

# Introduction to Java and Core OOP Concepts

Dr. Ratnesh Prasad Srivastava

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

Academic Year: 2026-27

## Course Information

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

## Contents

<b>1 Unit V: Java Database Connectivity (JDBC) and Java 8+ Features</b>	<b>2</b>
1.1 Learning Objectives . . . . .	2
<b>2 Java Database Connectivity (JDBC) Basics</b>	<b>2</b>
2.1 JDBC Architecture . . . . .	2
2.2 Steps to Connect to a Database . . . . .	5
<b>3 Introduction to Lambda Expressions (Java 8+)</b>	<b>17</b>
3.1 Lambda Expressions Fundamentals . . . . .	17
<b>4 Stream API (Java 8+)</b>	<b>26</b>
<b>5 Practical Integration Example</b>	<b>39</b>
<b>6 Summary and Key Takeaways</b>	<b>49</b>
6.1 Assessment Questions . . . . .	49
6.2 Further Reading . . . . .	50

# 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.

```
1  /*
2      *****
3
4  * PROGRAM: JDBCArchitecture.java
5  *
6  * PURPOSE: This program explains the complete JDBC architecture and
7  *           components.
8  *           It provides a conceptual understanding of how Java
9  *           applications
10 *            communicate with databases through different layers.
11 *
12 * PROBLEM SOLVED: Students often struggle to understand how JDBC works
13 *                 behind
14 *                 the scenes. This program visualizes the layered
15 *                 architecture
16 *                 and explains each component's role in database
17 *                 connectivity.
18 *
19 * WHY NECESSARY: Understanding JDBC architecture is fundamental before
20 *                writing
21 *                database code. It helps developers:
22 *                1. Troubleshoot connection issues
```

```

15 *           2. Choose appropriate driver types
16 *           3. Understand performance implications
17 *           4. Make informed decisions about database
18 *           connectivity
19 *
20 * KEY CONCEPTS DEMONSTRATED:
21 * - 4-layer JDBC architecture
22 * - Different types of JDBC drivers
23 * - Role of each JDBC component
24 * - Java application to database communication flow
25 *****
26 */
27
28 public class JDBCArchitecture {
29
30     // ===== JDBC ARCHITECTURE EXPLANATION
31     =====
32     public static void explainArchitecture() {
33         System.out.println("=== JDBC ARCHITECTURE ===");
34         System.out.println();
35         System.out.println("1. Java Application Layer");
36         System.out.println("    - Your Java program using JDBC API");
37         System.out.println("    - Uses interfaces: Connection, Statement
38             , ResultSet");
39
40         System.out.println("\n2. JDBC API Layer");
41         System.out.println("    - java.sql and javax.sql packages");
42         System.out.println("    - Provides standard interfaces");
43         System.out.println("    - DriverManager: Manages database
44             drivers");
45
46         System.out.println("\n3. JDBC Driver Layer");
47         System.out.println("    - Database-specific implementations");
48         System.out.println("    - Types: Type 1, 2, 3, 4");
49         System.out.println("    - Converts JDBC calls to database-
50             specific calls");
51
52         System.out.println("\n4. Database Layer");
53         System.out.println("    - Actual RDBMS (MySQL, Oracle,
54             PostgreSQL, etc.)");
55         System.out.println("    - Stores and manages data");
56
57         System.out.println("\n=== JDBC DRIVER TYPES ===");
58         System.out.println("Type 1: JDBC-ODBC Bridge Driver (Deprecated
59             )");
60         System.out.println("Type 2: Native-API Driver (Part Java, Part
61             Native)");
62         System.out.println("Type 3: Network Protocol Driver (Pure Java)
63             ");
64         System.out.println("Type 4: Thin Driver (Pure Java, Direct) -
65             Most Common");
66     }
67
68     // ===== JDBC COMPONENTS =====
69     public static void explainComponents() {
70         System.out.println("\n=== JDBC CORE COMPONENTS ===");
71
72         System.out.println("\n1. DriverManager:");

```

```

62     System.out.println("    - Manages database drivers");
63     System.out.println("    - Establishes database connections");
64     System.out.println("    - Methods: getConnection(),
        registerDriver()");
65
66     System.out.println("\n2. Connection:");
67     System.out.println("    - Represents a connection to database");
68     System.out.println("    - Creates Statement objects");
69     System.out.println("    - Manages transactions");
70     System.out.println("    - Methods: createStatement(),
        prepareStatement()");
71
72     System.out.println("\n3. Statement:");
73     System.out.println("    - Executes SQL queries");
74     System.out.println("    - Types: Statement, PreparedStatement,
        CallableStatement");
75     System.out.println("    - Methods: executeQuery(), executeUpdate
        ()");
76
77     System.out.println("\n4. ResultSet:");
78     System.out.println("    - Contains query results");
79     System.out.println("    - Navigable cursor through rows");
80     System.out.println("    - Methods: next(), getString(), getInt()
        ");
81
82     System.out.println("\n5. SQLException:");
83     System.out.println("    - Checked exception for database errors
        ");
84     System.out.println("    - Provides error codes and messages");
85 }
86
87 // ===== JDBC WORKFLOW =====
88 public static void explainWorkflow() {
89     System.out.println("\n=== JDBC WORKFLOW ===");
90
91     System.out.println("Step 1: Load and Register Driver");
92     System.out.println("    Class.forName(\"com.mysql.cj.jdbc.Driver
        \");");
93
94     System.out.println("\nStep 2: Establish Connection");
95     System.out.println("    Connection conn = DriverManager.
        getConnection(url, user, pass);");
96
97     System.out.println("\nStep 3: Create Statement");
98     System.out.println("    Statement stmt = conn.createStatement();
        ");
99
100    System.out.println("\nStep 4: Execute Query");
101    System.out.println("    ResultSet rs = stmt.executeQuery(\"
        SELECT * FROM table\");");
102
103    System.out.println("\nStep 5: Process Results");
104    System.out.println("    while(rs.next()) { /* process each row
        */ }");
105
106    System.out.println("\nStep 6: Close Resources");
107    System.out.println("    rs.close(); stmt.close(); conn.close();
        ");

```

```

108     }
109
110     public static void main(String[] args) {
111         System.out.println("=== JDBC ARCHITECTURE AND COMPONENTS ===\n"
112             );
113         explainArchitecture();
114         explainComponents();
115         explainWorkflow();
116     }
117 }

```

Listing 1: JDBC Architecture Overview Program

## 2.2 Steps to Connect to a Database

```

1  /*
2      *****
3
4  * PROGRAM: JDBCConnectionSteps.java
5  *
6  * PURPOSE: This program demonstrates all 6 steps required to connect
7  *           Java
8  *           applications to databases using JDBC. It provides hands-on
9  *           examples
10 *           of each step with multiple implementation approaches.
11 *
12 * PROBLEM SOLVED: Beginners often get confused about the proper
13 *                 sequence and
14 *                 implementation of database connectivity. This
15 *                 program solves:
16 *                 1. Clear step-by-step breakdown
17 *                 2. Multiple ways to accomplish each step
18 *                 3. Proper resource management
19 *                 4. Error handling best practices
20 *
21 * WHY NECESSARY: Database connectivity is a core skill in enterprise
22 *                 Java
23 *                 development. This program is necessary because:
24 *                 1. Most applications need database persistence
25 *                 2. Proper connection management prevents resource
26 *                 leaks
27 *                 3. Understanding different approaches helps in
28 *                 different scenarios
29 *                 4. Industry requires standardized database access
30 *                 patterns
31 *
32 * KEY CONCEPTS DEMONSTRATED:
33 * - 6 essential JDBC steps
34 * - Driver loading (manual vs auto)
35 * - Multiple connection establishment methods
36 * - Different Statement types and their use cases
37 * - Proper ResultSet processing
38 * - Resource cleanup (traditional vs try-with-resources)
39 * - Complete CRUD operations
40 * - Transaction management

```

```

31  *****/
32  */
33  import java.sql.*;
34  import java.util.Properties;
35
36  public class JDBCConnectionSteps {
37
38      // Database configuration - Replace with your database details
39      private static final String URL = "jdbc:mysql://localhost:3306/
40          university";
41      private static final String USER = "root";
42      private static final String PASSWORD = "password";
43
44      // ===== STEP 1: LOAD DATABASE DRIVER
45      =====
46      public static void step1_LoadDriver() {
47          System.out.println("=== STEP 1: LOAD DATABASE DRIVER ===\n");
48
49          try {
50              // Method 1: Using Class.forName() (Legacy but clear)
51              System.out.println("Method 1: Using Class.forName()");
52              Class.forName("com.mysql.cj.jdbc.Driver");
53              System.out.println("MySQL JDBC Driver loaded successfully");
54              ;
55
56          } catch (ClassNotFoundException e) {
57              System.out.println("Error: MySQL JDBC Driver not found!");
58              System.out.println("Make sure you have mysql-connector-java
59                  in classpath");
60              System.out.println("Maven dependency: mysql:mysql-connector
61                  -java:8.0.33");
62              e.printStackTrace();
63          }
64
65          System.out.println("\nAlternative: Modern JDBC 4.0+ auto-loads
66              drivers");
67          System.out.println("from META-INF/services/java.sql.Driver");
68      }
69
70      // ===== STEP 2: ESTABLISH CONNECTION
71      =====
72      public static Connection step2_EstablishConnection() {
73          System.out.println("\n=== STEP 2: ESTABLISH DATABASE CONNECTION
74              ===\n");
75
76          Connection connection = null;
77
78          try {
79              // Method 1: Basic connection with parameters
80              System.out.println("Method 1: Basic connection");
81              connection = DriverManager.getConnection(URL, USER,
82                  PASSWORD);
83
84              // Method 2: Connection with Properties
85              System.out.println("\nMethod 2: Connection with Properties"
86                  );
87              Properties props = new Properties();

```

```

78     props.setProperty("user", USER);
79     props.setProperty("password", PASSWORD);
80     props.setProperty("useSSL", "false");
81     props.setProperty("serverTimezone", "UTC");
82
83     Connection conn2 = DriverManager.getConnection(URL, props);
84     conn2.close();
85
86     // Method 3: Connection with URL parameters
87     System.out.println("\nMethod 3: Connection with URL
88         parameters");
89     String urlWithParams = URL + "?user=" + USER +
90         "&password=" + PASSWORD +
91         "&useSSL=false&serverTimezone=UTC";
92     Connection conn3 = DriverManager.getConnection(
93         urlWithParams);
94     conn3.close();
95
96     if (connection != null) {
97         System.out.println("\n Connection established
98             successfully!");
99         System.out.println("Connection URL: " + URL);
100        System.out.println("Database: " + connection.getCatalog
101            ());
102        System.out.println("Auto Commit: " + connection.
103            getAutoCommit());
104        System.out.println("Transaction Isolation: " +
105            connection.getTransactionIsolation());
106    }
107
108    } catch (SQLException e) {
109        System.out.println("\n Failed to establish connection!");
110        ;
111        System.out.println("SQL State: " + e.getSQLState());
112        System.out.println("Error Code: " + e.getErrorCode());
113        System.out.println("Message: " + e.getMessage());
114        e.printStackTrace();
115    }
116
117    return connection;
118 }
119
120 // ===== STEP 3: CREATE STATEMENT
121 // =====
122 public static void step3_CreateStatement(Connection connection) {
123     System.out.println("\n=== STEP 3: CREATE STATEMENT OBJECTS ===\n");
124
125     if (connection == null) {
126         System.out.println("No connection available. Skipping
127             statement creation.");
128         return;
129     }
130
131     try {
132         // 1. Regular Statement (for static SQL)
133         System.out.println("1. Regular Statement:");
134         Statement statement = connection.createStatement();

```

```

127         System.out.println("    Created: Statement for static SQL
128             queries");
129
130         // 2. PreparedStatement (for parameterized SQL -
131             RECOMMENDED)
132         System.out.println("\n2. PreparedStatement:");
133         String sql = "SELECT * FROM students WHERE age > ? AND
134             department = ?";
135         PreparedStatement preparedStatement = connection.
136             prepareStatement(sql);
137         System.out.println("    Created: PreparedStatement for
138             parameterized queries");
139         System.out.println("    SQL: " + sql);
140         System.out.println("    Benefits: Precompiled, prevents SQL
141             injection");
142
143         // 3. CallableStatement (for stored procedures)
144         System.out.println("\n3. CallableStatement:");
145         CallableStatement callableStatement =
146             connection.prepareCall("{call get_student_by_id(?)}");
147         System.out.println("    Created: CallableStatement for
148             stored procedures");
149
150         // 4. Statement with result set type
151         System.out.println("\n4. Statement with ResultSet type:");
152         Statement scrollableStmt = connection.createStatement(
153             ResultSet.TYPE_SCROLL_INSENSITIVE,
154             ResultSet.CONCUR_READ_ONLY
155         );
156         System.out.println("    Created: Scrollable, read-only
157             ResultSet");
158
159         // Clean up
160         statement.close();
161         preparedStatement.close();
162         callableStatement.close();
163         scrollableStmt.close();
164
165     } catch (SQLException e) {
166         System.out.println("Error creating statement: " + e.
167             getMessage());
168     }
169 }
170
171 // ===== STEP 4: EXECUTE QUERIES
172 // =====
173 public static void step4_ExecuteQueries(Connection connection) {
174     System.out.println("\n=== STEP 4: EXECUTE SQL QUERIES ===\n");
175
176     if (connection == null) {
177         System.out.println("No connection available. Skipping query
178             execution.");
179         return;
180     }
181
182     try {
183         // Create a test table if it doesn't exist
184         createTestTable(connection);
185     }
186 }

```

```

174 // 1. Execute SELECT query (executeQuery)
175 System.out.println("1. SELECT Query (executeQuery):");
176 Statement selectStmt = connection.createStatement();
177 String selectSQL = "SELECT * FROM employees";
178 ResultSet resultSet = selectStmt.executeQuery(selectSQL);
179 System.out.println("    Executed: " + selectSQL);
180
181 // Process results
182 System.out.println("\n    Results:");
183 while (resultSet.next()) {
184     int id = resultSet.getInt("id");
185     String name = resultSet.getString("name");
186     double salary = resultSet.getDouble("salary");
187     System.out.println("    ID: " + id + ", Name: " + name +
188         ", Salary: " + salary);
189 }
190 resultSet.close();
191 selectStmt.close();
192
193 // 2. Execute INSERT query (executeUpdate)
194 System.out.println("\n2. INSERT Query (executeUpdate):");
195 Statement insertStmt = connection.createStatement();
196 String insertSQL = "INSERT INTO employees (name, salary,
197     department) " +
198     "VALUES ('John Doe', 50000.00, 'IT')";
199 int rowsInserted = insertStmt.executeUpdate(insertSQL);
200 System.out.println("    Executed: " + insertSQL);
201 System.out.println("    Rows inserted: " + rowsInserted);
202 insertStmt.close();
203
204 // 3. Execute UPDATE query
205 System.out.println("\n3. UPDATE Query:");
206 Statement updateStmt = connection.createStatement();
207 String updateSQL = "UPDATE employees SET salary = salary *
208     1.1 WHERE department = 'IT'";
209 int rowsUpdated = updateStmt.executeUpdate(updateSQL);
210 System.out.println("    Executed: " + updateSQL);
211 System.out.println("    Rows updated: " + rowsUpdated);
212 updateStmt.close();
213
214 // 4. Execute DELETE query
215 System.out.println("\n4. DELETE Query:");
216 Statement deleteStmt = connection.createStatement();
217 String deleteSQL = "DELETE FROM employees WHERE name = '
218     John Doe'";
219 int rowsDeleted = deleteStmt.executeUpdate(deleteSQL);
220 System.out.println("    Executed: " + deleteSQL);
221 System.out.println("    Rows deleted: " + rowsDeleted);
222 deleteStmt.close();
223
224 // 5. Execute with PreparedStatement
225 System.out.println("\n5. PreparedStatement Example:");
226 String prepSQL = "INSERT INTO employees (name, salary,
227     department) VALUES (?, ?, ?)";
228 PreparedStatement pstmt = connection.prepareStatement(
229     prepSQL);

```

```

226 // Insert multiple records
227 Object[][] employees = {
228     {"Alice Smith", 60000.0, "HR"},
229     {"Bob Johnson", 75000.0, "Engineering"},
230     {"Carol Williams", 55000.0, "Marketing"}
231 };
232
233 for (Object[] emp : employees) {
234     pstmt.setString(1, (String) emp[0]);
235     pstmt.setDouble(2, (Double) emp[1]);
236     pstmt.setString(3, (String) emp[2]);
237     int affected = pstmt.executeUpdate();
238     System.out.println("    Inserted: " + emp[0] + " (" +
239         affected + " row)");
240 }
241 pstmt.close();
242
243 } catch (SQLException e) {
244     System.out.println("Error executing queries: " + e.
245         getMessage());
246 }
247
248 // ===== STEP 5: PROCESS RESULTSET
249 // =====
250 public static void step5_ProcessResultSet(Connection connection) {
251     System.out.println("\n=== STEP 5: PROCESS RESULTSET ===\n");
252
253     if (connection == null) {
254         System.out.println("No connection available. Skipping
255             ResultSet processing.");
256         return;
257     }
258
259     try {
260         Statement stmt = connection.createStatement(
261             ResultSet.TYPE_SCROLL_INSENSITIVE,
262             ResultSet.CONCUR_READ_ONLY
263         );
264
265         ResultSet rs = stmt.executeQuery("SELECT * FROM employees
266             ORDER BY salary DESC");
267
268         System.out.println("ResultSet Metadata:");
269         ResultSetMetaData metaData = rs.getMetaData();
270         int columnCount = metaData.getColumnCount();
271         System.out.println("Number of columns: " + columnCount);
272
273         for (int i = 1; i <= columnCount; i++) {
274             System.out.println("    Column " + i + ": " +
275                 metaData.getColumnName(i) + " (" +
276                 metaData.getColumnTypeName(i) + ")");
277         }
278
279         System.out.println("\nProcessing ResultSet:");
280
281         // Method 1: Using column names (Recommended)
282         System.out.println("\n1. Using column names:");

```

```

279     while (rs.next()) {
280         int id = rs.getInt("id");
281         String name = rs.getString("name");
282         double salary = rs.getDouble("salary");
283         String dept = rs.getString("department");
284         System.out.println("    ID: " + id + ", Name: " + name +
285             ", Salary: $" + salary + ", Dept: " +
                dept);
286     }
287
288     // Method 2: Using column indexes (Faster but less readable
289     )
290     System.out.println("\n2. Using column indexes:");
291     rs.beforeFirst(); // Reset cursor
292     while (rs.next()) {
293         System.out.println("    ID: " + rs.getInt(1) +
294             ", Name: " + rs.getString(2) +
295             ", Salary: $" + rs.getDouble(3) +
296             ", Dept: " + rs.getString(4));
297     }
298
299     // Method 3: Scrollable ResultSet navigation
300     System.out.println("\n3. Scrollable ResultSet Navigation:");
301     ;
302     if (rs.last()) {
303         System.out.println("    Last row - ID: " + rs.getInt("id
304             "));
305     }
306
307     if (rs.first()) {
308         System.out.println("    First row - ID: " + rs.getInt("
309             id"));
310     }
311
312     if (rs.absolute(2)) {
313         System.out.println("    Row 2 - Name: " + rs.getString("
314             name"));
315     }
316
317     // Method 4: Getting different data types
318     System.out.println("\n4. Different data type getters:");
319     rs.beforeFirst();
320     if (rs.next()) {
321         System.out.println("    getObject(): " + rs.getObject("
322             name"));
323         System.out.println("    getString(): " + rs.getString("
324             name"));
325         System.out.println("    getInt(): " + rs.getInt("id"));
326         System.out.println("    getDouble(): " + rs.getDouble("
327             salary"));
328         System.out.println("    getDate(): " + rs.getDate("
329             hire_date"));
330     }
331
332     rs.close();
333     stmt.close();
334
335 } catch (SQLException e) {

```

```

327         System.out.println("Error processing ResultSet: " + e.
328             getMessage());
329     }
330 }
331 // ===== STEP 6: CLOSE RESOURCES
332 // =====
333 public static void step6_CloseResources(Connection connection) {
334     System.out.println("\n=== STEP 6: CLOSE RESOURCES PROPERLY ===\n");
335
336     // Method 1: Traditional try-catch-finally
337     System.out.println("Method 1: Traditional try-catch-finally");
338
339     Connection conn = null;
340     Statement stmt = null;
341     ResultSet rs = null;
342
343     try {
344         conn = DriverManager.getConnection(URL, USER, PASSWORD);
345         stmt = conn.createStatement();
346         rs = stmt.executeQuery("SELECT 1");
347
348         // Process results...
349     } catch (SQLException e) {
350         System.out.println("Error: " + e.getMessage());
351     } finally {
352         // Close in reverse order: ResultSet -> Statement ->
353         // Connection
354         try {
355             if (rs != null) rs.close();
356         } catch (SQLException e) {
357             System.out.println("Error closing ResultSet: " + e.
358                 getMessage());
359         }
360
361         try {
362             if (stmt != null) stmt.close();
363         } catch (SQLException e) {
364             System.out.println("Error closing Statement: " + e.
365                 getMessage());
366         }
367
368         try {
369             if (conn != null) conn.close();
370         } catch (SQLException e) {
371             System.out.println("Error closing Connection: " + e.
372                 getMessage());
373         }
374
375         // Method 2: Try-with-resources (Java 7+)
376         System.out.println("\nMethod 2: Try-with-resources (Recommended)");
377
378         try (Connection conn2 = DriverManager.getConnection(URL, USER,
379             PASSWORD));

```

```

376         Statement stmt2 = conn2.createStatement();
377         ResultSet rs2 = stmt2.executeQuery("SELECT 1")) {
378
379             System.out.println("    Resources automatically closed");
380
381         } catch (SQLException e) {
382             System.out.println("Error: " + e.getMessage());
383         }
384
385         if (connection != null) {
386             try {
387                 connection.close();
388                 System.out.println("\n    Main connection closed
389                                     successfully");
390             } catch (SQLException e) {
391                 System.out.println("\n    Error closing main connection
392                                     : " + e.getMessage());
393             }
394         }
395
396         // ===== HELPER METHOD: CREATE TEST TABLE
397         =====
398         private static void createTestTable(Connection connection) throws
399         SQLException {
400             Statement stmt = connection.createStatement();
401
402             // Drop table if exists (for clean testing)
403             try {
404                 stmt.execute("DROP TABLE IF EXISTS employees");
405             } catch (SQLException e) {
406                 // Ignore if table doesn't exist
407             }
408
409             // Create employees table
410             String createTableSQL = "CREATE TABLE employees (" +
411                                     "id INT PRIMARY KEY AUTO_INCREMENT, " +
412                                     "name VARCHAR(100) NOT NULL, " +
413                                     "salary DECIMAL(10,2), " +
414                                     "department VARCHAR(50), " +
415                                     "hire_date DATE DEFAULT CURRENT_DATE)";
416
417             stmt.execute(createTableSQL);
418
419             // Insert sample data
420             String insertDataSQL = "INSERT INTO employees (name, salary,
421                                     department) VALUES " +
422                                     "('John Smith', 50000.00, 'IT'), " +
423                                     "('Jane Doe', 60000.00, 'HR'), " +
424                                     "('Mike Johnson', 75000.00, 'Engineering
425                                     ')" );
426
427             stmt.execute(insertDataSQL);
428
429             stmt.close();
430             System.out.println("Test table 'employees' created with sample
431                                     data");
432         }

```

```

427 // ===== COMPLETE EXAMPLE =====
428 public static void completeExample() {
429     System.out.println("\n=== COMPLETE JDBC EXAMPLE ===");
430
431     // Using try-with-resources for automatic resource management
432     try (Connection conn = DriverManager.getConnection(URL, USER,
433         PASSWORD)) {
434
435         System.out.println("1. Connection established");
436
437         // Create a table for demonstration
438         try (Statement createStmt = conn.createStatement()) {
439             createStmt.execute("CREATE TABLE IF NOT EXISTS products
440                 (" +
441                     "id INT PRIMARY KEY AUTO_INCREMENT, "
442                     +
443                     "name VARCHAR(100), " +
444                     "price DECIMAL(10,2), " +
445                     "quantity INT)");
446             System.out.println("2. Table created/verified");
447         }
448
449         // Insert data using PreparedStatement
450         String insertSQL = "INSERT INTO products (name, price,
451             quantity) VALUES (?, ?, ?)";
452         try (PreparedStatement pstmt = conn.prepareStatement(
453             insertSQL)) {
454
455             Object[][] products = {
456                 {"Laptop", 999.99, 10},
457                 {"Mouse", 25.50, 100},
458                 {"Keyboard", 75.00, 50},
459                 {"Monitor", 299.99, 20}
460             };
461
462             for (Object[] product : products) {
463                 pstmt.setString(1, (String) product[0]);
464                 pstmt.setDouble(2, (Double) product[1]);
465                 pstmt.setInt(3, (Integer) product[2]);
466                 pstmt.executeUpdate();
467             }
468             System.out.println("3. Data inserted");
469         }
470
471         // Query data
472         try (Statement stmt = conn.createStatement();
473             ResultSet rs = stmt.executeQuery("SELECT * FROM
474             products")) {
475
476             System.out.println("\n4. Query Results:");
477             System.out.println("ID\tName\t\tPrice\tQuantity");
478             System.out.println("
479                 -----");
480
481             double totalValue = 0;
482             while (rs.next()) {
483                 int id = rs.getInt("id");

```

```

478         String name = rs.getString("name");
479         double price = rs.getDouble("price");
480         int quantity = rs.getInt("quantity");
481
482         System.out.printf("%d\t%-10s\t$%.2f\t%d\n", id,
483             name, price, quantity);
484
485         totalValue += price * quantity;
486     }
487
488     System.out.println("
489         -----");
490     System.out.printf("Total inventory value: $%.2f\n",
491         totalValue);
492 }
493
494 // Update data
495 try (Statement updateStmt = conn.createStatement()) {
496     int rowsUpdated = updateStmt.executeUpdate(
497         "UPDATE products SET price = price * 0.9 WHERE
498             quantity > 30"
499     );
500     System.out.println("\n5. Prices updated for " +
501         rowsUpdated + " products");
502 }
503
504 // Transaction example
505 System.out.println("\n6. Transaction Example:");
506 conn.setAutoCommit(false); // Start transaction
507
508 try {
509     try (Statement transStmt = conn.createStatement()) {
510         // Multiple operations
511         transStmt.executeUpdate("UPDATE products SET
512             quantity = quantity - 5 WHERE name = 'Laptop'");
513         transStmt.executeUpdate("UPDATE products SET
514             quantity = quantity + 5 WHERE name = 'Mouse'");
515
516         // Simulate an error condition
517         boolean errorCondition = false; // Change to true
518             to test rollback
519         if (errorCondition) {
520             throw new SQLException("Simulated error during
521                 transaction");
522         }
523
524         conn.commit(); // Commit transaction
525         System.out.println(" Transaction committed
526             successfully");
527     }
528 } catch (SQLException e) {
529     conn.rollback(); // Rollback on error
530     System.out.println(" Transaction rolled back due to:
531         " + e.getMessage());
532 } finally {
533     conn.setAutoCommit(true); // Restore auto-commit
534 }

```

```

525     } catch (SQLException e) {
526         System.out.println("Database error: " + e.getMessage());
527         e.printStackTrace();
528     }
529 }
530
531 // ===== MAIN METHOD =====
532 public static void main(String[] args) {
533     System.out.println("=== JDBC CONNECTION STEPS - COMPLETE GUIDE
534         ===\n");
535
536     System.out.println("This example demonstrates all 6 steps of
537         JDBC:");
538     System.out.println("1. Load Database Driver");
539     System.out.println("2. Establish Connection");
540     System.out.println("3. Create Statement");
541     System.out.println("4. Execute Queries");
542     System.out.println("5. Process ResultSet");
543     System.out.println("6. Close Resources\n");
544
545     // Step 1: Load Driver
546     step1_LoadDriver();
547
548     // Step 2: Establish Connection
549     Connection connection = step2_EstablishConnection();
550
551     // Step 3: Create Statement
552     step3_CreateStatement(connection);
553
554     // Step 4: Execute Queries
555     step4_ExecuteQueries(connection);
556
557     // Step 5: Process ResultSet
558     step5_ProcessResultSet(connection);
559
560     // Step 6: Close Resources
561     step6_CloseResources(connection);
562
563     // Complete Example
564     System.out.println("\n".repeat(3));
565     System.out.println("=".repeat(60));
566     System.out.println("DEMONSTRATING COMPLETE JDBC WORKFLOW");
567     System.out.println("=".repeat(60));
568     completeExample();
569
570     System.out.println("\n=== JDBC BEST PRACTICES ===");
571     System.out.println("1. Use PreparedStatement to prevent SQL
572         injection");
573     System.out.println("2. Always close resources in finally block
574         or use try-with-resources");
575     System.out.println("3. Use connection pooling for production
576         applications");
577     System.out.println("4. Handle SQLException properly with
578         specific error messages");
579     System.out.println("5. Use transactions for multiple related
580         operations");
581     System.out.println("6. Validate and sanitize user input before
582         database operations");

```

```

575     System.out.println("7. Use appropriate data types (getInt for
        INT, getString for VARCHAR)");
576     System.out.println("8. Limit ResultSet size for large queries (
        use LIMIT clause)");
577     System.out.println("9. Use batch updates for multiple insert/
        update operations");
578     System.out.println("10. Test with different database
        configurations");
579 }
580 }

```

Listing 2: Complete JDBC Connection Example Program

## 3 Introduction to Lambda Expressions (Java 8+)

### 3.1 Lambda Expressions Fundamentals

```

1  /*
    *****
2  * PROGRAM: LambdaExpressionsGuide.java
3  *
4  * PURPOSE: This comprehensive program introduces Lambda Expressions in
        Java 8+,
5  *         showing how they enable functional programming and reduce
        boilerplate
6  *         code. It covers syntax, functional interfaces, method
        references,
7  *         and real-world applications.
8  *
9  * PROBLEM SOLVED: Traditional Java code often involves verbose
        anonymous inner
10 *         classes. This program solves:
11 *         1. Verbosity reduction in event handlers, threads,
        comparators
12 *         2. Readability improvement through concise syntax
13 *         3. Functional programming adoption in Java
14 *         4. Stream API compatibility requirement
15 *
16 * WHY NECESSARY: Lambda expressions are essential for modern Java
        development:
17 *         1. Required for using Stream API effectively
18 *         2. Industry standard for concise, readable code
19 *         3. Foundation for reactive programming
20 *         4. Enables parallel processing patterns
21 *         5. Reduces boilerplate code by 70-80%
22 *
23 * KEY CONCEPTS DEMONSTRATED:
24 * - Lambda syntax (parameters, arrow, body)
25 * - Functional interfaces (built-in and custom)
26 * - Method references (4 types)
27 * - Variable capture rules
28 * - Real-world use cases
29 * - Best practices and common pitfalls
30 *****
    */

```

```

31
32 import java.util.*;
33 import java.util.function.*;
34
35 public class LambdaExpressionsGuide {
36
37     // ===== FUNCTIONAL INTERFACES =====
38     @FunctionalInterface
39     interface SimpleCalculator {
40         int calculate(int a, int b);
41     }
42
43     @FunctionalInterface
44     interface StringTransformer {
45         String transform(String input);
46     }
47
48     @FunctionalInterface
49     interface ConditionChecker {
50         boolean check(int number);
51     }
52
53     // ===== 1. BASIC LAMBDA SYNTAX =====
54     public static void basicLambdaSyntax() {
55         System.out.println("=== 1. BASIC LAMBDA SYNTAX ===\n");
56
57         // Before Java 8: Anonymous class
58         System.out.println("Before Java 8 - Anonymous Class:");
59         Runnable oldRunnable = new Runnable() {
60             @Override
61             public void run() {
62                 System.out.println("    Running with anonymous class");
63             }
64         };
65         oldRunnable.run();
66
67         // Java 8+: Lambda expression
68         System.out.println("\nJava 8+ - Lambda Expression:");
69         Runnable newRunnable = () -> System.out.println("    Running
70             with lambda");
71         newRunnable.run();
72
73         // Lambda with parameters
74         System.out.println("\nLambda with Parameters:");
75         SimpleCalculator adder = (a, b) -> a + b;
76         SimpleCalculator multiplier = (x, y) -> x * y;
77
78         System.out.println("    Addition: 5 + 3 = " + adder.calculate(5,
79             3));
80         System.out.println("    Multiplication: 5 * 3 = " + multiplier.
81             calculate(5, 3));
82
83         // Lambda with explicit types
84         System.out.println("\nLambda with Explicit Types:");
85         SimpleCalculator subtractor = (int a, int b) -> a - b;
86         System.out.println("    Subtraction: 10 - 4 = " + subtractor.
87             calculate(10, 4));

```

```

85 // Lambda with multiple statements
86 System.out.println("\nLambda with Multiple Statements:");
87 SimpleCalculator complexCalc = (a, b) -> {
88     int sum = a + b;
89     int product = a * b;
90     return sum + product;
91 };
92 System.out.println("    Complex calculation (5,3): " +
93     complexCalc.calculate(5, 3));
94 }
95 // ===== 2. BUILT-IN FUNCTIONAL INTERFACES
96 // =====
97 public static void builtInFunctionalInterfaces() {
98     System.out.println("\n=== 2. BUILT-IN FUNCTIONAL INTERFACES
99     ===\n");
100
101     // 1. Predicate<T> - Tests a condition
102     System.out.println("1. Predicate<T> - Tests a condition:");
103     Predicate<Integer> isEven = n -> n % 2 == 0;
104     Predicate<Integer> isPositive = n -> n > 0;
105     Predicate<String> isEmpty = String::isEmpty;
106
107     System.out.println("    Is 10 even? " + isEven.test(10));
108     System.out.println("    Is -5 positive? " + isPositive.test(-5));
109     ;
110     System.out.println("    Is empty string? " + isEmpty.test(""));
111
112     // Predicate chaining
113     Predicate<Integer> isEvenAndPositive = isEven.and(isPositive);
114     System.out.println("    Is 6 even AND positive? " +
115         isEvenAndPositive.test(6));
116
117     // 2. Function<T, R> - Transforms input to output
118     System.out.println("\n2. Function<T, R> - Transforms input to
119         output:");
120     Function<String, Integer> stringLength = String::length;
121     Function<Integer, String> intToString = Object::toString;
122     Function<String, String> toUpperCase = String::toUpperCase;
123     Function<String, String> addExclamation = s -> s + "!";
124
125     System.out.println("    Length of 'Hello': " + stringLength.
126         apply("Hello"));
127     System.out.println("    123 as string: " + intToString.apply
128         (123));
129
130     // Function composition
131     Function<String, String> shout = toUpperCase.andThen(
132         addExclamation);
133     System.out.println("    Shout 'hello': " + shout.apply("hello"))
134         ;
135
136     // 3. Consumer<T> - Consumes input, returns nothing
137     System.out.println("\n3. Consumer<T> - Consumes input, returns
138         nothing:");
139     Consumer<String> printer = System.out::println;
140     Consumer<Integer> squarePrinter = n -> System.out.println(n * n
141         );

```

```

131
132 System.out.print("    Printing with consumer: ");
133 printer.accept("Hello Consumer!");
134 System.out.print("    Square of 5: ");
135 squarePrinter.accept(5);
136
137 // Consumer chaining
138 Consumer<String> printAndUpperCase = printer.andThen(s ->
139     System.out.println("Uppercase: " + s.toUpperCase()));
140 System.out.print("    Chained consumer: ");
141 printAndUpperCase.accept("test");
142
143 // 4. Supplier<T> - Provides values
144 System.out.println("\n4. Supplier<T> - Provides values:");
145 Supplier<Double> randomSupplier = Math::random;
146 Supplier<String> greetingSupplier = () -> "Hello World!";
147 Supplier<List<String>> listSupplier = ArrayList::new;
148
149 System.out.println("    Random number: " + randomSupplier.get())
150     ;
151 System.out.println("    Greeting: " + greetingSupplier.get());
152 System.out.println("    New list: " + listSupplier.get());
153
154 // 5. UnaryOperator<T> - Function where input and output are
155     same type
156 System.out.println("\n5. UnaryOperator<T> - Function with same
157     input/output type:");
158 UnaryOperator<Integer> square = n -> n * n;
159 UnaryOperator<String> reverse = s -> new StringBuilder(s).
160     reverse().toString();
161
162 System.out.println("    Square of 7: " + square.apply(7));
163 System.out.println("    Reverse 'lambda': " + reverse.apply("
164     lambda"));
165
166 // 6. BinaryOperator<T> - Takes two inputs, returns same type
167 System.out.println("\n6. BinaryOperator<T> - Two inputs,
168     returns same type:");
169 BinaryOperator<Integer> max = Math::max;
170 BinaryOperator<String> concatenator = (s1, s2) -> s1 + " " + s2
171     ;
172
173 System.out.println("    Max of 10 and 20: " + max.apply(10, 20))
174     ;
175 System.out.println("    Concatenate: " + concatenator.apply("
176     Hello", "World"));
177 }
178
179 // ===== 3. LAMBDA WITH COLLECTIONS
180     =====
181 public static void lambdaWithCollections() {
182     System.out.println("\n=== 3. LAMBDA WITH COLLECTIONS ===\n");
183
184     List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry"
185         , "Date", "Elderberry");
186
187     // Before Java 8: External iteration
188     System.out.println("Before Java 8 - External iteration:");

```

```

178     for (String fruit : fruits) {
179         if (fruit.startsWith("A")) {
180             System.out.println("    " + fruit);
181         }
182     }
183
184     // Java 8: Internal iteration with forEach
185     System.out.println("\nJava 8 - Internal iteration with forEach:
186         ");
187     fruits.forEach(fruit -> System.out.println("    " + fruit));
188
189     // Method reference
190     System.out.println("\nUsing Method Reference:");
191     fruits.forEach(System.out::println);
192
193     // Filtering with Predicate
194     System.out.println("\nFiltering fruits starting with 'C':");
195     Predicate<String> startsWithC = s -> s.startsWith("C");
196     fruits.stream()
197         .filter(startsWithC)
198         .forEach(System.out::println);
199
200     // Transforming with Function
201     System.out.println("\nTransforming to uppercase:");
202     Function<String, String> toUpper = String::toUpperCase;
203     fruits.stream()
204         .map(toUpper)
205         .forEach(System.out::println);
206
207     // Sorting with Comparator
208     System.out.println("\nSorted by length:");
209     Comparator<String> byLength = (s1, s2) -> s1.length() - s2.
210         length();
211     fruits.stream()
212         .sorted(byLength)
213         .forEach(f -> System.out.println("    " + f));
214
215     // Custom sorting
216     System.out.println("\nSorted by length then alphabetically:");
217     Comparator<String> byLengthThenAlpha =
218         Comparator.comparingInt(String::length)
219             .thenComparing(Comparator.naturalOrder());
220
221     fruits.stream()
222         .sorted(byLengthThenAlpha)
223         .forEach(f -> System.out.println("    " + f));
224
225     }
226
227     // ===== 4. METHOD REFERENCES =====
228     public static void methodReferences() {
229         System.out.println("\n=== 4. METHOD REFERENCES ===\n");
230
231         List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "
232             David");
233
234         // 1. Static method reference
235         System.out.println("1. Static Method Reference:");

```

```

232     names.forEach(System.out::println); // Equivalent to: s ->
        System.out.println(s)
233
234     // Using custom static method
235     System.out.println("\nUsing custom static method:");
236     names.forEach(LambdaExpressionsGuide::printWithPrefix);
237
238     // 2. Instance method reference on specific instance
239     System.out.println("\n2. Instance Method Reference (specific
        instance):");
240     String prefix = "Name: ";
241     names.forEach(prefix::concat); // Equivalent to: s -> prefix.
        concat(s)
242
243     // 3. Instance method reference on arbitrary instance
244     System.out.println("\n3. Instance Method Reference (arbitrary
        instance):");
245     names.forEach(String::toUpperCase); // Equivalent to: s -> s.
        toUpperCase()
246
247     // 4. Constructor reference
248     System.out.println("\n4. Constructor Reference:");
249     Supplier<List<String>> listSupplier = ArrayList::new; //
        Equivalent to: () -> new ArrayList<>()
250     List<String> newList = listSupplier.get();
251     newList.add("New Element");
252     System.out.println("    New list: " + newList);
253
254     // Constructor reference with parameters
255     System.out.println("\nConstructor reference with parameters:");
256     Function<String, Integer> parseInt = Integer::new; //
        Equivalent to: s -> new Integer(s)
257     System.out.println("    Parsed '123': " + parseInt.apply("123"))
        ;
258 }
259
260 private static void printWithPrefix(String s) {
261     System.out.println("    >>> " + s);
262 }
263
264 // ===== 5. REAL-WORLD EXAMPLES =====
265 public static void realWorldExamples() {
266     System.out.println("\n=== 5. REAL-WORLD LAMBDA EXAMPLES ===\n")
        ;
267
268     // Example 1: Event Handlers (GUI)
269     System.out.println("Example 1: Event Handlers in GUI");
270     System.out.println("// Old way:");
271     System.out.println("button.addActionListener(new ActionListener
        () {});
272     System.out.println("    public void actionPerformed(ActionEvent
        e) {});
273     System.out.println("        System.out.println(\"Button clicked
        !\");");
274     System.out.println("    }");
275     System.out.println("});");
276
277     System.out.println("\n// New way with lambda:");

```

```

278     System.out.println("button.addActionListener(e -> System.out.
        println(\"Button clicked!\"));");
279
280     // Example 2: Thread Creation
281     System.out.println("\nExample 2: Thread Creation");
282     System.out.println("// Old way:");
283     System.out.println("new Thread(new Runnable() {");
284     System.out.println("    public void run() {");
285     System.out.println("        System.out.println(\"Thread running
        \");");
286     System.out.println("    }");
287     System.out.println("}).start();");
288
289     System.out.println("\n// New way with lambda:");
290     System.out.println("new Thread(() -> System.out.println(\"
        Thread running\")).start();");
291
292     // Example 3: Sorting Collections
293     System.out.println("\nExample 3: Sorting Employees");
294     List<Employee> employees = Arrays.asList(
295         new Employee("Alice", "Engineering", 75000),
296         new Employee("Bob", "Sales", 60000),
297         new Employee("Charlie", "Engineering", 80000),
298         new Employee("Diana", "Marketing", 55000)
299     );
300
301     System.out.println("\nEmployees sorted by salary (descending):"
        );
302     employees.sort((e1, e2) -> e2.getSalary() - e1.getSalary());
303     employees.forEach(e -> System.out.println("    " + e));
304
305     // Example 4: Filtering and Mapping
306     System.out.println("\nExample 4: Filtering high-salary
        Engineering employees:");
307     employees.stream()
308         .filter(e -> e.getDepartment().equals("Engineering"))
309         .filter(e -> e.getSalary() > 70000)
310         .map(Employee::getName)
311         .forEach(name -> System.out.println("    " + name));
312
313     // Example 5: Custom Functional Interface
314     System.out.println("\nExample 5: Custom Validator");
315     Validator<String> emailValidator = email -> email.contains("@")
        && email.contains(".");
316     Validator<Integer> ageValidator = age -> age >= 18;
317
318     System.out.println("    Valid email 'test@example.com'? " +
        emailValidator.validate("test@example.com"));
319     System.out.println("    Valid age 25? " + ageValidator.validate
        (25));
320
321 }
322
323 // ===== 6. VARIABLE CAPTURE =====
324 public static void variableCapture() {
325     System.out.println("\n=== 6. VARIABLE CAPTURE IN LAMBDA ===\n"
        );
326
327     // Effectively final local variable

```

```

328     final String fixedPrefix = "Item: ";
329     String variablePrefix = "Product: ";
330
331     // variablePrefix must be effectively final
332     // variablePrefix = "Changed: "; // This would cause error
333
334     List<String> items = Arrays.asList("Book", "Pen", "Notebook");
335
336     System.out.println("Using effectively final variables:");
337     items.forEach(item -> {
338         // Can access final/effectively final variables
339         System.out.println("    " + fixedPrefix + item);
340         System.out.println("    " + variablePrefix + item);
341
342         // Cannot modify captured variables
343         // variablePrefix = "New: "; // Compilation error
344     });
345
346     // Instance and static variable capture
347     System.out.println("\nInstance and static variable capture:");
348     LambdaDemo demo = new LambdaDemo();
349     demo.instanceVariable = 100;
350
351     items.forEach(item -> {
352         // Can modify instance variables
353         demo.instanceVariable++;
354         // Can modify static variables
355         LambdaDemo.staticVariable++;
356
357         System.out.println("    Instance: " + demo.instanceVariable
358             +
359             ", Static: " + LambdaDemo.staticVariable);
360     });
361 }
362
363 // ===== 7. LAMBDA BEST PRACTICES
364 // =====
365 public static void lambdaBestPractices() {
366     System.out.println("\n=== 7. LAMBDA BEST PRACTICES ===\n");
367
368     System.out.println("1. Keep Lambdas Short and Simple:");
369     System.out.println("    Good: names.stream().filter(n -> n.
370         length() > 3)");
371     System.out.println("    Bad: Complex logic in lambda - extract
372         to method");
373
374     System.out.println("\n2. Use Method References When Possible:");
375     ;
376     System.out.println("    Instead of: s -> s.toUpperCase()");
377     System.out.println("    Use: String::toUpperCase");
378
379     System.out.println("\n3. Avoid Side Effects:");
380     System.out.println("    Pure functions are better than mutating
381         external state");
382
383     System.out.println("\n4. Use Descriptive Parameter Names:");
384     System.out.println("    Good: (person, department) -> ...");
385     System.out.println("    Bad: (p, d) -> ...");

```

```

380
381     System.out.println("\n5. Consider Type Inference:");
382     System.out.println("    Let compiler infer types when clear");
383     System.out.println("    (a, b) -> a + b instead of (int a, int
        b) -> a + b");
384
385     System.out.println("\n6. Chain Operations Readably:");
386     System.out.println("    list.stream()");
387     System.out.println("        .filter(...)");
388     System.out.println("        .map(...)");
389     System.out.println("        .collect(...)");
390 }
391
392 // ===== MAIN METHOD =====
393 public static void main(String[] args) {
394     System.out.println("=== LAMBDA EXPRESSIONS - COMPLETE GUIDE
        ===\n");
395
396     System.out.println("Lambda Expressions introduce functional
        programming");
397     System.out.println("features to Java, enabling concise,
        readable code.\n");
398
399     // 1. Basic Syntax
400     basicLambdaSyntax();
401
402     // 2. Built-in Functional Interfaces
403     builtInFunctionalInterfaces();
404
405     // 3. Lambda with Collections
406     lambdaWithCollections();
407
408     // 4. Method References
409     methodReferences();
410
411     // 5. Real-world Examples
412     realWorldExamples();
413
414     // 6. Variable Capture
415     variableCapture();
416
417     // 7. Best Practices
418     lambdaBestPractices();
419
420     System.out.println("\n=== KEY BENEFITS OF LAMBDA EXPRESSIONS
        ===");
421     System.out.println("1. Conciseness: Less boilerplate code");
422     System.out.println("2. Readability: More expressive code");
423     System.out.println("3. Functional Programming: Support for FP
        paradigms");
424     System.out.println("4. Parallelism: Easier parallel processing"
        );
425     System.out.println("5. API Design: Enables fluent APIs");
426
427     System.out.println("\n=== COMMON PITFALLS ===");
428     System.out.println("1. Overusing lambdas for complex logic");
429     System.out.println("2. Not understanding variable capture rules
        ");

```

```

430     System.out.println("3. Ignoring exception handling in lambdas")
431     ;
432     System.out.println("4. Performance overhead in some cases");
433     System.out.println("5. Debugging can be more challenging");
434 }
435 // ===== SUPPORTING CLASSES =====
436 static class Employee {
437     private String name;
438     private String department;
439     private int salary;
440
441     public Employee(String name, String department, int salary) {
442         this.name = name;
443         this.department = department;
444         this.salary = salary;
445     }
446
447     public String getName() { return name; }
448     public String getDepartment() { return department; }
449     public int getSalary() { return salary; }
450
451     @Override
452     public String toString() {
453         return name + " (" + department + "): $" + salary;
454     }
455 }
456
457 @FunctionalInterface
458 interface Validator<T> {
459     boolean validate(T value);
460 }
461
462 static class LambdaDemo {
463     int instanceVariable;
464     static int staticVariable = 0;
465 }
466 }

```

Listing 3: Lambda Expressions - Complete Guide Program

## 4 Stream API (Java 8+)

```

1  /*
2     *****
3
4     * PROGRAM: StreamAPIGuide.java
5     *
6     * PURPOSE: This comprehensive guide demonstrates Java 8+ Stream API
7     *           for
8     *           declarative data processing. It shows how to process
9     *           collections
10    *           efficiently using functional programming patterns.
11    *
12    * PROBLEM SOLVED: Traditional Java collection processing involves:
13    *                  1. Verbose iteration code

```

```

10  *           2. Complex filtering and transformation logic
11  *           3. Difficulty in parallel processing
12  *           4. Code duplication for common operations
13  *
14  *           Stream API solves these by providing:
15  *           1. Declarative syntax (what, not how)
16  *           2. Composable operations (pipeline)
17  *           3. Automatic parallelization
18  *           4. Built-in common operations
19  *
20  * WHY NECESSARY: Stream API is essential for modern Java because:
21  *           1. Industry standard for data processing
22  *           2. Enables functional programming in Java
23  *           3. Simplifies complex data transformations
24  *           4. Improves code readability and maintainability
25  *           5. Built-in support for parallel processing
26  *           6. Required for big data and analytics applications
27  *
28  * KEY CONCEPTS DEMONSTRATED:
29  * - Stream creation (from collections, arrays, generators)
30  * - Intermediate operations (filter, map, flatMap, sorted, distinct)
31  * - Terminal operations (collect, reduce, forEach, count)
32  * - Primitive streams (IntStream, LongStream, DoubleStream)
33  * - Parallel streams and performance considerations
34  * - Collectors for advanced data aggregation
35  * - Real-world data processing scenarios
36  * *****
    */
37
38 import java.util.*;
39 import java.util.stream.*;
40 import java.util.function.*;
41 import java.time.LocalDate;
42
43 public class StreamAPIGuide {
44
45     // ===== 1. STREAM BASICS =====
46     public static void streamBasics() {
47         System.out.println("=== 1. STREAM API BASICS ===\n");
48
49         List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry",
50             , "Date", "Elderberry");
51
52         System.out.println("Source Collection: " + fruits);
53
54         // Creating streams
55         System.out.println("\n1. Different ways to create streams:");
56
57         // From Collection
58         Stream<String> stream1 = fruits.stream();
59         System.out.println("    From Collection: fruits.stream()");
60
61         // From Array
62         String[] array = {"One", "Two", "Three"};
63         Stream<String> stream2 = Arrays.stream(array);
64         System.out.println("    From Array: Arrays.stream(array)");
65
66         // Static factory methods

```

```

66 Stream<String> stream3 = Stream.of("A", "B", "C");
67 System.out.println("    Using Stream.of(): Stream.of(\"A\", \"B
    \", \"C\")");
68
69 Stream<Integer> stream4 = Stream.iterate(1, n -> n + 1).limit
    (5);
70 System.out.println("    Infinite stream: Stream.iterate(1, n ->
    n + 1).limit(5)");
71
72 Stream<Double> stream5 = Stream.generate(Math::random).limit(3)
    ;
73 System.out.println("    Generated stream: Stream.generate(Math::
    random).limit(3)");
74
75 // Stream operations pipeline
76 System.out.println("\n2. Stream Operations Pipeline:");
77 System.out.println("    Source        Intermediate Operations
    Terminal Operation");
78
79 long count = fruits.stream()                // Source
80     .filter(f -> f.length() > 5)           // Intermediate
81     .map(String::toUpperCase)              // Intermediate
82     .count();                               // Terminal
83
84 System.out.println("    Example: Count fruits with length > 5 =
    " + count);
85
86 // Characteristics of streams
87 System.out.println("\n3. Stream Characteristics:");
88 System.out.println("    - Not a data structure, carries values
    from source");
89 System.out.println("    - Functional in nature (doesn't modify
    source)");
90 System.out.println("    - Lazily evaluated (only when terminal
    operation called)");
91 System.out.println("    - Possibly unbounded");
92 System.out.println("    - Consumable (can be traversed only once
    )");
93 }
94
95 // ===== 2. INTERMEDIATE OPERATIONS
    =====
96 public static void intermediateOperations() {
97     System.out.println("\n=== 2. INTERMEDIATE OPERATIONS ===\n");
98
99     List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
100         9, 10);
101     System.out.println("Original numbers: " + numbers);
102
103     // 1. filter() - Selects elements based on predicate
104     System.out.println("\n1. filter() - Selects elements:");
105     List<Integer> evenNumbers = numbers.stream()
106         .filter(n -> n % 2 == 0)
107         .collect(Collectors.toList());
108
109     System.out.println("    Even numbers: " + evenNumbers);
110
111     // 2. map() - Transforms each element

```

```

110 System.out.println("\n2. map() - Transforms elements:");
111 List<Integer> squares = numbers.stream()
112     .map(n -> n * n)
113     .collect(Collectors.toList());
114 System.out.println("    Squares: " + squares);
115
116 List<String> numberStrings = numbers.stream()
117     .map(n -> "Number: " + n)
118     .collect(Collectors.toList()
119         );
120 System.out.println("    Number strings: " + numberStrings);
121
122 // 3. flatMap() - Flattens nested structures
123 System.out.println("\n3. flatMap() - Flattens nested structures
124     :");
125 List<List<String>> nestedLists = Arrays.asList(
126     Arrays.asList("A", "B", "C"),
127     Arrays.asList("D", "E", "F"),
128     Arrays.asList("G", "H", "I")
129 );
130 List<String> flatList = nestedLists.stream()
131     .flatMap(List::stream)
132     .collect(Collectors.toList());
133 System.out.println("    Nested: " + nestedLists);
134 System.out.println("    Flattened: " + flatList);
135
136 // 4. distinct() - Removes duplicates
137 System.out.println("\n4. distinct() - Removes duplicates:");
138 List<Integer> withDuplicates = Arrays.asList(1, 2, 2, 3, 3, 3,
139     4, 4, 4, 4);
140 List<Integer> distinct = withDuplicates.stream()
141     .distinct()
142     .collect(Collectors.toList());
143 System.out.println("    With duplicates: " + withDuplicates);
144 System.out.println("    Distinct: " + distinct);
145
146 // 5. sorted() - Sorts elements
147 System.out.println("\n5. sorted() - Sorts elements:");
148 List<Integer> shuffled = Arrays.asList(5, 3, 8, 1, 9, 2);
149 List<Integer> sortedAsc = shuffled.stream()
150     .sorted()
151     .collect(Collectors.toList());
152 List<Integer> sortedDesc = shuffled.stream()
153     .sorted(Comparator.reverseOrder())
154     .collect(Collectors.toList());
155 System.out.println("    Original: " + shuffled);
156 System.out.println("    Sorted ascending: " + sortedAsc);
157 System.out.println("    Sorted descending: " + sortedDesc);
158
159 // 6. peek() - Debugging operation
160 System.out.println("\n6. peek() - For debugging:");
161 List<Integer> processed = numbers.stream()

```

```

160         .peek(n -> System.out.print("
161             Before filter: " + n + " "
162         ))
163         .filter(n -> n > 5)
164         .peek(n -> System.out.println("
165             After filter: " + n))
166         .collect(Collectors.toList());
167 System.out.println("    Result: " + processed);
168
169 // 7. limit() and skip()
170 System.out.println("\n7. limit() and skip():");
171 List<Integer> limited = numbers.stream()
172     .skip(3)           // Skip first 3
173     .limit(4)         // Take next 4
174     .collect(Collectors.toList());
175 System.out.println("    Skip 3, limit 4: " + limited);
176
177 // 8. Multiple operations chained
178 System.out.println("\n8. Chained Intermediate Operations:");
179 List<String> result = numbers.stream()
180     .filter(n -> n % 2 != 0)    //
181     .map(n -> n * 2)           //
182     .filter(n -> n > 5)        //
183     .map(n -> "#" + n)         //
184     .collect(Collectors.toList());
185 System.out.println("    Chained result: " + result);
186 }
187
188 // ===== 3. TERMINAL OPERATIONS =====
189 public static void terminalOperations() {
190     System.out.println("\n=== 3. TERMINAL OPERATIONS ===\n");
191
192     List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);
193     System.out.println("Numbers: " + numbers);
194
195     // 1. forEach() - Performs action on each element
196     System.out.println("\n1. forEach() - Performs action:");
197     System.out.print("    Printing numbers: ");
198     numbers.stream().forEach(n -> System.out.print(n + " "));
199
200     // 2. collect() - Accumulates elements into collection
201     System.out.println("\n\n2. collect() - Accumulates elements:");
202     Set<Integer> numberSet = numbers.stream().collect(Collectors.
203         toSet());
204     System.out.println("    As Set: " + numberSet);
205
206     // 3. toArray() - Converts to array
207     System.out.println("\n3. toArray() - Converts to array:");
208     Integer[] array = numbers.stream().toArray(Integer[]::new);
209     System.out.println("    Array: " + Arrays.toString(array));
210
211     // 4. reduce() - Combines elements
212     System.out.println("\n4. reduce() - Combines elements:");

```

```

209 Optional<Integer> sum = numbers.stream().reduce((a, b) -> a + b
    );
210 Optional<Integer> product = numbers.stream().reduce((a, b) -> a
    * b);
211 Integer sumWithIdentity = numbers.stream().reduce(0, (a, b) ->
    a + b);
212
213 System.out.println("    Sum: " + sum.orElse(0));
214 System.out.println("    Product: " + product.orElse(0));
215 System.out.println("    Sum with identity: " + sumWithIdentity);
216
217 // 5. min() and max() - Finds min/max element
218 System.out.println("\n5. min() and max():");
219 Optional<Integer> min = numbers.stream().min(Integer::compare);
220 Optional<Integer> max = numbers.stream().max(Integer::compare);
221
222 System.out.println("    Min: " + min.orElse(null));
223 System.out.println("    Max: " + max.orElse(null));
224
225 // 6. count() - Counts elements
226 System.out.println("\n6. count():");
227 long count = numbers.stream().count();
228 System.out.println("    Count: " + count);
229
230 // 7. anyMatch(), allMatch(), noneMatch() - Boolean checks
231 System.out.println("\n7. anyMatch(), allMatch(), noneMatch():")
    ;
232 boolean anyGreaterThan25 = numbers.stream().anyMatch(n -> n >
    25);
233 boolean allGreaterThan5 = numbers.stream().allMatch(n -> n > 5)
    ;
234 boolean noneGreaterThan100 = numbers.stream().noneMatch(n -> n
    > 100);
235
236 System.out.println("    Any > 25: " + anyGreaterThan25);
237 System.out.println("    All > 5: " + allGreaterThan5);
238 System.out.println("    None > 100: " + noneGreaterThan100);
239
240 // 8. findFirst() and findAny() - Finding elements
241 System.out.println("\n8. findFirst() and findAny():");
242 Optional<Integer> first = numbers.stream().findFirst();
243 Optional<Integer> any = numbers.stream().findAny();
244
245 System.out.println("    First: " + first.orElse(null));
246 System.out.println("    Any: " + any.orElse(null));
247
248 // 9. Collectors utility methods
249 System.out.println("\n9. Advanced Collectors:");
250 Double average = numbers.stream()
251     .collect(Collectors.averagingInt(n -> n)
    );
252 Integer summing = numbers.stream()
253     .collect(Collectors.summingInt(n -> n))
    ;
254 IntSummaryStatistics stats = numbers.stream()
255     .collect(Collectors
    summarizingInt(n -> n));
256

```

```

257     System.out.println("    Average: " + average);
258     System.out.println("    Sum: " + summing);
259     System.out.println("    Statistics: " + stats);
260
261     // 10. Joining strings
262     System.out.println("\n10. Joining strings:");
263     String joined = numbers.stream()
264         .map(String::valueOf)
265         .collect(Collectors.joining(", ", "[", "]"));
266     System.out.println("    Joined: " + joined);
267 }
268
269 // ===== 4. PRIMITIVE STREAMS =====
270 public static void primitiveStreams() {
271     System.out.println("\n=== 4. PRIMITIVE STREAMS ===\n");
272
273     // IntStream
274     System.out.println("IntStream examples:");
275     IntStream.range(1, 6).forEach(n -> System.out.print(n + " "));
276
277     System.out.println("\n\nIntStream operations:");
278     int sum = IntStream.rangeClosed(1, 100).sum();
279     double avg = IntStream.rangeClosed(1, 100).average().orElse(0);
280     OptionalInt max = IntStream.rangeClosed(1, 100).max();
281
282     System.out.println("    Sum 1-100: " + sum);
283     System.out.println("    Average 1-100: " + avg);
284     System.out.println("    Max 1-100: " + max.orElse(0));
285
286     // LongStream
287     System.out.println("\n\nLongStream examples:");
288     long factorial = LongStream.rangeClosed(1, 10)
289         .reduce(1, (a, b) -> a * b);
290     System.out.println("    10! = " + factorial);
291
292     // DoubleStream
293     System.out.println("\n\nDoubleStream examples:");
294     double randomAvg = DoubleStream.generate(Math::random)
295         .limit(1000)
296         .average()
297         .orElse(0);
298     System.out.println("    Average of 1000 random numbers: " +
299         randomAvg);
300
301     // Converting between stream types
302     System.out.println("\n\nConverting between stream types:");
303     List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
304
305     // Convert to IntStream
306     int intSum = numbers.stream()
307         .mapToInt(Integer::intValue)
308         .sum();
309     System.out.println("    Sum using mapToInt: " + intSum);
310
311     // Boxed stream (primitive to object)
312     Stream<Integer> boxedStream = IntStream.range(1, 6).boxed();

```

```

312     List<Integer> boxedList = boxedStream.collect(Collectors.toList
        ());
313     System.out.println("    Boxed list: " + boxedList);
314 }
315
316 // ===== 5. PARALLEL STREAMS =====
317 public static void parallelStreams() {
318     System.out.println("\n=== 5. PARALLEL STREAMS ===\n");
319
320     List<Integer> numbers = IntStream.rangeClosed(1, 1000)
321                                     .boxed()
322                                     .collect(Collectors.toList());
323
324     System.out.println("Processing " + numbers.size() + " numbers"
        );
325
326     // Sequential stream
327     long startTime = System.currentTimeMillis();
328     long seqCount = numbers.stream()
329                             .filter(n -> isPrime(n))
330                             .count();
331     long seqTime = System.currentTimeMillis() - startTime;
332
333     // Parallel stream
334     startTime = System.currentTimeMillis();
335     long parallelCount = numbers.parallelStream()
336                             .filter(n -> isPrime(n))
337                             .count();
338     long parallelTime = System.currentTimeMillis() - startTime;
339
340     System.out.println("\nPrime numbers between 1 and 1000:");
341     System.out.println("    Sequential: " + seqCount + " primes in "
        + seqTime + "ms");
342     System.out.println("    Parallel: " + parallelCount + " primes
        in " + parallelTime + "ms");
343     System.out.println("    Speedup: " + (seqTime / (double)
        parallelTime) + "x");
344
345     // When to use parallel streams
346     System.out.println("\nWhen to use parallel streams:");
347     System.out.println("    Large datasets");
348     System.out.println("    Computationally intensive operations
        ");
349     System.out.println("    Stateless, independent operations");
350     System.out.println("    Small datasets");
351     System.out.println("    Stateful operations");
352     System.out.println("    I/O bound operations");
353
354     // Parallel stream considerations
355     System.out.println("\nParallel stream considerations:");
356     System.out.println("    - Order may not be preserved");
357     System.out.println("    - Thread safety is important");
358     System.out.println("    - Overhead for small tasks");
359     System.out.println("    - Use parallel() judiciously");
360 }
361
362 private static boolean isPrime(int n) {
363     if (n <= 1) return false;

```

```

364     for (int i = 2; i <= Math.sqrt(n); i++) {
365         if (n % i == 0) return false;
366     }
367     return true;
368 }
369
370 // ===== 6. REAL-WORLD EXAMPLES =====
371 public static void realWorldStreamExamples() {
372     System.out.println("\n=== 6. REAL-WORLD STREAM EXAMPLES ===\n");
373     ;
374
375     List<Employee> employees = Arrays.asList(
376         new Employee("Alice", "Engineering", 75000, 28),
377         new Employee("Bob", "Sales", 60000, 35),
378         new Employee("Charlie", "Engineering", 80000, 32),
379         new Employee("Diana", "Marketing", 55000, 29),
380         new Employee("Eve", "Engineering", 90000, 40),
381         new Employee("Frank", "Sales", 65000, 45),
382         new Employee("Grace", "HR", 50000, 30)
383     );
384
385     System.out.println("Employee Data:");
386     employees.forEach(e -> System.out.println("    " + e));
387
388     // Example 1: Find average salary by department
389     System.out.println("\n1. Average salary by department:");
390     Map<String, Double> avgSalaryByDept = employees.stream()
391         .collect(Collectors.groupingBy(
392             Employee::getDepartment,
393             Collectors.averagingDouble(Employee::getSalary)
394         ));
395     avgSalaryByDept.forEach((dept, avg) ->
396         System.out.printf("    %s: $%.2f\n", dept, avg));
397
398     // Example 2: Highest paid employee in each department
399     System.out.println("\n2. Highest paid in each department:");
400     Map<String, Optional<Employee>> topByDept = employees.stream()
401         .collect(Collectors.groupingBy(
402             Employee::getDepartment,
403             Collectors.maxBy(Comparator.comparing(Employee::
404                 getSalary))
405         ));
406     topByDept.forEach((dept, emp) ->
407         System.out.printf("    %s: %s ($%.0f)\n",
408             dept, emp.map(Employee::getName).orElse("None"),
409             emp.map(Employee::getSalary).orElse(0.0)));
410
411     // Example 3: Employees grouped by age range
412     System.out.println("\n3. Employees by age range:");
413     Map<String, List<Employee>> byAgeRange = employees.stream()
414         .collect(Collectors.groupingBy(e -> {
415             if (e.getAge() < 30) return "Under 30";
416             else if (e.getAge() < 40) return "30-39";
417             else return "40+";
418         }));
419     byAgeRange.forEach((range, empList) -> {
420         System.out.println("    " + range + ": " +

```

```

419         empList.stream().map(Employee::getName).collect(
420             Collectors.joining(", "));
421     });
422     // Example 4: Total salary budget by department
423     System.out.println("\n4. Total salary budget by department:");
424     Map<String, Double> budgetByDept = employees.stream()
425         .collect(Collectors.groupingBy(
426             Employee::getDepartment,
427             Collectors.summingDouble(Employee::getSalary)
428         ));
429     budgetByDept.forEach((dept, total) ->
430         System.out.printf("    %s: $%.0f%n", dept, total));
431
432     // Example 5: Find engineering employees with salary > 70000
433     System.out.println("\n5. Engineering employees earning > $70
434         ,000:");
435     List<String> highEarners = employees.stream()
436         .filter(e -> e.getDepartment().equals("Engineering"))
437         .filter(e -> e.getSalary() > 70000)
438         .map(Employee::getName)
439         .sorted()
440         .collect(Collectors.toList());
441     System.out.println("    " + String.join(", ", highEarners));
442
443     // Example 6: Statistics for engineering department
444     System.out.println("\n6. Engineering department statistics:");
445     DoubleSummaryStatistics engStats = employees.stream()
446         .filter(e -> e.getDepartment().equals("Engineering"))
447         .mapToDouble(Employee::getSalary)
448         .summaryStatistics();
449     System.out.printf("    Count: %d%n", engStats.getCount());
450     System.out.printf("    Average: $%.2f%n", engStats.getAverage());
451     ;
452     System.out.printf("    Max: $%.2f%n", engStats.getMax());
453     System.out.printf("    Min: $%.2f%n", engStats.getMin());
454     System.out.printf("    Sum: $%.2f%n", engStats.getSum());
455 }
456
457 // ===== 7. STREAM BEST PRACTICES
458 // =====
459 public static void streamBestPractices() {
460     System.out.println("\n=== 7. STREAM BEST PRACTICES ===\n");
461
462     System.out.println("1. Use Method References:");
463     System.out.println("    Prefer: .map(String::toUpperCase)");
464     System.out.println("    Over:   .map(s -> s.toUpperCase())");
465
466     System.out.println("\n2. Avoid Side Effects:");
467     System.out.println("    Don't modify external state in streams");
468     ;
469     System.out.println("    Use pure functions where possible");
470
471     System.out.println("\n3. Choose Right Data Structure:");
472     System.out.println("    ArrayList      .stream()");
473     System.out.println("    Large datasets .parallelStream()");
474
475     System.out.println("\n4. Order Operations Efficiently:");

```

```

472     System.out.println("    Filter early to reduce elements");
473     System.out.println("    Expensive operations after filtering");
474
475     System.out.println("\n5. Use Primitive Streams for Performance:
476         ");
477     System.out.println("    Use IntStream, LongStream, DoubleStream
478         ");
479     System.out.println("    Avoid boxing/unboxing overhead");
480
481     System.out.println("\n6. Handle Optional Properly:");
482     System.out.println("    Don't call .get() without checking .
483         isPresent()");
484     System.out.println("    Use .orElse(), .orElseGet(), .
485         orElseThrow()");
486
487     System.out.println("\n7. Limit Infinite Streams:");
488     System.out.println("    Always use limit() with generate() or
489         iterate()");
490
491     System.out.println("\n8. Collect Once:");
492     System.out.println("    Don't create multiple terminal
493         operations");
494     System.out.println("    Collect once and reuse");
495 }
496
497 // ===== 8. COMPLETE EXAMPLE =====
498 public static void completeStreamExample() {
499     System.out.println("\n=== 8. COMPLETE STREAM API EXAMPLE ===\n");
500
501     // Create sample data
502     List<Transaction> transactions = Arrays.asList(
503         new Transaction(1001, "GROCERY", 150.50),
504         new Transaction(1002, "ELECTRONICS", 1200.00),
505         new Transaction(1003, "GROCERY", 75.25),
506         new Transaction(1004, "CLOTHING", 200.00),
507         new Transaction(1005, "ELECTRONICS", 850.00),
508         new Transaction(1006, "GROCERY", 45.75),
509         new Transaction(1007, "CLOTHING", 120.00),
510         new Transaction(1008, "GROCERY", 300.00)
511     );
512
513     System.out.println("All Transactions:");
514     transactions.forEach(t -> System.out.println("    " + t));
515
516     // Business Logic using Stream API
517     System.out.println("\n--- Analysis Results ---");
518
519     // 1. Total amount of all transactions
520     double totalAmount = transactions.stream()
521         .mapToDouble(Transaction::getAmount)
522         .sum();
523     System.out.printf("1. Total amount: $%.2f\n", totalAmount);
524
525     // 2. Average transaction amount
526     double avgAmount = transactions.stream()
527         .mapToDouble(Transaction::getAmount)
528         .average()

```

```

523         .orElse(0);
524     System.out.printf("2. Average transaction: $%.2f%n", avgAmount)
525         ;
526     // 3. Highest transaction
527     Optional<Transaction> highest = transactions.stream()
528         .max(Comparator.comparing(Transaction::getAmount));
529     highest.ifPresent(t ->
530         System.out.printf("3. Highest transaction: %s - $%.2f%n",
531             t.getType(), t.getAmount()));
532
533     // 4. Group by type with totals
534     System.out.println("\n4. Transactions by type:");
535     Map<String, DoubleSummaryStatistics> byType = transactions.
536         stream()
537         .collect(Collectors.groupingBy(
538             Transaction::getType,
539             Collectors.summarizingDouble(Transaction::getAmount)
540         ));
541     byType.forEach((type, stats) -> {
542         System.out.printf("    %s:%n", type);
543         System.out.printf("        Count: %d%n", stats.getCount());
544         System.out.printf("        Total: $%.2f%n", stats.getSum());
545         System.out.printf("        Average: $%.2f%n", stats.getAverage
546             ());
547         System.out.printf("        Max: $%.2f%n", stats.getMax());
548     });
549     // 5. Find all expensive transactions (> $500)
550     System.out.println("\n5. Expensive transactions (> $500):");
551     List<Transaction> expensive = transactions.stream()
552         .filter(t -> t.getAmount() > 500)
553         .sorted(Comparator.comparing(Transaction::getAmount).
554             reversed())
555         .collect(Collectors.toList());
556     expensive.forEach(t ->
557         System.out.printf("    ID %d: %s - $%.2f%n",
558             t.getId(), t.getType(), t.getAmount()));
559
560     // 6. Transaction IDs as comma-separated string
561     String transactionIds = transactions.stream()
562         .map(t -> String.valueOf(t.getId()))
563         .collect(Collectors.joining(", ", "[", "]"));
564     System.out.println("\n6. Transaction IDs: " + transactionIds);
565 }
566
567 // ===== MAIN METHOD =====
568 public static void main(String[] args) {
569     System.out.println("=== STREAM API - COMPLETE GUIDE ===\n");
570
571     System.out.println("Stream API provides a functional approach
572         to");
573     System.out.println("processing collections of data in Java.\n");
574     ;
575
576     // 1. Stream Basics

```

```

575     streamBasics();
576
577     // 2. Intermediate Operations
578     intermediateOperations();
579
580     // 3. Terminal Operations
581     terminalOperations();
582
583     // 4. Primitive Streams
584     primitiveStreams();
585
586     // 5. Parallel Streams
587     parallelStreams();
588
589     // 6. Real-world Examples
590     realWorldStreamExamples();
591
592     // 7. Best Practices
593     streamBestPractices();
594
595     // 8. Complete Example
596     completeStreamExample();
597
598     System.out.println("\n=== STREAM API BENEFITS ===");
599     System.out.println("1. Declarative: Say what you want, not how"
600         );
601     System.out.println("2. Composable: Chain operations easily");
602     System.out.println("3. Parallelizable: Easy parallel processing"
603         );
604     System.out.println("4. Lazy Evaluation: Efficient execution");
605     System.out.println("5. Functional: Encourages pure functions");
606
607     System.out.println("\n=== COMMON MISTAKES ===");
608     System.out.println("1. Reusing streams (they're one-time use)"
609         );
610     System.out.println("2. Forgetting terminal operations (nothing
611         happens)");
612     System.out.println("3. Modifying source collection during
613         stream ops");
614     System.out.println("4. Ignoring ordering in parallel streams");
615     System.out.println("5. Not handling Optional properly");
616
617     System.out.println("\n=== WHEN TO USE STREAMS ===");
618     System.out.println("    Processing collections of data");
619     System.out.println("    Transformations and filtering");
620     System.out.println("    Aggregations and summaries");
621     System.out.println("    Parallel processing needs");
622     System.out.println("    Functional programming style");
623
624     System.out.println("\n=== WHEN NOT TO USE STREAMS ===");
625     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");
}

```

```

626 // ===== SUPPORTING CLASSES =====
627 static class Employee {
628     private String name;
629     private String department;
630     private double salary;
631     private int age;
632
633     public Employee(String name, String department, double salary,
634         int age) {
635         this.name = name;
636         this.department = department;
637         this.salary = salary;
638         this.age = age;
639     }
640
641     public String getName() { return name; }
642     public String getDepartment() { return department; }
643     public double getSalary() { return salary; }
644     public int getAge() { return age; }
645
646     @Override
647     public String toString() {
648         return String.format("%-10s %-15s $%8.2f age:%2d",
649             name, department, salary, age);
650     }
651
652     static class Transaction {
653         private int id;
654         private String type;
655         private double amount;
656
657         public Transaction(int id, String type, double amount) {
658             this.id = id;
659             this.type = type;
660             this.amount = amount;
661         }
662
663         public int getId() { return id; }
664         public String getType() { return type; }
665         public double getAmount() { return amount; }
666
667         @Override
668         public String toString() {
669             return String.format("ID:%4d %-12s $%7.2f", id, type,
670                 amount);
671         }
672     }
}

```

Listing 4: Stream API - Complete Guide Program

## 5 Practical Integration Example

```

1 /*
*****

```

```

2  * PROGRAM: JDBCLambdaStreamIntegration.java
3  *
4  * PURPOSE: This program demonstrates the integration of JDBC, Lambda
5  *           Expressions,
6  *           and Stream API to create modern, efficient database
7  *           applications.
8  *           It shows how functional programming concepts can enhance
9  *           traditional
10 *           database operations.
11 *
12 * PROBLEM SOLVED: Traditional database programming in Java suffers
13 *           from:
14 *
15 *           1. Verbose boilerplate code for CRUD operations
16 *           2. Complex data transformation logic
17 *           3. Difficulty in composing database operations
18 *           4. Inefficient data processing patterns
19 *
20 *           This integration solves these by:
21 *           1. Using lambdas for concise database operations
22 *           2. Applying streams for efficient data processing
23 *           3. Composing operations in a functional style
24 *           4. Enabling parallel database processing
25 *
26 * WHY NECESSARY: Modern enterprise applications require:
27 *           1. Clean, maintainable database code
28 *           2. Efficient data processing pipelines
29 *           3. Functional programming patterns for scalability
30 *           4. Integration of modern Java features with legacy
31 *           systems
32 *           5. Industry-standard patterns for data access
33 *
34 * KEY CONCEPTS DEMONSTRATED:
35 * - JDBC operations enhanced with lambdas
36 * - Stream processing of ResultSet data
37 * - Functional composition of database queries
38 * - Parallel processing of database results
39 * - Real-world employee management system
40 * - Best practices for integrated database programming
41 *****
42 */
43
44 import java.sql.*;
45 import java.util.*;
46 import java.util.stream.*;
47 import java.util.function.*;
48
49 public class JDBCLambdaStreamIntegration {
50
51     private static final String URL = "jdbc:mysql://localhost:3306/
52         company";
53     private static final String USER = "root";
54     private static final String PASSWORD = "password";
55
56     // ===== DATABASE SETUP =====
57     public static void setupDatabase() {
58         String createTableSQL =
59             "CREATE TABLE IF NOT EXISTS employees (" +

```

```

52         "id INT PRIMARY KEY AUTO_INCREMENT, " +
53         "name VARCHAR(100) NOT NULL, " +
54         "department VARCHAR(50), " +
55         "salary DECIMAL(10,2), " +
56         "age INT, " +
57         "hire_date DATE, " +
58         "active BOOLEAN DEFAULT true)";
59
60     String insertDataSQL =
61         "INSERT INTO employees (name, department, salary, age,
62             hire_date) VALUES " +
63         "('John Smith', 'Engineering', 75000.00, 30, '2020-01-15'),
64             " +
65         "('Alice Johnson', 'Sales', 65000.00, 28, '2021-03-10'), "
66             +
67         "('Bob Williams', 'Engineering', 82000.00, 35,
68             '2019-05-20'), " +
69         "('Carol Davis', 'Marketing', 55000.00, 32, '2022-07-05'),
70             " +
71         "('David Brown', 'Engineering', 90000.00, 40, '2018-11-30')
72             , " +
73         "('Eve Miller', 'Sales', 70000.00, 29, '2021-09-15'), " +
74         "('Frank Wilson', 'HR', 50000.00, 45, '2020-12-01'), " +
75         "('Grace Moore', 'Engineering', 78000.00, 33, '2022-02-28')
76             , " +
77         "('Henry Taylor', 'Marketing', 60000.00, 38, '2021-06-10'),
78             " +
79         "('Irene Anderson', 'Sales', 72000.00, 31, '2023-01-05')";
80
81     try (Connection conn = DriverManager.getConnection(URL, USER,
82         PASSWORD);
83         Statement stmt = conn.createStatement()) {
84
85         // Create table
86         stmt.executeUpdate(createTableSQL);
87         System.out.println("Table created/verified");
88
89         // Clear existing data
90         stmt.executeUpdate("DELETE FROM employees");
91
92         // Insert sample data
93         stmt.executeUpdate(insertDataSQL);
94         System.out.println("Sample data inserted");
95
96     } catch (SQLException e) {
97         System.out.println("Database setup error: " + e.getMessage
98             ());
99     }
100
101     // ===== EMPLOYEE CLASS =====
102     static class Employee {
103         private int id;
104         private String name;
105         private String department;
106         private double salary;
107         private int age;
108         private Date hireDate;

```

```

100     private boolean active;
101
102     public Employee(int id, String name, String department,
103                    double salary, int age, Date hireDate, boolean
104                    active) {
105         this.id = id;
106         this.name = name;
107         this.department = department;
108         this.salary = salary;
109         this.age = age;
110         this.hireDate = hireDate;
111         this.active = active;
112     }
113
114     // Getters
115     public int getId() { return id; }
116     public String getName() { return name; }
117     public String getDepartment() { return department; }
118     public double getSalary() { return salary; }
119     public int getAge() { return age; }
120     public Date getHireDate() { return hireDate; }
121     public boolean isActive() { return active; }
122
123     @Override
124     public String toString() {
125         return String.format("%2d %-15s %-12s $%8.2f %3dyrs %tF %s"
126                               ,
127                               id, name, department, salary, age, hireDate, active ? "
128                               Active" : "Inactive");
129     }
130
131     // ===== DATABASE OPERATIONS WITH LAMBDAS
132     // =====
133     public static List<Employee> getAllEmployees() {
134         List<Employee> employees = new ArrayList<>();
135
136         String sql = "SELECT * FROM employees";
137
138         try (Connection conn = DriverManager.getConnection(URL, USER,
139                 PASSWORD);
140             Statement stmt = conn.createStatement();
141             ResultSet rs = stmt.executeQuery(sql)) {
142
143             while (rs.next()) {
144                 Employee emp = new Employee(
145                     rs.getInt("id"),
146                     rs.getString("name"),
147                     rs.getString("department"),
148                     rs.getDouble("salary"),
149                     rs.getInt("age"),
150                     rs.getDate("hire_date"),
151                     rs.getBoolean("active")
152                 );
153                 employees.add(emp);
154             }
155         } catch (SQLException e) {

```

```

153         System.out.println("Error fetching employees: " + e.
154             getMessage());
155     }
156     return employees;
157 }
158
159 public static List<Employee> getEmployeesByCondition(Predicate<
160     Employee> condition) {
161     return getAllEmployees().stream()
162         .filter(condition)
163         .collect(Collectors.toList());
164 }
165
166 public static double calculateDepartmentSalary(String department,
167     Function<Employee,
168     Double> mapper) {
169     return getAllEmployees().stream()
170         .filter(e -> e.getDepartment().equals(
171             department))
172         .map(mapper)
173         .reduce(0.0, Double::sum);
174 }
175
176 // ===== STREAM OPERATIONS ON DATABASE DATA
177 // =====
178 public static void performAnalysis() {
179     System.out.println("\n=== EMPLOYEE DATA ANALYSIS USING STREAMS
180         ===\n");
181
182     List<Employee> employees = getAllEmployees();
183
184     System.out.println("All Employees:");
185     employees.forEach(System.out::println);
186
187     // 1. Group employees by department
188     System.out.println("\n1. Employees grouped by department:");
189     Map<String, List<Employee>> byDepartment = employees.stream()
190         .collect(Collectors.groupingBy(Employee::getDepartment));
191
192     byDepartment.forEach((dept, empList) -> {
193         System.out.println("\n    " + dept + " Department:");
194         empList.forEach(e -> System.out.println("        " + e.getName
195             ()));
196     });
197
198     // 2. Average salary by department
199     System.out.println("\n2. Average salary by department:");
200     Map<String, Double> avgSalaryByDept = employees.stream()
201         .collect(Collectors.groupingBy(
202             Employee::getDepartment,
203             Collectors.averagingDouble(Employee::getSalary)
204         ));
205
206     avgSalaryByDept.forEach((dept, avg) ->
207         System.out.printf("    %-12s: $%.2f\n", dept, avg));
208
209     // 3. Top 3 highest paid employees

```

```

204 System.out.println("\n3. Top 3 highest paid employees:");
205 employees.stream()
206     .sorted(Comparator.comparingDouble(Employee::getSalary).
207         reversed())
208     .limit(3)
209     .forEach(e -> System.out.printf("    %-15s: $%.2f%n", e.
210         getName(), e.getSalary()));
211
212 // 4. Employees by age group
213 System.out.println("\n4. Employees by age group:");
214 Map<String, List<Employee>> byAgeGroup = employees.stream()
215     .collect(Collectors.groupingBy(e -> {
216         if (e.getAge() < 30) return "Under 30";
217         else if (e.getAge() < 40) return "30-39";
218         else return "40+";
219     }));
220
221 byAgeGroup.forEach((group, empList) -> {
222     System.out.printf("    %s (%d employees):%n", group, empList
223         .size());
224     empList.forEach(e -> System.out.printf("        %s (%d)%n", e.
225         getName(), e.getAge()));
226 });
227
228 // 5. Department with highest total salary
229 System.out.println("\n5. Department salary statistics:");
230 Map<String, Double> totalSalaryByDept = employees.stream()
231     .collect(Collectors.groupingBy(
232         Employee::getDepartment,
233         Collectors.summingDouble(Employee::getSalary)
234     ));
235
236 totalSalaryByDept.entrySet().stream()
237     .sorted(Map.Entry.<String, Double>comparingByValue().
238         reversed())
239     .forEach(entry ->
240         System.out.printf("    %-12s: $%.2f%n", entry.getKey(),
241             entry.getValue()));
242
243 // 6. Find employees with salary above department average
244 System.out.println("\n6. Employees earning above their
245     department average:");
246 employees.stream()
247     .collect(Collectors.groupingBy(Employee::getDepartment))
248     .forEach((dept, empList) -> {
249         double deptAvg = empList.stream()
250             .mapToDouble(Employee::getSalary)
251             .average()
252             .orElse(0);
253
254         System.out.printf("\n    %s (Average: $%.2f):%n", dept,
255             deptAvg);
256         empList.stream()
257             .filter(e -> e.getSalary() > deptAvg)
258             .forEach(e -> System.out.printf("        %-15s: $%.2f%
259                 n", e.getName(), e.getSalary()));
260     });
261 }

```

```

253
254 // ===== DATABASE UPDATES WITH LAMBDA
      =====
255 public static void updateSalaries(Function<Employee, Double>
      salaryCalculator) {
256     String updateSQL = "UPDATE employees SET salary = ? WHERE id =
          ?";
257
258     try (Connection conn = DriverManager.getConnection(URL, USER,
          PASSWORD);
259         PreparedStatement pstmt = conn.prepareStatement(updateSQL)
          ) {
260
261         List<Employee> employees = getAllEmployees();
262
263         employees.forEach(emp -> {
264             try {
265                 double newSalary = salaryCalculator.apply(emp);
266                 pstmt.setDouble(1, newSalary);
267                 pstmt.setInt(2, emp.getId());
268                 pstmt.addBatch();
269             } catch (SQLException e) {
270                 System.out.println("Error updating salary for " +
          emp.getName() + ": " + e.getMessage());
271             }
272         });
273
274         int[] updateCounts = pstmt.executeBatch();
275         System.out.println("Updated " + Arrays.stream(updateCounts)
          .sum() + " salaries");
276
277     } catch (SQLException e) {
278         System.out.println("Error updating salaries: " + e.
          getMessage());
279     }
280 }
281
282 // ===== COMPLEX QUERY WITH STREAM PROCESSING
      =====
283 public static void complexEmployeeReport() {
284     System.out.println("\n=== COMPLEX EMPLOYEE REPORT ===\n");
285
286     List<Employee> employees = getAllEmployees();
287
288     // Create comprehensive report
289     Map<String, Map<String, Object>> report = employees.stream()
290         .collect(Collectors.groupingBy(
291             Employee::getDepartment,
292             Collectors.collectingAndThen(
293                 Collectors.toList(),
294                 empList -> {
295                     Map<String, Object> deptStats = new HashMap<>()
296                         ;
297                     deptStats.put("count", empList.size());
298                     deptStats.put("totalSalary",
          empList.stream().mapToDouble(Employee::
          getSalary).sum());
299                     deptStats.put("averageSalary",

```

```

300         empList.stream().mapToDouble(Employee::
301             getSalary).average().orElse(0));
302     deptStats.put("averageAge",
303         empList.stream().mapToInt(Employee::getAge)
304             .average().orElse(0));
305     deptStats.put("highestPaid",
306         empList.stream().max(Comparator.
307             comparingDouble(Employee::getSalary))
308             .map(Employee::getName).orElse("None
309             "));
310     deptStats.put("employees",
311         empList.stream().map(Employee::getName).
312             collect(Collectors.toList()));
313     return deptStats;
314 }
315 )
316 ));
317
318 // Print report
319 report.forEach((dept, stats) -> {
320     System.out.println("=".repeat(50));
321     System.out.println("DEPARTMENT: " + dept);
322     System.out.println("=".repeat(50));
323     System.out.printf("Employee Count: %d\n", stats.get("count"
324         ));
325     System.out.printf("Total Salary: $%.2f\n", stats.get("
326         totalSalary"));
327     System.out.printf("Average Salary: $%.2f\n", stats.get("
328         averageSalary"));
329     System.out.printf("Average Age: %.1f years\n", stats.get("
330         averageAge"));
331     System.out.printf("Highest Paid: %s\n", stats.get("
332         highestPaid"));
333     System.out.println("Employees: " + String.join(", ", (List<
334         String>) stats.get("employees")));
335     System.out.println();
336 });
337
338 // Overall statistics
339 System.out.println("\n=== OVERALL COMPANY STATISTICS ===");
340 System.out.printf("Total Employees: %d\n", employees.size());
341 System.out.printf("Total Salary Budget: $%.2f\n",
342     employees.stream().mapToDouble(Employee::getSalary).sum());
343 System.out.printf("Average Company Salary: $%.2f\n",
344     employees.stream().mapToDouble(Employee::getSalary).average
345         ().orElse(0));
346 System.out.printf("Average Employee Age: %.1f years\n",
347     employees.stream().mapToInt(Employee::getAge).average().
348         orElse(0));
349 }
350
351 // ===== MAIN METHOD =====
352 public static void main(String[] args) {
353     System.out.println("=== JDBC WITH LAMBDA AND STREAM INTEGRATION
354         ===\n");
355
356     // Setup database
357     setupDatabase();

```

```

344 // Get all employees using traditional JDBC
345 System.out.println("\nFetching employees from database...");
346 List<Employee> allEmployees = getAllEmployees();
347 System.out.println("Total employees: " + allEmployees.size());
348
349 // Example 1: Using lambda predicates to filter employees
350 System.out.println("\n=== EXAMPLE 1: FILTERING WITH LAMBDA
351     PREDICATES ===");
352
353 Predicate<Employee> highSalary = e -> e.getSalary() > 70000;
354 Predicate<Employee> engineeringDept = e -> e.getDepartment().
    equals("Engineering");
355 Predicate<Employee> youngEmployee = e -> e.getAge() < 35;
356
357 System.out.println("\nHigh salary employees (> $70,000):");
358 getEmployeesByCondition(highSalary)
359     .forEach(e -> System.out.println("    " + e.getName() + ": $
    " + e.getSalary()));
360
361 System.out.println("\nEngineering department employees:");
362 getEmployeesByCondition(engineeringDept)
363     .forEach(e -> System.out.println("    " + e.getName()));
364
365 System.out.println("\nYoung engineering employees with high
    salary:");
366 getEmployeesByCondition(engineeringDept.and(highSalary).and(
    youngEmployee))
367     .forEach(e -> System.out.println("    " + e.getName() + " (
    age: " + e.getAge() +
368         ", salary: $" + e.getSalary
    () + ")"));
369
370 // Example 2: Using lambda functions for calculations
371 System.out.println("\n=== EXAMPLE 2: CALCULATIONS WITH LAMBDA
    FUNCTIONS ===");
372
373 Function<Employee, Double> baseSalary = Employee::getSalary;
374 Function<Employee, Double> salaryWithBonus = e -> e.getSalary()
    * 1.1; // 10% bonus
375 Function<Employee, Double> salaryWithRaise = e -> e.getSalary()
    + 5000; // $5000 raise
376
377 System.out.println("\nTotal base salary for Engineering: $" +
    calculateDepartmentSalary("Engineering",
    baseSalary));
378
379 System.out.println("Total salary with 10% bonus for Engineering
    : $" +
380     calculateDepartmentSalary("Engineering",
    salaryWithBonus));
381
382 System.out.println("Total salary with $5000 raise for Sales: $"
    +
383     calculateDepartmentSalary("Sales",
    salaryWithRaise));
384
385 // Example 3: Stream analysis
386 performAnalysis();

```

```

387 // Example 4: Database updates with lambda
388 System.out.println("\n=== EXAMPLE 4: DATABASE UPDATES WITH
      LAMBDA ===");
389 System.out.println("Giving 5% raise to Engineering department
      ...");
390
391 Function<Employee, Double> engineeringRaise = e ->
392     e.getDepartment().equals("Engineering") ? e.getSalary() *
      1.05 : e.getSalary();
393
394 updateSalaries(engineeringRaise);
395
396 // Example 5: Complex report
397 complexEmployeeReport();
398
399 System.out.println("\n=== INTEGRATION BENEFITS ===");
400 System.out.println("1. Concise database operations with lambdas
      ");
401 System.out.println("2. Powerful data processing with Stream API
      ");
402 System.out.println("3. Functional programming style for data
      transformation");
403 System.out.println("4. Type safety and compile-time checking");
404 System.out.println("5. Easy parallelization of database
      processing");
405
406 System.out.println("\n=== BEST PRACTICES ===");
407 System.out.println("1. Use PreparedStatement with lambda
      parameters");
408 System.out.println("2. Process large datasets with streams (
      lazy evaluation)");
409 System.out.println("3. Combine database filtering with stream
      filtering wisely");
410 System.out.println("4. Use transactions for batch updates");
411 System.out.println("5. Handle exceptions properly in lambda
      expressions");
412     }
413 }

```

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:

```
1 new Thread(new Runnable() {  
2     public void run() {  
3         System.out.println("Hello World");  
4     }  
5 }).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 LAMBDA ===

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: 6, Number: 7, Number: 8, Number: 9, Number: 10]
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 filter: 5  
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