



JAVA SE PROGRAMMING LANGUAGE

Lab Guides


Document Code	25e-BM/HR/HDCV/FSOFT
Version	1.1
Effective Date	20/11/2012

RECORD OF CHANGES

No	Effective Date	Change Description	Reason	Reviewer	Approver
1	01/Oct/2018	Create new	Draft		
2	01/Jun/2019	Update template	Fsoft template	DieuNT1	VinhNV

Contents

Unit 16: Building Database Applications with JDBC.....	4
Objectives:.....	4
Product Architecture:	4
Lab Specifications:.....	5
Business Rules:	5
Functional Requirements:	5
Screen Designs:	6
Guidelines:.....	6

	CODE:	JPL.L.L301
	TYPE:	LONG
	LOC:	200
	DURATION:	120 MINUTES

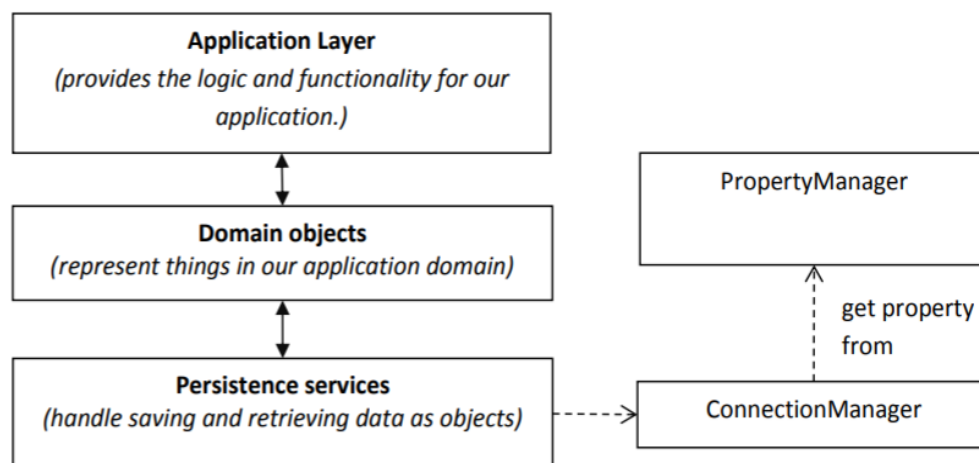
Unit 16: Building Database Applications with JDBC

Objectives:

- » Understand how to connect to database server and all components of JDBC APIs.
- » How to work with Statement, ResultSet, PreparedStatement, CallableStatement, Stored procedures using INPUT/OUTPUT parameters
- » How to call and execute stored procedures with java program.

Product Architecture:

This application will be developed using following architecture:

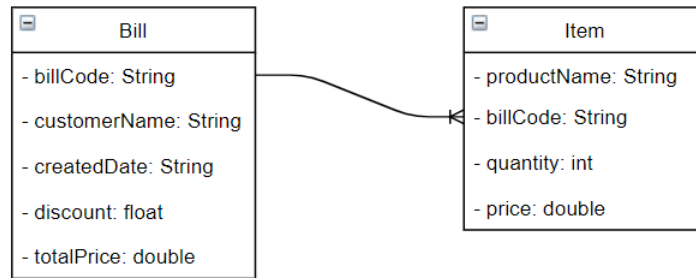


- » The domain layer contains objects and logic related to our application.
- » The persistence layer contains data access objects (DAO) that provide services for accessing persistent data. DAO provide 4 basic services referred to as CRUD:
 - ✓ Create: save new object data to the database.
 - ✓ Retrieve: find object data in the database and recreate objects. There may be several methods for this service to enable different forms of object lookup.
 - ✓ Update: update data for an object already saved to the database.
 - ✓ Delete: delete object data from the database.

Write a program that simulates the functions of the sales system.

Lab Specifications:

For the hierarchy below, the trainee will create java classes that will implement this class diagram. Your classes should be able to show relationship between the entities.



Create a class called **Bill** with the following information:

- » Five private instance variables: billCode (String), customerName (String), createdDate (String), discount(float), totalPrice(double).
- » Default constructor, getter and setter method. Also overriding toString method().

And a class called **Item** with the following information:

- » Four private instance variables: billCode (String), productName (String), quantity (int), price(double).
- » Default constructor, getter and setter method. Also overriding toString method().

Business Rules:

- » Bill code: start with B letter follows 4 digit numbers (i.e. B0000).
- » One bill will has multiple items and a customer can has several bills.

Functional Requirements:

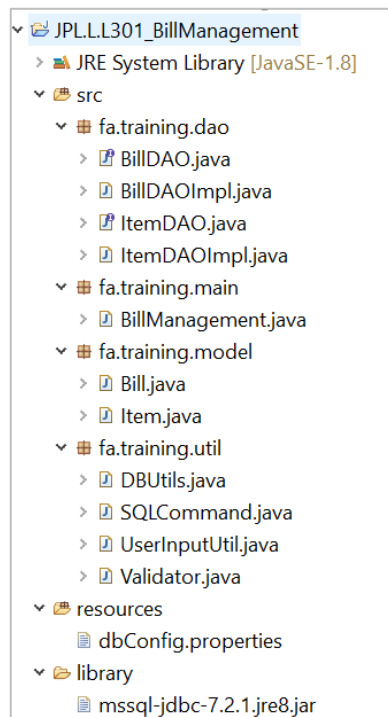
- a. Create a new bill and save into the database using Stored Procedure (method named **boolean** `saveBill(Bill bill)`).
- b. Add one or more item(s) for a bill into the database (method named **boolean** `addItem(final List<Item> items)`).
- c. Display all bills from the database, sorted by created date (method named `List<Bill> getAll()`).
- d. Display all bills which belongs to a specific customer, sorted by created date (method named `List<Bill> findBillsByCustomerName(final String customerName)`).
- e. Display all items from a specific bill, sorted by item name using function (method named `Bill findItemsByBillCode(final String billCode)`).
- f. Delete one or more item(s) from a bill using Stored Procedure (method named **boolean** `deleteItems(final List<Item> items)`).

Screen Designs:

```
-----Menu-----  
1. Create new bill  
2. Add one or more item(s) into a specific bill  
3. Delete one or more item(s) from a bill  
4. Display all bills, sorted by created date  
5. Display customer's bills, sorted by created date  
6. Display items from a specific bill  
7. Exit  
Enter your choice:
```

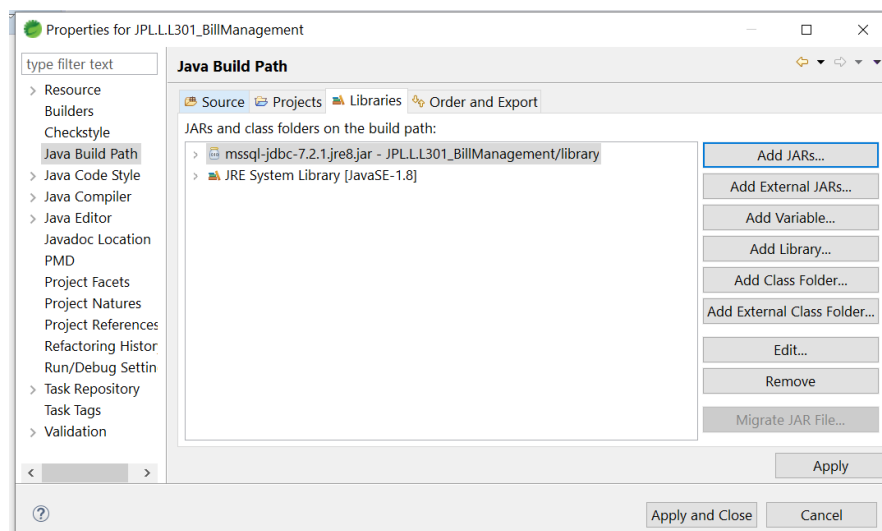
Guidelines:

- » Step 1. Create a new project named **JPL.L.L301_BillManagement**



- » Step 2. Add **mssql-jdbc-7.2.1.jre8.jar** to Java build path.

Right click on the project, choose **Properties -> Java Build Path -> Add JARs**.



- » Step 3. Create *Source folder* named **resources** with the following property files:

dbConfig.properties file:

```
dbConfig.properties
1 driver=com.microsoft.sqlserver.jdbc.SQLServerDriver
2 url=jdbc:sqlserver://localhost:1433;databaseName=BillManagement
3 userName=sa
4 password=12345678
```

- » Step 4. Create package *fa.training.model* that contains classes named **Bill** class and **Item** class as follows:

Bill class:

```
1. public class Bill implements Serializable {
2.     private static final long serialVersionUID = 1L;
3.
4.     private String billCode;
5.     private String customerName;
6.     private String createdDate;
7.     private float discount;
8.     private double totalPrice;
9.
10.    public Bill() {
11.        super();
12.    }
13.
14.    public Bill(String billCode, String customerName, String createdDate,
15.        float discount, double totalPrice) {
16.        super();
17.        this.billCode = billCode;
18.        this.customerName = customerName;
19.        this.createdDate = createdDate;
20.        this.discount = discount;
21.        this.totalPrice = totalPrice;
22.    }
23.
24.    // getter and setter
25.    // override toString() method
26. }
```

Item class:

```
1. public class Item implements Serializable {
2.     private static final long serialVersionUID = 1L;
3.     private String productName;
4.     private String billCode;
5.     private int quantity;
6.     private double price;
7.
8.     public Item() {
9.         super();
10.    }
11.
12.    public Item(String productName, String billCode, int quantity, double price) {
13.        super();
14.        this.productName = productName;
15.        this.billCode = billCode;
16.        this.quantity = quantity;
17.        this.price = price;
18.    }
19.
20.    //getter and setter
21.    // override toString() method
22. }
```

- » Step 5. Create package *fa.training.util* that contains classes named **DBUtils**, **UserInputUtil** and **Validator** class as follows:

DBUtils class:

```
1. public class DBUtils {
2.
3.     private static DBUtils instance;
4.     private Connection connection;
5.
6.     private DBUtils() {
7.         Properties properties = new Properties();
8.
9.         try {
10.             properties.load(
11.                 DBUtils.class.getResourceAsStream("/dbConfig.properties"));
12.
13.             String driver = properties.getProperty("driver");
14.             String url = properties.getProperty("url");
15.             String userName = properties.getProperty("userName");
16.             String password = properties.getProperty("password");
17.
18.             Class.forName(driver);
19.
20.             connection = DriverManager.getConnection(url, userName, password);
21.
22.         } catch (ClassNotFoundException | SQLException | IOException e) {
23.             e.printStackTrace();
24.         }
25.     }
26.     /**
27.      * Get the connection from the instance
28.      *
29.      * @return {@link Connection}
30.      */
31.     public Connection getConnection() {
32.         return connection;
33.     }
34.
35.     /**
36.      * Create new instance which connects with the database.
37.      *
38.      * @return DBUtils
39.      * @throws SQLException if connection false.
40.      */
41.     public static DBUtils getInstance() throws SQLException {
42.         if (instance == null || instance.getConnection().isClosed()) {
43.             instance = new DBUtils();
44.         }
45.         return instance;
46.     }
47. }
```

Validator class:

```
1. public class Validator {
2.     /**
3.      * Check bill code follow the pattern.
4.      *
5.      * @method isValidBillCode
6.      * @param billCode
7.      * @return true if bill code is valid, else false
8.      */
9.     public static boolean isValidBillCode(String billCode) {
10.         return Pattern.matches("^([B])[0-9]{4}$", billCode);
11.     }
12. }
```


UserInputUtil class:

```
13. package fa.training.util;
14.
15. import java.util.Scanner;
16. public class UserInputUtil {
17.
18.     /**
19.      * Get value type integer from console.
20.      *
21.      * @method inputTypeInt
22.      * @param value
23.      */
24.     public static int inputTypeInt(String value) {
25.         int intValue = 0;
26.         do {
27.             try {
28.                 intValue = Integer.parseInt(value);
29.             } catch (Exception e) {
30.                 System.out.println("Please input int value!");
31.             }
32.             break;
33.         } while (true);
34.         return intValue;
35.     }
36.
37.     /**
38.      * Get value type float from console.
39.      *
40.      * @method inputTypeFloat
41.      * @param value
42.      */
43.     public static float inputTypeFloat(String value) {
44.         float floatValue = 0;
45.         do {
46.             try {
47.                 floatValue = Float.parseFloat(value);
48.             } catch (Exception e) {
49.                 System.out.println("Please input float value!");
50.             }
51.             break;
52.         } while (true);
53.         return floatValue;
54.     }
55.
56.     /**
57.      * Get value type double from console.
58.      *
59.      * @method inputTypeDouble
60.      * @param value
61.      */
62.     public static double inputTypeDouble(String value) {
63.         double doubleValue = 0;
64.         do {
65.             try {
66.                 doubleValue = Double.parseDouble(value);
67.             } catch (Exception e) {
68.                 System.out.println("Please input double value!");
69.             }
70.             break;
71.         } while (true);
72.         return doubleValue;
73.     }
74.
75.     /**
76.      * Get a valid bill code from console.
77.      *
78.      * @method checkBillCode
```

```

79.  * @param scanner
80.  * @return
81.  */
82.  public static String checkBillCode(Scanner scanner) {
83.      String billCode;
84.
85.      System.out.println("Enter bill code:");
86.      billCode = scanner.nextLine();
87.
88.      while (!Validator.isValidBillCode(billCode)) {
89.          System.out.println("Invalid bill code: (example: B0000)");
90.          billCode = scanner.nextLine();
91.      }
92.
93.      return billCode;
94.  }
95. }

```

SQLCommand class:

```

1.  public class SQLCommand {
2.      public static String BILL_QUERY_FIND_ALL =
3.          "SELECT *, dbo.udf_ComputeBillTotal(bill_code) AS total_price FROM Bill";
4.      public static String BILL_QUERY_ADD = "{CALL usp_AddBill(?, ?, ?, ?, ?)}";
5.      public static String BILL_QUERY_DELETE = "{CALL usp_DeleteBill(?, ?)}";
6.      public static String BILL_QUERY_FIND_BY_CODE =
7.          "SELECT *, dbo.udf_ComputeBillTotal(bill_code) AS total_price
8.          FROM Bill WHERE bill_code=?";
9.      public static String BILL_QUERY_FIND_BY_CUSTOMER_NAME =
10.         "SELECT *, dbo.udf_ComputeBillTotal(bill_code) AS total_price
11.         FROM Bill WHERE customer_name=?";
12.     public static String ITEM_QUERY_FIND_ALL =
13.         "SELECT * FROM dbo.udf_FindItemsByBillCode(?)";
14.     public static String ITEM_QUERY_ADD =
15.         "INSERT INTO Item(product_name, bill_code, quantity, price) VALUES (?, ?, ?, ?)";
16.     public static String ITEM_QUERY_DELETE =
17.         "DELETE FROM Item WHERE bill_code=? AND product_name=?";
18.     public static String ITEM_QUERY_FIND_CODE_AND_PRODUCT_NAME =
19.         "SELECT * FROM Item WHERE bill_code=? AND product_name=?";
20. }

```

- » Step 6. Create package *fa.training.dao* that contains **BillIDAO**, **ItemDAO** interfaces and classes named **BillIDAOImpl**, **ItemDAOImpl** class as follows:

BillIDAO interface:

```

1.  package fa.training.dao;
2.
3.  import java.sql.SQLException;
4.  import java.util.List;
5.
6.  import fa.training.model.Bill;
7.
8.  public interface BillIDAO {
9.
10.     /**
11.      * Execute a query to get all bills from database.
12.      *
13.      * @method getAll
14.      * @return list of bills
15.      * @throws SQLException
16.      */
17.     List<Bill> getAll() throws SQLException;
18.
19.     /**
20.      * Call a stored procedure to save a bill to database.
21.      *
22.      * @method saveBill
23.      * @param bill

```

```
24.     * @return true if inserts success to database, else false
25.     * @throws SQLException
26.     */
27.     boolean saveBill(Bill bill) throws SQLException;
28.
29.     /**
30.     * Execute a query to retrieve a bill by its code.
31.     *
32.     * @method findBillsByBillCode
33.     * @param billCode
34.     * @return bill if found, else null
35.     * @throws SQLException
36.     */
37.     Bill findBillsByBillCode(String billCode) throws SQLException;
38.
39.     /**
40.     * Execute a query to retrieve bills by its customer name.
41.     *
42.     * @method findBillsByCustomerName
43.     * @param customerName
44.     * @return list of bills
45.     * @throws SQLException
46.     */
47.     List<Bill> findBillsByCustomerName(String customerName) throws SQLException;
48.
49. }
```

ItemDAO interface:

```
1. package fa.training.dao;
2.
3. import java.sql.SQLException;
4. import java.util.List;
5.
6. import fa.training.model.Item;
7.
8. public interface ItemDAO {
9.
10.     /**
11.     * This method is for saving items to the database, using batch.
12.     *
13.     * @method addItems
14.     * @param items
15.     * @return true if saves success, else false
16.     * @throws SQLException
17.     */
18.     boolean addItems(List<Item> items) throws SQLException;
19.
20.     /**
21.     * This method is for deleting items from the database, using batch.
22.     *
23.     * @method deleteItems
24.     * @param items
25.     * @return true if deletes success, else false
26.     * @throws SQLException
27.     */
28.     boolean deleteItems(List<Item> items) throws SQLException;
29.
30.     /**
31.     * Execute a query to get all items of a specific bill, using batch.
32.     *
33.     * @method getAllByBillCode
34.     * @param billCode
35.     * @return list of items
36.     * @throws SQLException
37.     */
38.     List<Item> getAllByBillCode(String billCode) throws SQLException;
39. }
```

```
40.  /**
41.   * Execute a query to check an item was exist or not.
42.   *
43.   * @method checkItemExist
44.   * @param item
45.   * @return true if exist, else false
46.   * @throws SQLException
47.   */
48.  boolean checkItemExist(Item item) throws SQLException;
49. }
```

BillDaoImpl class:

```
1.  package fa.training.dao;
2.
3.  //Imports
4.
5.  /**
6.   * author Duy Bach.
7.   *
8.   * @time 4:04:17 PM
9.   * @date Jun 16, 2019
10.  */
11. public class BillDAOImpl implements BillDAO {
12.
13.     private Connection connection = null;
14.     private PreparedStatement preparedStatement = null;
15.     private CallableStatement caStatement = null;
16.     private ResultSet results = null;
17.
18.     @Override
19.     public List<Bill> getAll() throws SQLException {
20.         List<Bill> bills = new ArrayList<>();
21.         Bill bill = null;
22.         try {
23.             connection = DBUtils.getInstance().getConnection();
24.             preparedStatement =
25.                 connection.prepareStatement(SQLCommand.BILL_QUERY_FIND_ALL);
26.             results = preparedStatement.executeQuery();
27.             while (results.next()) {
28.                 bill = new Bill();
29.
30.                 bill.setBillCode(results.getString("bill_code").trim());
31.                 bill.setCustomerName(results.getString("customer_name"));
32.                 bill.setCreatedDate(results.getString("created_date"));
33.                 bill.setDiscount(results.getInt("discount"));
34.                 bill.setTotalPrice(results.getDouble("total_price"));
35.                 bills.add(bill);
36.             }
37.         } finally {
38.             try {
39.                 if (connection != null) {
40.                     connection.close();
41.                 }
42.                 if (preparedStatement != null) {
43.                     preparedStatement.close();
44.                 }
45.             } catch (SQLException e) {
46.                 e.printStackTrace();
47.             }
48.         }
49.         return bills;
50.     }
51.
52.     @Override
53.     public boolean saveBill(Bill bill)
54.         throws SQLException {
55.         boolean check = false;
56.     }
```

```
57.         try {
58.             connection = DBUtils.getInstance().getConnection();
59.             caStatement = connection.prepareStatement(SQLCommand.BILL_QUERY_ADD);
60.
61.             caStatement.setString(1, bill.getBillCode());
62.             caStatement.setString(2, bill.getCustomerName());
63.             caStatement.setString(3, bill.getCreatedDate());
64.             caStatement.setFloat(4, bill.getDiscount());
65.             caStatement.registerOutParameter(5, Types.INTEGER);
66.             caStatement.execute();
67.             if (caStatement.getInt(5) == 1) {
68.                 check = true;
69.             }
70.
71.         } finally {
72.             try {
73.                 if (connection != null) {
74.                     connection.close();
75.                 }
76.                 if (caStatement != null) {
77.                     caStatement.close();
78.                 }
79.             } catch (SQLException e) {
80.                 e.printStackTrace();
81.             }
82.         }
83.         return check;
84.     }
85.
86.     @Override
87.     public Bill findBillsByBillCode(final String billCode)
88.         throws SQLException {
89.         Bill bill = null;
90.         try {
91.             connection = DBUtils.getInstance().getConnection();
92.             preparedStatement =
93.                 connection.prepareStatement(SQLCommand.BILL_QUERY_FIND_BY_CODE);
94.             preparedStatement.setString(1, billCode);
95.             results = preparedStatement.executeQuery();
96.             if (results.next()) {
97.                 bill = new Bill();
98.
99.                 bill.setBillCode(results.getString("bill_code").trim());
100.                bill.setCustomerName(results.getString("customer_name"));
101.                bill.setCreatedDate(results.getString("created_date"));
102.                bill.setDiscount(results.getInt("discount"));
103.                // bill.setTotalPrice(results.getDouble("total_price"));
104.            }
105.        } finally {
106.            try {
107.                if (connection != null) {
108.                    connection.close();
109.                }
110.                if (preparedStatement != null) {
111.                    preparedStatement.close();
112.                }
113.            } catch (SQLException e) {
114.                e.printStackTrace();
115.            }
116.        }
117.        return bill;
118.    }
119.
120.    /**
121.     * Execute a query to retrieve bills by its customer name.
122.     *
123.     * @method findBillsByCustomerName
124.     * @param customerName
```

```
125.      * @return list of bills
126.      * @throws SQLException
127.      */
128.      public List<Bill> findBillsByCustomerName(final String customerName)
129.                                          throws SQLException {
130.          List<Bill> bills = new ArrayList<>();
131.          Bill bill = null;
132.          try {
133.              connection = DBUtils.getInstance().getConnection();
134.              preparedStatement =
135.                  connection.prepareStatement(
136.                      SQLCommand.BILL_QUERY_FIND_BY_CUSTOMER_NAME);
137.              preparedStatement.setString(1, customerName);
138.              results = preparedStatement.executeQuery();
139.              while (results.next()) {
140.                  bill = new Bill();
141.
142.                  bill.setBillCode(results.getString("bill_code").trim());
143.                  bill.setCustomerName(results.getString("customer_name"));
144.                  bill.setCreatedDate(results.getString("created_date"));
145.                  bill.setDiscount(results.getInt("discount"));
146.                  bill.setTotalPrice(results.getDouble("total_price"));
147.
148.                  bills.add(bill);
149.              }
150.          } finally {
151.              try {
152.                  if (connection != null) {
153.                      connection.close();
154.                  }
155.                  if (preparedStatement != null) {
156.                      preparedStatement.close();
157.                  }
158.              } catch (SQLException e) {
159.                  e.printStackTrace();
160.              }
161.          }
162.          return bills;
163.      }
164.
165.  }
```

ItemDAOImpl class:

```
1.  package fa.training.dao;
2.
3.  // Imports
4.
5.  /**
6.   * author Duy Bach.
7.   *
8.   * @time 4:04:50 PM
9.   * @date Jun 16, 2019
10.  */
11. public class ItemDAOImpl implements ItemDAO {
12.     private Connection connection = null;
13.     private PreparedStatement preparedStatement = null;
14.     private ResultSet results = null;
15.
16.     @Override
17.     public boolean addItem(final List<Item> items) throws SQLException {
18.
19.         boolean check = false;
20.         int results[] = null;
21.         try {
22.             connection = DBUtils.getInstance().getConnection();
23.             connection.setAutoCommit(false);
24.             preparedStatement =
25.                 connection.prepareStatement(SQLCommand.ITEM_QUERY_ADD);
```

```
26.
27.         items.stream().forEach((item) -> {
28.             try {
29.                 preparedStatement.setString(1, item.getProductName());
30.                 preparedStatement.setString(2, item.getBillCode().trim());
31.                 preparedStatement.setInt(3, item.getQuantity());
32.                 preparedStatement.setDouble(4, item.getPrice());
33.
34.                 preparedStatement.addBatch();
35.             } catch (SQLException e) {
36.                 e.printStackTrace();
37.             }
38.         });
39.         results = preparedStatement.executeBatch();
40.         connection.commit();
41.     } finally {
42.         try {
43.             if (connection != null) {
44.                 connection.close();
45.             }
46.             if (preparedStatement != null) {
47.                 preparedStatement.close();
48.             }
49.         } catch (SQLException e) {
50.             e.printStackTrace();
51.         }
52.     }
53.
54.     if (results.length > 0) {
55.         check = true;
56.     }
57.     return check;
58. }
59.
60. @Override
61. public boolean deleteItems(final List<Item> items) throws SQLException {
62.     boolean check = false;
63.     int results[] = null;
64.     try {
65.         connection = DBUtils.getInstance().getConnection();
66.         connection.setAutoCommit(false);
67.         preparedStatement =
68.             connection.prepareStatement(SQLCommand.ITEM_QUERY_DELETE);
69.
70.         items.stream().forEach((item) -> {
71.             try {
72.
73.                 preparedStatement.setString(1, item.getBillCode());
74.                 preparedStatement.setString(2, item.getProductName());
75.
76.                 preparedStatement.addBatch();
77.             } catch (SQLException e) {
78.                 e.printStackTrace();
79.             }
80.         });
81.         results = preparedStatement.executeBatch();
82.         connection.commit();
83.     } finally {
84.         try {
85.             if (connection != null) {
86.                 connection.close();
87.             }
88.             if (preparedStatement != null) {
89.                 preparedStatement.close();
90.             }
91.         } catch (SQLException e) {
92.             e.printStackTrace();
93.         }
94.     }
```

```
94.     }
95.
96.     if (results.length > 0) {
97.         check = true;
98.     }
99.     return check;
100. }
101.
102. @Override
103. public List<Item> getAllByBillCode(String billCode)
104.     throws SQLException {
105.
106.     List<Item> items = new ArrayList<>();
107.     Item item = null;
108.     try {
109.         connection = DBUtils.getInstance().getConnection();
110.         preparedStatement =
111.             connection.prepareStatement(SQLCommand.ITEM_QUERY_FIND_ALL);
112.         preparedStatement.setString(1, billCode);
113.         results = preparedStatement.executeQuery();
114.         while (results.next()) {
115.             item = new Item();
116.
117.             item.setBillCode(results.getString("bill_code"));
118.             item.setProductName(results.getString("product_name"));
119.             item.setQuantity(results.getInt("quantity"));
120.             item.setPrice(results.getDouble("price"));
121.
122.             items.add(item);
123.         }
124.     } finally {
125.         try {
126.             if (connection != null) {
127.                 connection.close();
128.             }
129.             if (preparedStatement != null) {
130.                 preparedStatement.close();
131.             }
132.         } catch (SQLException e) {
133.             e.printStackTrace();
134.         }
135.     }
136.     return items;
137. }
138.
139. @Override
140. public boolean checkItemExist(Item item) throws SQLException {
141.     boolean check = false;
142.     try {
143.         connection = DBUtils.getInstance().getConnection();
144.         preparedStatement =
145.             connection.prepareStatement(
146.                 SQLCommand.ITEM_QUERY_FIND_CODE_AND_PRODUCT_NAME);
147.         preparedStatement.setString(1, item.getBillCode());
148.         preparedStatement.setString(2, item.getProductName());
149.         results = preparedStatement.executeQuery();
150.         if (results.next()) {
151.             check = true;
152.         }
153.     } finally {
154.         try {
155.             if (connection != null) {
156.                 connection.close();
157.             }
158.             if (preparedStatement != null) {
159.                 preparedStatement.close();
160.             }
161.         } catch (SQLException e) {
162.             e.printStackTrace();
163.         }
164.     }
165. }
```



```
162.         }
163.     }
164.     return check;
165. }
166.
167. }
```

» Step 7. Create package *fa.training.main* that contains **BillManagement** class as follows:

BillManagement class:

```
1. public class BillManagement {
2.
3.     static BillDAO billDAO = new BillDAOImpl();
4.     static ItemDAO itemDAO = new ItemDAOImpl();
5.
6.     public static void main(String[] args) {
7.         Scanner scanner = new Scanner(System.in);
8.         List<Bill> bills = new ArrayList<>();
9.         List<Item> items = new ArrayList<>();
10.        String billCode;
11.        String loop = "";
12.        Item item = null;
13.        String choice = "";
14.        do {
15.            getMenu();
16.            System.out.println("Enter your choice:");
17.            choice = scanner.nextLine();
18.
19.            switch (choice) {
20.                case "1":
21.                    Bill bill = new Bill();
22.                    do {
23.                        bill.setBillCode(UserInputUtil.checkBillCode(scanner));
24.                    } while (billDAO.findBillsByBillCode(bill.getBillCode()) != null);
25.
26.                    System.out.println("Enter customer name:");
27.                    bill.setCustomerName(scanner.nextLine());
28.
29.                    System.out.println("Enter discount:");
30.                    bill.setDiscount(UserInputUtil.inputTypeFloat(scanner.nextLine()));
31.
32.                    bill.setCreateDate(getCurrentDate());
33.
34.                    boolean check = billDAO.saveBill(bill);
35.                    if (check) {
36.                        System.out.println("Saved success!");
37.                    }
38.                    break;
39.                case "2":
40.                    if (!items.isEmpty()) {
41.                        items.clear();
42.                    }
43.
44.                    billCode = UserInputUtil.checkBillCode(scanner);
45.
46.                    if (billDAO.findBillsByBillCode(billCode) == null) {
47.                        System.out.println("No bill code = " + billCode + " found!");
48.                    } else {
49.                        do {
50.                            item = new Item();
51.
52.                            item.setBillCode(billCode);
53.
54.                            do {
55.                                System.out.println("Enter product name:");
56.                                item.setProductName(scanner.nextLine());
57.                            } while (!checkProductExist(items, item.getProductName()));
```

```
58.
59.         System.out.println("Enter quantity:");
60.         item.setQuantity(UserInputUtil.inputTypeInt(scanner.nextLine()));
61.         System.out.println("Enter product price:");
62.         item.setPrice(UserInputUtil.inputTypeDouble(scanner.nextLine()));
63.
64.         items.add(item);
65.
66.         System.out.println("Do you want to continue adding (Y|N)?");
67.         loop = scanner.nextLine();
68.     } while (loop.charAt(0) == 'Y' || loop.charAt(0) == 'y');
69.
70.     itemDAO.addItem(items);
71. }
72. break;
73. case "3":
74.     if (!items.isEmpty()) {
75.         items.clear();
76.     }
77.     loop = "";
78.     billCode = UserInputUtil.checkBillCode(scanner);
79.
80.     if (billDAO.findBillsByBillCode(billCode) == null) {
81.         System.out.println("No bill code = " + billCode + " found!");
82.     } else {
83.         do {
84.             item = new Item();
85.
86.             item.setBillCode(billCode);
87.
88.             do {
89.                 System.out.println("Enter product name:");
90.                 item.setProductName(scanner.nextLine());
91.             } while (!checkProductExist(items, item.getProductName()));
92.
93.             items.add(item);
94.
95.             System.out.println("Do you want to continue deleting (Y|N)?");
96.             loop = scanner.nextLine();
97.         } while (loop.charAt(0) == 'Y' || loop.charAt(0) == 'y');
98.
99.         itemDAO.deleteItems(items);
100.    }
101.    break;
102. case "4":
103.     bills = billDAO.getAll();
104.     if (bills.isEmpty()) {
105.         System.out.println("No bill found!");
106.     } else {
107.         System.out.println("---List of bills---");
108.
109.         bills.stream().sorted(Comparator.comparing(Bill::getCreatedDate))
110.             .forEach(System.out::println);
111.     }
112.    break;
113. case "5":
114.     System.out.println("Enter customer name:");
115.     String customerName = scanner.nextLine();
116.     bills = billDAO.findBillsByCustomerName(customerName);
117.     if (bills.isEmpty()) {
118.         System.out.println("No bill found!");
119.     } else {
120.         System.out.println("---List of bills---");
121.
122.         bills.stream().sorted(Comparator.comparing(Bill::getCreatedDate))
123.             .forEach(System.out::println);
124.     }
125.    break;
```

```
126.         case "6":
127.             billCode = UserInputUtil.checkBillCode(scanner);
128.
129.             items = itemDAO.getAllByBillCode(billCode);
130.
131.             if (items.isEmpty()) {
132.                 System.out.println("No data found!");
133.             } else {
134.                 System.out.println("---List all items from bill---");
135.                 items.stream().forEach(System.out::println);
136.             }
137.             break;
138.         case "7":
139.             System.exit(0);
140.             break;
141.         default:
142.             System.out.println("Invalid input!");
143.             break;
144.     }
145. } while (true);
146.
147. }
148.
149. public static void getMenu() {
150.     System.out.println("-----Menu-----");
151.     System.out.println("1. Create new bill");
152.     System.out.println("2. Add one or more item(s) into a specific bill");
153.     System.out.println("3. Delete one or more item(s) from a bill");
154.     System.out.println("4. Display all bills, sorted by created date");
155.     System.out.println("5. Display customer's bills,
156.                        sorted by created date");
157.     System.out.println("6. Display items from a specific bill");
158.     System.out.println("7. Exit");
159. }
160.
161. /**
162.  * This method is checked if the product name exist in the list or not
163.  *
164.  * @method checkProductExist
165.  * @param items, productName
166.  * @return true if product name is exist, otherwise false
167.  */
168. private static boolean checkProductExist(final List<Item> items,
169.     final String productName) {
170.     boolean check = items.stream()
171.         .anyMatch((Item item) ->
172.             productName.equals(item.getProductName()));
173.     return check;
174. }
175.
176. public static String getCurrentDate() {
177.     SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
178.     return format.format(new Date());
179. }
180. }
```

----oOo----

THE END