)");
        System.out.println("   Complex control flow (break, continue, return)");
        System.out.println("   Stateful operations");
        System.out.println("   Performance-critical small loops");
    }

    // ==================== SUPPORTING CLASSES ====================
    static class Employee {
        private String name;
        private String department;
        private double salary;
        private int age;

        public Employee(String name, String department, double salary, int age) {
            this.name = name;
            this.department = department;
            this.salary = salary;
            this.age = age;
        }

        public String getName() { return name; }
        public String getDepartment() { return department; }
        public double getSalary() { return salary; }
        public int getAge() { return age; }

        @Override
        public String toString() {
            return String.format("%-10s %-15s $%8.2f age:%2d",
                name, department, salary, age);
        }
    }

    static class Transaction {
        private int id;
        private String type;
        private double amount;

        public Transaction(int id, String type, double amount) {
            this.id = id;
            this.type = type;
            this.amount = amount;
        }

        public int getId() { return id; }
        public String getType() { return type; }
        public double getAmount() { return amount; }

        @Override
        public String toString() {
            return String.format("ID:%4d %-12s $%7.2f", id, type, amount);
        }
    }
}
```

Listing 4: Stream API - Complete Guide Program

5 Practical Integration Example

```java
/* ******************************************************************************
```

```java
 * PROGRAM: JDBCLambdaStreamIntegration.java
 *
 * PURPOSE: This program demonstrates the integration of JDBC, Lambda Expressions,
 *          and Stream API to create modern, efficient database applications.
 *          It shows how functional programming concepts can enhance traditional
 *          database operations.
 *
 * PROBLEM SOLVED: Traditional database programming in Java suffers from:
 *                 1. Verbose boilerplate code for CRUD operations
 *                 2. Complex data transformation logic
 *                 3. Difficulty in composing database operations
 *                 4. Inefficient data processing patterns
 *
 *                 This integration solves these by:
 *                 1. Using lambdas for concise database operations
 *                 2. Applying streams for efficient data processing
 *                 3. Composing operations in a functional style
 *                 4. Enabling parallel database processing
 *
 * WHY NECESSARY: Modern enterprise applications require:
 *                1. Clean, maintainable database code
 *                2. Efficient data processing pipelines
 *                3. Functional programming patterns for scalability
 *                4. Integration of modern Java features with legacy systems
 *                5. Industry-standard patterns for data access
 *
 * KEY CONCEPTS DEMONSTRATED:
 * - JDBC operations enhanced with lambdas
 * - Stream processing of ResultSet data
 * - Functional composition of database queries
 * - Parallel processing of database results
 * - Real-world employee management system
 * - Best practices for integrated database programming
 *******************************************************************************/

import java.sql.*;
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

public class JDBCLambdaStreamIntegration {

    private static final String URL = "jdbc:mysql://localhost:3306/company";
    private static final String USER = "root";
    private static final String PASSWORD = "password";

    // ==================== DATABASE SETUP ====================
    public static void setupDatabase() {
        String createTableSQL =
            "CREATE TABLE IF NOT EXISTS employees (" +
            "id INT PRIMARY KEY AUTO_INCREMENT, " +
            "name VARCHAR(100) NOT NULL, " +
            "department VARCHAR(50), " +
            "salary DECIMAL(10,2), " +
            "age INT, " +
            "hire_date DATE, " +
            "active BOOLEAN DEFAULT true)";

        String insertDataSQL =
            "INSERT INTO employees (name, department, salary, age, hire_date) VALUES " +
            "(’John Smith’, ’Engineering’, 75000.00, 30, ’2020-01-15’), " +
            "(’Alice Johnson’, ’Sales’, 65000.00, 28, ’2021-03-10’), " +
            "(’Bob Williams’, ’Engineering’, 82000.00, 35, ’2019-05-20’), " +
            "(’Carol Davis’, ’Marketing’, 55000.00, 32, ’2022-07-05’), " +
            "(’David Brown’, ’Engineering’, 90000.00, 40, ’2018-11-30’), " +
            "(’Eve Miller’, ’Sales’, 70000.00, 29, ’2021-09-15’), " +
            "(’Frank Wilson’, ’HR’, 50000.00, 45, ’2020-12-01’), " +
            "(’Grace Moore’, ’Engineering’, 78000.00, 33, ’2022-02-28’), " +
            "(’Henry Taylor’, ’Marketing’, 60000.00, 38, ’2021-06-10’), " +
            "(’Irene Anderson’, ’Sales’, 72000.00, 31, ’2023-01-05’)";

        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
             Statement stmt = conn.createStatement()) {

            // Create table
            stmt.executeUpdate(createTableSQL);
            System.out.println("Table created/verified");

            // Clear existing data
            stmt.executeUpdate("DELETE FROM employees");

            // Insert sample data
            stmt.executeUpdate(insertDataSQL);
            System.out.println("Sample data inserted");

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

    // ==================== EMPLOYEE CLASS ====================
    static class Employee {
        private int id;
        private String name;
        private String department;
        private double salary;
        private int age;
        private Date hireDate;
        private boolean active;

        public Employee(int id, String name, String department,
                       double salary, int age, Date hireDate, boolean active) {
            this.id = id;
            this.name = name;
            this.department = department;
            this.salary = salary;
            this.age = age;
            this.hireDate = hireDate;
            this.active = active;
        }

        // Getters
        public int getId() { return id; }
        public String getName() { return name; }
        public String getDepartment() { return department; }
        public double getSalary() { return salary; }
        public int getAge() { return age; }
        public Date getHireDate() { return hireDate; }
        public boolean isActive() { return active; }

        @Override
        public String toString() {
            return String.format("%2d %-15s %-12s $%8.2f %3dyrs %tF %s",
                id, name, department, salary, age, hireDate, active ? "Active" : "Inactive");
        }
    }

    // ==================== DATABASE OPERATIONS WITH LAMBDAS ====================
    public static List<Employee> getAllEmployees() {
        List<Employee> employees = new ArrayList<>();

        String sql = "SELECT * FROM employees";

        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
             Statement stmt = conn.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {

            while (rs.next()) {
                Employee emp = new Employee(
                    rs.getInt("id"),
                    rs.getString("name"),
                    rs.getString("department"),
                    rs.getDouble("salary"),
                    rs.getInt("age"),
                    rs.getDate("hire_date"),
                    rs.getBoolean("active")
                );
                employees.add(emp);
            }

        } catch (SQLException e) {
            System.out.println("Error fetching employees: " + e.getMessage());
        }

        return employees;
    }

    public static List<Employee> getEmployeesByCondition(Predicate<Employee> condition) {
        return getAllEmployees().stream()
                               .filter(condition)
                               .collect(Collectors.toList());
    }

    public static double calculateDepartmentSalary(String department,
                                                  Function<Employee, Double> mapper) {
        return getAllEmployees().stream()
                               .filter(e -> e.getDepartment().equals(department))
                               .map(mapper)
                               .reduce(0.0, Double::sum);
    }

    // ==================== STREAM OPERATIONS ON DATABASE DATA ====================
    public static void performAnalysis() {
        System.out.println("\n=== EMPLOYEE DATA ANALYSIS USING STREAMS ===\n");

        List<Employee> employees = getAllEmployees();

        System.out.println("All Employees:");
        employees.forEach(System.out::println);

        // 1. Group employees by department
        System.out.println("\n1. Employees grouped by department:");
        Map<String, List<Employee>> byDepartment = employees.stream()
            .collect(Collectors.groupingBy(Employee::getDepartment));

        byDepartment.forEach((dept, empList) -> {
            System.out.println("\n   " + dept + " Department:");
            empList.forEach(e -> System.out.println("     " + e.getName()));
        });

        // 2. Average salary by department
        System.out.println("\n2. Average salary by department:");
        Map<String, Double> avgSalaryByDept = employees.stream()
            .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.averagingDouble(Employee::getSalary)
            ));

        avgSalaryByDept.forEach((dept, avg) ->
            System.out.printf("   %-12s: $%.2f%n", dept, avg));

        // 3. Top 3 highest paid employees
```

```java
        System.out.println("\n3. Top 3 highest paid employees:");
        employees.stream()
            .sorted(Comparator.comparingDouble(Employee::getSalary).reversed())
            .limit(3)
            .forEach(e -> System.out.printf("   %-15s: $%.2f%n", e.getName(), e.getSalary()));

        // 4. Employees by age group
        System.out.println("\n4. Employees by age group:");
        Map<String, List<Employee>> byAgeGroup = employees.stream()
            .collect(Collectors.groupingBy(e -> {
                if (e.getAge() < 30) return "Under 30";
                else if (e.getAge() < 40) return "30-39";
                else return "40+";
            }));

        byAgeGroup.forEach((group, empList) -> {
            System.out.printf("   %s (%d employees):%n", group, empList.size());
            empList.forEach(e -> System.out.printf("     %s (%d)%n", e.getName(), e.getAge()));
        });

        // 5. Department with highest total salary
        System.out.println("\n5. Department salary statistics:");
        Map<String, Double> totalSalaryByDept = employees.stream()
            .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.summingDouble(Employee::getSalary)
            ));

        totalSalaryByDept.entrySet().stream()
            .sorted(Map.Entry.<String, Double>comparingByValue().reversed())
            .forEach(entry ->
                System.out.printf("   %-12s: $%.2f%n", entry.getKey(), entry.getValue()));

        // 6. Find employees with salary above department average
        System.out.println("\n6. Employees earning above their department average:");
        employees.stream()
            .collect(Collectors.groupingBy(Employee::getDepartment))
            .forEach((dept, empList) -> {
                double deptAvg = empList.stream()
                    .mapToDouble(Employee::getSalary)
                    .average()
                    .orElse(0);

                System.out.printf("\n   %s (Average: $%.2f):%n", dept, deptAvg);
                empList.stream()
                    .filter(e -> e.getSalary() > deptAvg)
                    .forEach(e -> System.out.printf("     %-15s: $%.2f%n", e.getName(), e.getSalary()));
            });
    }

    // ==================== DATABASE UPDATES WITH LAMBDA ====================
    public static void updateSalaries(Function<Employee, Double> salaryCalculator) {
        String updateSQL = "UPDATE employees SET salary = ? WHERE id = ?";

        try (Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
             PreparedStatement pstmt = conn.prepareStatement(updateSQL)) {

            List<Employee> employees = getAllEmployees();

            employees.forEach(emp -> {
                try {
                    double newSalary = salaryCalculator.apply(emp);
                    pstmt.setDouble(1, newSalary);
                    pstmt.setInt(2, emp.getId());
                    pstmt.addBatch();
                } catch (SQLException e) {
                    System.out.println("Error updating salary for " + emp.getName() + ": " + e.getMessage());
                }
            });

            int[] updateCounts = pstmt.executeBatch();
            System.out.println("Updated " + Arrays.stream(updateCounts).sum() + " salaries");

        } catch (SQLException e) {
            System.out.println("Error updating salaries: " + e.getMessage());
        }
    }

    // ==================== COMPLEX QUERY WITH STREAM PROCESSING ====================
    public static void complexEmployeeReport() {
        System.out.println("\n=== COMPLEX EMPLOYEE REPORT ===\n");

        List<Employee> employees = getAllEmployees();

        // Create comprehensive report
        Map<String, Map<String, Object>> report = employees.stream()
            .collect(Collectors.groupingBy(
                Employee::getDepartment,
                Collectors.collectingAndThen(
                    Collectors.toList(),
                    empList -> {
                        Map<String, Object> deptStats = new HashMap<>();
                        deptStats.put("count", empList.size());
                        deptStats.put("totalSalary",
                            empList.stream().mapToDouble(Employee::getSalary).sum());
                        deptStats.put("averageSalary",
                            empList.stream().mapToDouble(Employee::getSalary).average().orElse(0));
                        deptStats.put("averageAge",
                            empList.stream().mapToInt(Employee::getAge).average().orElse(0));
                        deptStats.put("highestPaid",
                            empList.stream().max(Comparator.comparingDouble(Employee::getSalary))
                                    .map(Employee::getName).orElse("None"));
                        deptStats.put("employees",
                            empList.stream().map(Employee::getName).collect(Collectors.toList()));
                        return deptStats;
                    }
                )
            ));

        // Print report
        report.forEach((dept, stats) -> {
            System.out.println("=".repeat(50));
            System.out.println("DEPARTMENT: " + dept);
            System.out.println("=".repeat(50));
            System.out.printf("Employee Count: %d%n", stats.get("count"));
            System.out.printf("Total Salary: $%.2f%n", stats.get("totalSalary"));
            System.out.printf("Average Salary: $%.2f%n", stats.get("averageSalary"));
            System.out.printf("Average Age: %.1f years%n", stats.get("averageAge"));
            System.out.printf("Highest Paid: %s%n", stats.get("highestPaid"));
            System.out.println("Employees: " + String.join(", ", (List<String>) stats.get("employees")));
            System.out.println();
        });

        // Overall statistics
        System.out.println("\n=== OVERALL COMPANY STATISTICS ===");
        System.out.printf("Total Employees: %d%n", employees.size());
        System.out.printf("Total Salary Budget: $%.2f%n",
            employees.stream().mapToDouble(Employee::getSalary).sum());
        System.out.printf("Average Company Salary: $%.2f%n",
            employees.stream().mapToDouble(Employee::getSalary).average().orElse(0));
        System.out.printf("Average Employee Age: %.1f years%n",
            employees.stream().mapToInt(Employee::getAge).average().orElse(0));
    }

    // ==================== MAIN METHOD ====================
    public static void main(String[] args) {
        System.out.println("=== JDBC WITH LAMBDA AND STREAM INTEGRATION ===\n");

        // Setup database
        setupDatabase();

        // Get all employees using traditional JDBC
        System.out.println("\nFetching employees from database...");
        List<Employee> allEmployees = getAllEmployees();
        System.out.println("Total employees: " + allEmployees.size());

        // Example 1: Using lambda predicates to filter employees
        System.out.println("\n=== EXAMPLE 1: FILTERING WITH LAMBDA PREDICATES ===");

        Predicate<Employee> highSalary = e -> e.getSalary() > 70000;
        Predicate<Employee> engineeringDept = e -> e.getDepartment().equals("Engineering");
        Predicate<Employee> youngEmployee = e -> e.getAge() < 35;

        System.out.println("\nHigh salary employees (> $70,000):");
        getEmployeesByCondition(highSalary)
            .forEach(e -> System.out.println("  " + e.getName() + ": $" + e.getSalary()));

        System.out.println("\nEngineering department employees:");
        getEmployeesByCondition(engineeringDept)
            .forEach(e -> System.out.println("  " + e.getName()));

        System.out.println("\nYoung engineering employees with high salary:");
        getEmployeesByCondition(engineeringDept.and(highSalary).and(youngEmployee))
            .forEach(e -> System.out.println("  " + e.getName() + " (age: " + e.getAge() +
                                           ", salary: $" + e.getSalary() + ")"));

        // Example 2: Using lambda functions for calculations
        System.out.println("\n=== EXAMPLE 2: CALCULATIONS WITH LAMBDA FUNCTIONS ===");

        Function<Employee, Double> baseSalary = Employee::getSalary;
        Function<Employee, Double> salaryWithBonus = e -> e.getSalary() * 1.1; // 10% bonus
        Function<Employee, Double> salaryWithRaise = e -> e.getSalary() + 5000; // $5000 raise

        System.out.println("\nTotal base salary for Engineering: $" +
                          calculateDepartmentSalary("Engineering", baseSalary));
        System.out.println("Total salary with 10% bonus for Engineering: $" +
                          calculateDepartmentSalary("Engineering", salaryWithBonus));
        System.out.println("Total salary with $5000 raise for Sales: $" +
                          calculateDepartmentSalary("Sales", salaryWithRaise));

        // Example 3: Stream analysis
        performAnalysis();

        // Example 4: Database updates with lambda
        System.out.println("\n=== EXAMPLE 4: DATABASE UPDATES WITH LAMBDA ===");
        System.out.println("Giving 5% raise to Engineering department...");

        Function<Employee, Double> engineeringRaise = e ->
            e.getDepartment().equals("Engineering") ? e.getSalary() * 1.05 : e.getSalary();

        updateSalaries(engineeringRaise);

        // Example 5: Complex report
        complexEmployeeReport();

        System.out.println("\n=== INTEGRATION BENEFITS ===");
        System.out.println("1. Concise database operations with lambdas");
        System.out.println("2. Powerful data processing with Stream API");
        System.out.println("3. Functional programming style for data transformation");
        System.out.println("4. Type safety and compile-time checking");
        System.out.println("5. Easy parallelization of database processing");

        System.out.println("\n=== BEST PRACTICES ===");
        System.out.println("1. Use PreparedStatement with lambda parameters");
        System.out.println("2. Process large datasets with streams (lazy evaluation)");
        System.out.println("3. Combine database filtering with stream filtering wisely");
        System.out.println("4. Use transactions for batch updates");
        System.out.println("5. Handle exceptions properly in lambda expressions");
    }
}
```

Listing 5: JDBC with Lambda and Stream Integration Program

6 Summary and Key Takeaways

Unit V Summary

Key Learnings from Unit V:

- **JDBC Architecture**: Understand the 4-layer model and how Java applications communicate with databases
- **Database Connectivity**: Master the 6 essential steps for database operations
- **Lambda Expressions**: Transform verbose anonymous classes into concise functional code
- **Stream API**: Process data declaratively with built-in parallelization support
- **Integration**: Combine JDBC with modern Java features for efficient database applications

Industry Relevance

Why These Skills Matter in Industry:

1. **Enterprise Applications**: 90% of Java applications require database connectivity
2. **Modern Codebases**: Lambda and Stream API are now industry standards
3. **Performance**: Proper JDBC usage prevents resource leaks and improves performance
4. **Maintainability**: Functional programming leads to cleaner, more maintainable code
5. **Scalability**: Stream API enables easy parallel processing for big data

6.1 Assessment Questions

1. Explain the 4-layer JDBC architecture with a diagram.
2. Write a Java program that demonstrates all 6 steps of JDBC connectivity.
3. Convert the following anonymous class to a lambda expression:

```java
new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello World");
    }
}).start();
```

4. Given a list of integers, use Stream API to: - Filter even numbers - Square each number - Find the sum

5. Create a PreparedStatement example that prevents SQL injection.
6. Explain the difference between map() and flatMap() operations in Stream API.
7. Write a lambda expression that sorts a list of employees by salary in descending order.
8. Demonstrate try-with-resources for JDBC connections.
9. What are method references? Provide examples of all four types.
10. Create a complete employee management system using JDBC, Lambda, and Stream API.

6.2 Further Reading

- **Oracle JDBC Documentation**: Complete guide to JDBC API
- **Java 8 in Action**: Comprehensive coverage of Lambdas and Streams
- **Effective Java (3rd Edition)**: Best practices including modern Java features
- **Database Programming with JDBC and Java**: O'Reilly guide to database programming
- **Java Performance**: Optimizing database and stream operations

JDBCArchitecture Program Output

```
=== JDBC ARCHITECTURE AND COMPONENTS ===

=== JDBC ARCHITECTURE ===

1. Java Application Layer
   - Your Java program using JDBC API
   - Uses interfaces: Connection, Statement, ResultSet

2. JDBC API Layer
   - java.sql and javax.sql packages
   - Provides standard interfaces
   - DriverManager: Manages database drivers

3. JDBC Driver Layer
   - Database-specific implementations
   - Types: Type 1, 2, 3, 4
   - Converts JDBC calls to database-specific calls

4. Database Layer
   - Actual RDBMS (MySQL, Oracle, PostgreSQL, etc.)
   - Stores and manages data

=== JDBC DRIVER TYPES ===
Type 1: JDBC-ODBC Bridge Driver (Deprecated)
Type 2: Native-API Driver (Part Java, Part Native)
Type 3: Network Protocol Driver (Pure Java)
Type 4: Thin Driver (Pure Java, Direct) - Most Common

=== JDBC CORE COMPONENTS ===

1. DriverManager:
   - Manages database drivers
   - Establishes database connections
   - Methods: getConnection(), registerDriver()

2. Connection:
   - Represents a connection to database
   - Creates Statement objects
   - Manages transactions
   - Methods: createStatement(), prepareStatement()

3. Statement:
   - Executes SQL queries
   - Types: Statement, PreparedStatement, CallableStatement
   - Methods: executeQuery(), executeUpdate()

4. ResultSet:
   - Contains query results
   - Navigable cursor through rows
   - Methods: next(), getString(), getInt()

5. SQLException:
   - Checked exception for database errors
   - Provides error codes and messages

=== JDBC WORKFLOW ===
Step 1: Load and Register Driver
   Class.forName("com.mysql.cj.jdbc.Driver");

Step 2: Establish Connection
   Connection conn = DriverManager.getConnection(url, user, pass);

Step 3: Create Statement
   Statement stmt = conn.createStatement();

Step 4: Execute Query
   ResultSet rs = stmt.executeQuery("SELECT * FROM table");

Step 5: Process Results
   while(rs.next()) { /* process each row */ }
```

```
Step 6: Close Resources
   rs.close(); stmt.close(); conn.close();
```

JDBCConnectionSteps Program Output

```
=== JDBC CONNECTION STEPS - COMPLETE GUIDE ===

This example demonstrates all 6 steps of JDBC:
1. Load Database Driver
2. Establish Connection
3. Create Statement
4. Execute Queries
5. Process ResultSet
6. Close Resources

=== STEP 1: LOAD DATABASE DRIVER ===

Method 1: Using Class.forName()
MySQL JDBC Driver loaded successfully

Alternative: Modern JDBC 4.0+ auto-loads drivers
from META-INF/services/java.sql.Driver

=== STEP 2: ESTABLISH DATABASE CONNECTION ===

Method 1: Basic connection
Method 2: Connection with Properties
Method 3: Connection with URL parameters

 Connection established successfully!
Connection URL: jdbc:mysql://localhost:3306/university
Database: university
Auto Commit: true
Transaction Isolation: 2

=== STEP 3: CREATE STATEMENT OBJECTS ===

1. Regular Statement:
   Created: Statement for static SQL queries

2. PreparedStatement:
   Created: PreparedStatement for parameterized queries
   SQL: SELECT * FROM students WHERE age > ? AND department = ?
   Benefits: Precompiled, prevents SQL injection

3. CallableStatement:
   Created: CallableStatement for stored procedures
```

```
4. Statement with ResultSet type:
   Created: Scrollable, read-only ResultSet

=== STEP 4: EXECUTE SQL QUERIES ===

Test table ’employees’ created with sample data

1. SELECT Query (executeQuery):
   Executed: SELECT * FROM employees

   Results:
   ID: 1, Name: John Smith, Salary: 50000.0
   ID: 2, Name: Jane Doe, Salary: 60000.0
   ID: 3, Name: Mike Johnson, Salary: 75000.0

2. INSERT Query (executeUpdate):
   Executed: INSERT INTO employees (name, salary, department) VALUES (’John Doe’, 5
   Rows inserted: 1

3. UPDATE Query:
   Executed: UPDATE employees SET salary = salary * 1.1 WHERE department = ’IT’
   Rows updated: 1

4. DELETE Query:
   Executed: DELETE FROM employees WHERE name = ’John Doe’
   Rows deleted: 1

5. PreparedStatement Example:
   Inserted: Alice Smith (1 row)
   Inserted: Bob Johnson (1 row)
   Inserted: Carol Williams (1 row)

=== STEP 5: PROCESS RESULTSET ===

ResultSet Metadata:
Number of columns: 5
  Column 1: id (INT)
  Column 2: name (VARCHAR)
  Column 3: salary (DECIMAL)
  Column 4: department (VARCHAR)
  Column 5: hire_date (DATE)

Processing ResultSet:

1. Using column names:
   ID: 1, Name: John Smith, Salary: $50000.0, Dept: IT
   ID: 2, Name: Jane Doe, Salary: $60000.0, Dept: HR
   ID: 3, Name: Mike Johnson, Salary: $75000.0, Dept: Engineering
   ID: 4, Name: Alice Smith, Salary: $60000.0, Dept: HR
   ID: 5, Name: Bob Johnson, Salary: $75000.0, Dept: Engineering
   ID: 6, Name: Carol Williams, Salary: $55000.0, Dept: Marketing

2. Using column indexes:
   ID: 1, Name: John Smith, Salary: $50000.0, Dept: IT
   ID: 2, Name: Jane Doe, Salary: $60000.0, Dept: HR
   ID: 3, Name: Mike Johnson, Salary: $75000.0, Dept: Engineering
   ID: 4, Name: Alice Smith, Salary: $60000.0, Dept: HR
   ID: 5, Name: Bob Johnson, Salary: $75000.0, Dept: Engineering
   ID: 6, Name: Carol Williams, Salary: $55000.0, Dept: Marketing

3. Scrollable ResultSet Navigation:
   Last row - ID: 6
   First row - ID: 1
   Row 2 - Name: Jane Doe

4. Different data type getters:
   getObject(): John Smith
   getString(): John Smith
   getInt(): 1
   getDouble(): 50000.0
   getDate(): 2024-01-15

=== STEP 6: CLOSE RESOURCES PROPERLY ===

Method 1: Traditional try-catch-finally

Method 2: Try-with-resources (Recommended)
   Resources automatically closed

 Main connection closed successfully

============================================================
DEMONSTRATING COMPLETE JDBC WORKFLOW
============================================================
1. Connection established
2. Table created/verified
3. Data inserted

4. Query Results:
ID      Name            Price   Quantity
----------------------------------------
1       Laptop          $999.99 10
2       Mouse           $25.50  100
```

```
3       Keyboard        $75.00  50
4       Monitor         $299.99 20
----------------------------------------
Total inventory value: $29299.80

5. Prices updated for 2 products

6. Transaction Example:
   Transaction committed successfully

=== JDBC BEST PRACTICES ===
1. Use PreparedStatement to prevent SQL injection
2. Always close resources in finally block or use try-with-resources
3. Use connection pooling for production applications
4. Handle SQLException properly with specific error messages
5. Use transactions for multiple related operations
6. Validate and sanitize user input before database operations
7. Use appropriate data types (getInt for INT, getString for VARCHAR)
8. Limit ResultSet size for large queries (use LIMIT clause)
9. Use batch updates for multiple insert/update operations
10. Test with different database configurations
```

LambdaExpressionsGuide Program Output

```
=== LAMBDA EXPRESSIONS - COMPLETE GUIDE ===

Lambda Expressions introduce functional programming
features to Java, enabling concise, readable code.

=== 1. BASIC LAMBDA SYNTAX ===

Before Java 8 - Anonymous Class:
   Running with anonymous class

Java 8+ - Lambda Expression:
   Running with lambda

Lambda with Parameters:
   Addition: 5 + 3 = 8
   Multiplication: 5 * 3 = 15

Lambda with Explicit Types:
   Subtraction: 10 - 4 = 6

Lambda with Multiple Statements:
   Complex calculation (5,3): 23
```

```
=== 2. BUILT-IN FUNCTIONAL INTERFACES ===

1. Predicate<T> - Tests a condition:
   Is 10 even? true
   Is -5 positive? false
   Is empty string? false
   Is 6 even AND positive? true

2. Function<T, R> - Transforms input to output:
   Length of ’Hello’: 5
   123 as string: 123
   Shout ’hello’: HELLO!

3. Consumer<T> - Consumes input, returns nothing:
   Printing with consumer: Hello Consumer!
   Square of 5: 25
   Chained consumer: test
   Uppercase: TEST

4. Supplier<T> - Provides values:
   Random number: 0.456789123
   Greeting: Hello World!
   New list: []

5. UnaryOperator<T> - Function with same input/output type:
   Square of 7: 49
   Reverse ’lambda’: adbmal

6. BinaryOperator<T> - Two inputs, returns same type:
   Max of 10 and 20: 20
   Concatenate: Hello World

=== 3. LAMBDA WITH COLLECTIONS ===

Before Java 8 - External iteration:
   Apple

Java 8 - Internal iteration with forEach:
   Apple
   Banana
   Cherry
   Date
   Elderberry

Using Method Reference:
Apple
Banana
```

```
Cherry
Date
Elderberry

Filtering fruits starting with ’C’:
Cherry

Transforming to uppercase:
APPLE
BANANA
CHERRY
DATE
ELDERBERRY

Sorted by length:
   Date
   Apple
   Banana
   Cherry
   Elderberry

Sorted by length then alphabetically:
   Date
   Apple
   Banana
   Cherry
   Elderberry

=== 4. METHOD REFERENCES ===

1. Static Method Reference:
Alice
Bob
Charlie
David

Using custom static method:
   >>> Alice
   >>> Bob
   >>> Charlie
   >>> David

2. Instance Method Reference (specific instance):
   Name: Alice
   Name: Bob
   Name: Charlie
   Name: David
```

```
3. Instance Method Reference (arbitrary instance):
   ALICE
   BOB
   CHARLIE
   DAVID

4. Constructor Reference:
   New list: [New Element]

Constructor reference with parameters:
   Parsed ’123’: 123

=== 5. REAL-WORLD LAMBDA EXAMPLES ===

Example 1: Event Handlers in GUI
// Old way:
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});

// New way with lambda:
button.addActionListener(e -> System.out.println("Button clicked!"));

Example 2: Thread Creation
// Old way:
new Thread(new Runnable() {
    public void run() {
        System.out.println("Thread running");
    }
}).start();

// New way with lambda:
new Thread(() -> System.out.println("Thread running")).start();

Example 3: Sorting Employees

Employees sorted by salary (descending):
   Charlie (Engineering): $80000
   Alice (Engineering): $75000
   Bob (Sales): $60000
   Diana (Marketing): $55000

Example 4: Filtering high-salary Engineering employees:
   Alice
   Charlie

Example 5: Custom Validator
   Valid email ’test@example.com’? true
   Valid age 25? true

=== 6. VARIABLE CAPTURE IN LAMBDAS ===

Using effectively final variables:
   Item: Book
   Product: Book
   Item: Pen
   Product: Pen
   Item: Notebook
   Product: Notebook

Instance and static variable capture:
   Instance: 101, Static: 1
   Instance: 102, Static: 2
   Instance: 103, Static: 3

=== 7. LAMBDA BEST PRACTICES ===

1. Keep Lambdas Short and Simple:
   Good: names.stream().filter(n -> n.length() > 3)
   Bad: Complex logic in lambda - extract to method

2. Use Method References When Possible:
   Instead of: s -> s.toUpperCase()
   Use: String::toUpperCase

3. Avoid Side Effects:
   Pure functions are better than mutating external state

4. Use Descriptive Parameter Names:
   Good: (person, department) -> ...
   Bad: (p, d) -> ...

5. Consider Type Inference:
   Let compiler infer types when clear
   (a, b) -> a + b  instead of  (int a, int b) -> a + b

6. Chain Operations Readably:
   list.stream()
       .filter(...)
       .map(...)
       .collect(...);
```

```
=== KEY BENEFITS OF LAMBDA EXPRESSIONS ===
1. Conciseness: Less boilerplate code
2. Readability: More expressive code
3. Functional Programming: Support for FP paradigms
4. Parallelism: Easier parallel processing
5. API Design: Enables fluent APIs

=== COMMON PITFALLS ===
1. Overusing lambdas for complex logic
2. Not understanding variable capture rules
3. Ignoring exception handling in lambdas
4. Performance overhead in some cases
5. Debugging can be more challenging
```

StreamAPIGuide Program Output

```
=== STREAM API - COMPLETE GUIDE ===

Stream API provides a functional approach to
processing collections of data in Java.

=== 1. STREAM API BASICS ===

Source Collection: [Apple, Banana, Cherry, Date, Elderberry]

1. Different ways to create streams:
   From Collection: fruits.stream()
   From Array: Arrays.stream(array)
   Using Stream.of(): Stream.of("A", "B", "C")
   Infinite stream: Stream.iterate(1, n -> n + 1).limit(5)
   Generated stream: Stream.generate(Math::random).limit(3)

2. Stream Operations Pipeline:
   Source → Intermediate Operations → Terminal Operation
   Example: Count fruits with length > 5 = 2

3. Stream Characteristics:
   - Not a data structure, carries values from source
   - Functional in nature (doesn't modify source)
   - Lazily evaluated (only when terminal operation called)
   - Possibly unbounded
   - Consumable (can be traversed only once)

=== 2. INTERMEDIATE OPERATIONS ===

Original numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
1. filter() - Selects elements:
   Even numbers: [2, 4, 6, 8, 10]

2. map() - Transforms elements:
   Squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
   Number strings: [Number: 1, Number: 2, Number: 3, Number: 4, Number: 5, Number:

3. flatMap() - Flattens nested structures:
   Nested: [[A, B, C], [D, E, F], [G, H, I]]
   Flattened: [A, B, C, D, E, F, G, H, I]

4. distinct() - Removes duplicates:
   With duplicates: [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
   Distinct: [1, 2, 3, 4]

5. sorted() - Sorts elements:
   Original: [5, 3, 8, 1, 9, 2]
   Sorted ascending: [1, 2, 3, 5, 8, 9]
   Sorted descending: [9, 8, 5, 3, 2, 1]

6. peek() - For debugging:
   Before filter: 1 Before filter: 2 Before filter: 3 Before filter: 4 Before filte
   Before filter: 7 After filter: 7
   Before filter: 8 After filter: 8
   Before filter: 9 After filter: 9
   Before filter: 10 After filter: 10
   Result: [6, 7, 8, 9, 10]

7. limit() and skip():
   Skip 3, limit 4: [4, 5, 6, 7]

8. Chained Intermediate Operations:
   Chained result: [#6, #10, #14, #18]

=== 3. TERMINAL OPERATIONS ===

Numbers: [10, 20, 30, 40, 50]

1. forEach() - Performs action:
   Printing numbers: 10 20 30 40 50

2. collect() - Accumulates elements:
   As Set: [50, 20, 40, 10, 30]

3. toArray() - Converts to array:
   Array: [10, 20, 30, 40, 50]
```

```
4. reduce() - Combines elements:
   Sum: 150
   Product: 12000000
   Sum with identity: 150

5. min() and max():
   Min: 10
   Max: 50

6. count():
   Count: 5

7. anyMatch(), allMatch(), noneMatch():
   Any > 25: true
   All > 5: true
   None > 100: true

8. findFirst() and findAny():
   First: 10
   Any: 10

9. Advanced Collectors:
   Average: 30.0
   Sum: 150
   Statistics: IntSummaryStatistics{count=5, sum=150, min=10, average=30.000000, ma

10. Joining strings:
   Joined: [10, 20, 30, 40, 50]

=== 4. PRIMITIVE STREAMS ===

IntStream examples:
1 2 3 4 5

IntStream operations:
   Sum 1-100: 5050
   Average 1-100: 50.5
   Max 1-100: 100

LongStream examples:
   10! = 3628800

DoubleStream examples:
   Average of 1000 random numbers: 0.495

Converting between stream types:
   Sum using mapToInt: 15
   Boxed list: [1, 2, 3, 4, 5]

=== 5. PARALLEL STREAMS ===

Processing 1000 numbers

Prime numbers between 1 and 1000:
   Sequential: 168 primes in 45ms
   Parallel: 168 primes in 15ms
   Speedup: 3.0x

When to use parallel streams:
    Large datasets
    Computationally intensive operations
    Stateless, independent operations
    Small datasets
    Stateful operations
    I/O bound operations

Parallel stream considerations:
   - Order may not be preserved
   - Thread safety is important
   - Overhead for small tasks
   - Use parallel() judiciously

=== 6. REAL-WORLD STREAM EXAMPLES ===

Employee Data:
   Alice      Engineering      $75000.00 age:28
   Bob        Sales            $60000.00 age:35
   Charlie    Engineering      $80000.00 age:32
   Diana      Marketing        $55000.00 age:29
   Eve        Engineering      $90000.00 age:40
   Frank      Sales            $65000.00 age:45
   Grace      HR               $50000.00 age:30

1. Average salary by department:
   Engineering: $81666.67
   Sales: $62500.00
   HR: $50000.00
   Marketing: $55000.00

2. Highest paid in each department:
   HR: Grace ($50000)
   Engineering: Eve ($90000)
   Sales: Frank ($65000)
   Marketing: Diana ($55000)
```

```
3. Employees by age range:
   Under 30: Alice, Diana
   30-39: Bob, Charlie, Grace
   40+: Eve, Frank

4. Total salary budget by department:
   HR: $50000
   Engineering: $245000
   Sales: $125000
   Marketing: $55000

5. Engineering employees earning > $70,000:
   Alice, Charlie, Eve

6. Engineering department statistics:
   Count: 3
   Average: $81666.67
   Max: $90000.00
   Min: $75000.00
   Sum: $245000.00

=== 7. STREAM BEST PRACTICES ===

1. Use Method References:
   Prefer: .map(String::toUpperCase)
   Over:   .map(s -> s.toUpperCase())

2. Avoid Side Effects:
   Don't modify external state in streams
   Use pure functions where possible

3. Choose Right Data Structure:
   ArrayList → .stream()
   Large datasets → .parallelStream()

4. Order Operations Efficiently:
   Filter early to reduce elements
   Expensive operations after filtering

5. Use Primitive Streams for Performance:
   Use IntStream, LongStream, DoubleStream
   Avoid boxing/unboxing overhead

6. Handle Optional Properly:
   Don't call .get() without checking .isPresent()
   Use .orElse(), .orElseGet(), .orElseThrow()
```

```
7. Limit Infinite Streams:
   Always use limit() with generate() or iterate()

8. Collect Once:
   Don’t create multiple terminal operations
   Collect once and reuse

=== 8. COMPLETE STREAM API EXAMPLE ===

All Transactions:
   ID:1001 GROCERY      $ 150.50
   ID:1002 ELECTRONICS  $1200.00
   ID:1003 GROCERY      $  75.25
   ID:1004 CLOTHING     $ 200.00
   ID:1005 ELECTRONICS  $ 850.00
   ID:1006 GROCERY      $  45.75
   ID:1007 CLOTHING     $ 120.00
   ID:1008 GROCERY      $ 300.00

--- Analysis Results ---
1. Total amount: $2941.50
2. Average transaction: $367.69
3. Highest transaction: ELECTRONICS - $1200.00

4. Transactions by type:
   CLOTHING:
     Count: 2
     Total: $320.00
     Average: $160.00
     Max: $200.00
   ELECTRONICS:
     Count: 2
     Total: $2050.00
     Average: $1025.00
     Max: $1200.00
   GROCERY:
     Count: 4
     Total: $571.50
     Average: $142.88
     Max: $300.00

5. Expensive transactions (> $500):
   ID 1002: ELECTRONICS - $1200.00
   ID 1005: ELECTRONICS - $850.00
   ID 1008: GROCERY - $300.00
```

```
6. Transaction IDs: [1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008]

=== STREAM API BENEFITS ===
1. Declarative: Say what you want, not how
2. Composable: Chain operations easily
3. Parallelizable: Easy parallel processing
4. Lazy Evaluation: Efficient execution
5. Functional: Encourages pure functions

=== COMMON MISTAKES ===
1. Reusing streams (they're one-time use)
2. Forgetting terminal operations (nothing happens)
3. Modifying source collection during stream ops
4. Ignoring ordering in parallel streams
5. Not handling Optional properly

=== WHEN TO USE STREAMS ===
 Processing collections of data
 Transformations and filtering
 Aggregations and summaries
 Parallel processing needs
 Functional programming style

=== WHEN NOT TO USE STREAMS ===
 Simple loops (traditional for-loop might be clearer)
 Complex control flow (break, continue, return)
 Stateful operations
 Performance-critical small loops
```

JDBCLambdaStreamIntegration Program Output

```
=== JDBC WITH LAMBDA AND STREAM INTEGRATION ===

Table created/verified
Sample data inserted

Fetching employees from database...
Total employees: 10

=== EXAMPLE 1: FILTERING WITH LAMBDA PREDICATES ===

High salary employees (> $70,000):
   John Smith: $75000.0
   Bob Williams: $82000.0
   David Brown: $90000.0
   Grace Moore: $78000.0
   Irene Anderson: $72000.0
```

```
Engineering department employees:
   John Smith
   Bob Williams
   David Brown
   Grace Moore

Young engineering employees with high salary:
   John Smith (age: 30, salary: $75000.0)
   Grace Moore (age: 33, salary: $78000.0)

=== EXAMPLE 2: CALCULATIONS WITH LAMBDA FUNCTIONS ===

Total base salary for Engineering: $325000.00
Total salary with 10% bonus for Engineering: $357500.00
Total salary with $5000 raise for Sales: $207000.00

=== EMPLOYEE DATA ANALYSIS USING STREAMS ===

All Employees:
 1 John Smith      Engineering   $75000.00  30yrs 2020-01-15 Active
 2 Alice Johnson   Sales         $65000.00  28yrs 2021-03-10 Active
 3 Bob Williams    Engineering   $82000.00  35yrs 2019-05-20 Active
 4 Carol Davis     Marketing     $55000.00  32yrs 2022-07-05 Active
 5 David Brown     Engineering   $90000.00  40yrs 2018-11-30 Active
 6 Eve Miller      Sales         $70000.00  29yrs 2021-09-15 Active
 7 Frank Wilson    HR            $50000.00  45yrs 2020-12-01 Active
 8 Grace Moore     Engineering   $78000.00  33yrs 2022-02-28 Active
 9 Henry Taylor    Marketing     $60000.00  38yrs 2021-06-10 Active
10 Irene Anderson  Sales         $72000.00  31yrs 2023-01-05 Active

1. Employees grouped by department:

   HR Department:
     Frank Wilson

   Engineering Department:
     John Smith
     Bob Williams
     David Brown
     Grace Moore

   Sales Department:
     Alice Johnson
     Eve Miller
     Irene Anderson
   Marketing Department:
     Carol Davis
     Henry Taylor

2. Average salary by department:
   HR         : $50000.00
   Engineering: $81250.00
   Sales      : $69000.00
   Marketing  : $57500.00

3. Top 3 highest paid employees:
   David Brown      : $90000.00
   Bob Williams     : $82000.00
   Grace Moore      : $78000.00

4. Employees by age group:
   30-39 (5 employees):
     Bob Williams (35)
     Carol Davis (32)
     Grace Moore (33)
     Henry Taylor (38)
     Irene Anderson (31)
   Under 30 (3 employees):
     John Smith (30)
     Alice Johnson (28)
     Eve Miller (29)
   40+ (2 employees):
     David Brown (40)
     Frank Wilson (45)

5. Department salary statistics:
   Engineering: $325000.00
   Sales      : $207000.00
   Marketing  : $115000.00
   HR         : $50000.00

6. Employees earning above their department average:

   HR (Average: $50000.00):

   Engineering (Average: $81250.00):
     Bob Williams    : $82000.00
     David Brown     : $90000.00

   Sales (Average: $69000.00):
     Eve Miller      : $70000.00
     Irene Anderson  : $72000.00
   Marketing (Average: $57500.00):
     Henry Taylor    : $60000.00

=== EXAMPLE 4: DATABASE UPDATES WITH LAMBDA ===
Giving 5% raise to Engineering department...
Updated 10 salaries

=== COMPLEX EMPLOYEE REPORT ===

==================================================
DEPARTMENT: HR
==================================================
Employee Count: 1
Total Salary: $50000.00
Average Salary: $50000.00
Average Age: 45.0 years
Highest Paid: Frank Wilson
Employees: Frank Wilson

==================================================
DEPARTMENT: Engineering
==================================================
Employee Count: 4
Total Salary: $341250.00
Average Salary: $85312.50
Average Age: 34.5 years
Highest Paid: David Brown
Employees: John Smith, Bob Williams, David Brown, Grace Moore

==================================================
DEPARTMENT: Sales
==================================================
Employee Count: 3
Total Salary: $207000.00
Average Salary: $69000.00
Average Age: 29.3 years
Highest Paid: Irene Anderson
Employees: Alice Johnson, Eve Miller, Irene Anderson

==================================================
DEPARTMENT: Marketing
==================================================
Employee Count: 2
Total Salary: $115000.00
Average Salary: $57500.00
Average Age: 35.0 years
```

```
Highest Paid: Henry Taylor
Employees: Carol Davis, Henry Taylor

=== OVERALL COMPANY STATISTICS ===
Total Employees: 10
Total Salary Budget: $713250.00
Average Company Salary: $71325.00
Average Employee Age: 33.1 years

=== INTEGRATION BENEFITS ===
1. Concise database operations with lambdas
2. Powerful data processing with Stream API
3. Functional programming style for data transformation
4. Type safety and compile-time checking
5. Easy parallelization of database processing

=== BEST PRACTICES ===
1. Use PreparedStatement with lambda parameters
2. Process large datasets with streams (lazy evaluation)
3. Combine database filtering with stream filtering wisely
4. Use transactions for batch updates
5. Handle exceptions properly in lambda expressions
```