Practices for Lesson 13

Practices Overview

In these practices, you will work with the JavaDB (Derby) database, creating, reading, updating, and deleting data from a SQL database by using the Java JDBC API.

Practice 13-1: Summary Level: Working with the Derby Database and JDBC

Overview

In this practice, you will start the JavaDB (Derby) database, load some sample data using a script, and write an application to read the contents of an employee database table and print the results to the console.

Tasks

1. Create the Employee Database by using the SQL script provided in the resource directory.
   a. Open the Services window by selecting Windows > Services, or by pressing Ctrl-5.
   b. Expand the Databases folder.
   c. Right-click JavaDB and select Start Server.
   d. Right-click JavaDB again and select Create Database.
   e. Enter the following information:

Window/Page Description	Choices or Values
Database Name	EmployeeDB
User Name	public
Password	tiger
Confirm Password	tiger

   f. Click OK
   g. Right-click the connection that you created: `jdbc:derby://localhost:1527/EmployeeDB[public on PUBLIC]` and select Connect.
   h. Select File > Open File.
   i. Browse to `D:\labs\resources` and open the `EmployeeTable.sql` script. The file will open in a SQL Execute window.
   j. Select the connection that you created from the drop-down list, and click the Run-SQL icon  or press Ctrl-Shift-E to run the script.
   k. Expand the `EmployeeDB` connection. You will see that the `PUBLIC` schema is now created. Expand the `PUBLIC` Schema and look at the table `Employee`.
   l. Right-click the connection again and select Execute Command to open another SQL window. Enter the command:
   ```
   select * from Employee
   ```
   and click the Run-SQL icon to see the contents of the `Employee` table.

2. Open the `SimpleJDBCExample` project and run it.
   a. You should see all the records from the Employee table displayed.
3. (Optional) Add a SQL command to add a new Employee record.
   a. Modify the `SimpleJDBCExample` class to add a new Employee record to the database.
   b. The syntax for adding a row in a SQL database is:
      `INSERT INTO <table name> VALUES (<column 1 value>, <column 2 value>, ...)`
   c. Use the Statement `executeUpdate` method to execute the query. What is the return type for this method? What value should the return type be? Test to make sure that the value of the return is correct.

Practice 13-1: Detailed Level: Working with the Derby Database and JDBC

Overview

In this practice, you will start the JavaDB (Derby) database, load some sample data using a script, and write an application to read the contents of an employee database table and print the results to the console.

Tasks

1. Create the Employee Database by using the SQL script provided in the resource directory.
   a. Open the Services Window by selecting Windows > Services, or by pressing Ctrl-5.
   b. Expand the Databases folder.
   c. Right-click JavaDB and select Start Server.
   d. Right-click JavaDB again and select Create Database.
   e. Enter the following information:

Window/Page Description	Choices or Values
Database Name	EmployeeDB
User Name	public
Password	tiger
Confirm Password	tiger

   f. Click OK.
   g. Right-click the connection that you created: `jdbc:derby://localhost:1527/EmployeeDB[public on PUBLIC]` and select Connect.
   h. Select File > Open File.
   i. Browse to `D:\labs\resources` and open the `EmployeeTable.sql` script. The file will open in a SQL Execute window.
   j. Select the connection that you created from the drop-down list and click the Run-SQL icon  or press Ctrl-Shift-E to run the script.
   k. Expand the `EmployeeDB` connection. You will see that the `PUBLIC` schema is now created. Expand the `PUBLIC` Schema, expand Tables, and then expand the table `Employee`.
   l. Right-click the connection again and select Execute Command to open another SQL window. Enter the command:
   ```
   select * from Employee
   ```
   and click the Run-SQL icon to see the contents of the `Employee` table.

2. Open the `SimpleJDBCExample` Project and run it.
   a. Select Windows > Projects, or press Ctrl-1.
   b. Select File > Open Project.
   c. Select `D:\labs\13\practices\SimpleJDBCExample`. `(or your other folder)`
   d. Select “Open as Main Project.”
   e. Click OK.
   f. Expand the Source Packages and test packages and look at the `SimpleJDBCExample.java` class.
   g. Run the project: Right-click the project and select Run, or click the Run icon, or press F6.
   h. You should see all the records from the Employee table displayed.
3. (Optional) Add a SQL command to add a new Employee record.
   a. Modify the `SimpleJDBCExample` class to add a new Employee record to the database.
   b. The syntax for adding a row in a SQL database is:
      `INSERT INTO <table name> VALUES (<column 1 value>, <column 2 value>, ...)`
   c. Use the Statement `executeUpdate` method to execute the query. What is the return type for this method? What value should the return type be? Test to make sure that the value of the return is correct.
   d. Your code may look like this:

```
query = "INSERT INTO Employee VALUES (200, 'Bill',
'Murray','1950-09-21', 150000)";
if (stmt.executeUpdate(query) != 1) {
    System.out.println ("Failed to add a new employee record");
}
```

Note: If you run the application again, it will throw an exception, because this key already exists in the database.

Practice 13-2: Summary Level: Using the Data Access Object Pattern

Overview

In this practice, you will take the existing Employee DAO Memory application and refactor the code to use JDBC instead. The solution from the “Exceptions and Assertions” lesson has been renamed to `EmployeeDAOJDBC`. You will need to create an `EmployeeDAOJDBCImpl` class to replace the `EmployeeDAOMemoryImpl` class, and modify the `EmployeeDAOFactory` to return an instance of your new implementation class instead of the Memory version.

You will not have to alter the other classes. This example illustrates how a well designed Data Access Object application can use an alternative persistence class without significant change.

Tasks

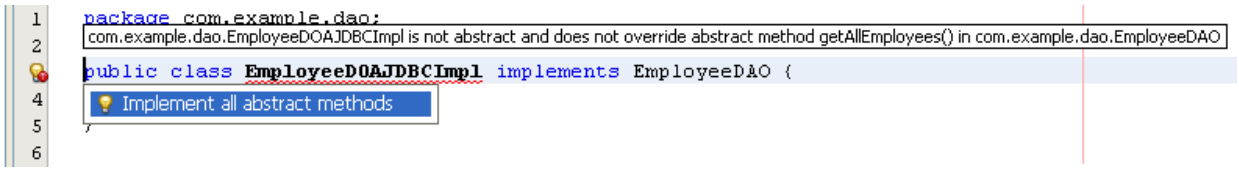
1. Open and examine the `EmployeeDAOJDBC` project in the `D:\labs\13\practices` folder. `(or your other folder)`
   a. In the `com.example.test` package, you see the class `EmployeeTestInteractive`. This class contains the main method and provides a console-based user interface. Through this UI, you will be able to create new records, read all the records, update a record, and delete a record from the Employee database. Note how the main method creates an instance of a Data Access Object (DAO).
   b. In the `com.example.model` package, look at the `Employee` class. This class is a Plain Old Java Object (POJO) that encapsulates all of the data from a single employee record and row in the Employee table. Note that there are no set methods in this class, only get methods. Once an Employee object is created, it cannot be changed. It is immutable.
   c. Expand the `com.example.dao` package. Look at the `EmployeeDAO` class and you see the methods that an implementation of this interface is expected to implement. Each of these methods throws a `DAOException`. Note that this interface extends AutoCloseable. Therefore, you will need to provide a `close()` method to satisfy the contract with AutoCloseable.
   d. Look at the `EmployeeDAOFactory`. You see that this class has one method, `getFactory()`, that returns an instance of an `EmployeeDAOMemoryImpl`.
   e. Look at the `EmployeeDAOMemoryImpl` class. This class is the workhorse of the DAO pattern. This is the class that the `EmployeeDAOJDBCFactory` returns as an instance from the `createEmployeeDAO` method. This is the class that you will replace with a JDBC implementation.
2. Create a new class, `EmployeeDAOJDBCImpl`, that implements `EmployeeDAO` in the `com.example.dao` package.
   a. Note that the class has an error.

3. Implement the method signatures defined by `EmployeeDAO`.

   a. Click anywhere in the line that is showing an error (a light bulb with a red dot on it):

   ```
   public class EmployeeDAOJDBCImpl implements EmployeeDAO {
   ```

   b. Press the Alt-Enter key combination to show the suggestions for fixing the error in this class. You should see the following:



   Note: Your line numbers may differ from the picture shown.

   c. The class must implement all the methods in the interface because this is a concrete class (not abstract). You can have NetBeans provide all the method bodies by pressing the Enter key to accept the suggestion "Implement all abstract methods."

   d. You will notice that the error in the file goes away immediately, and NetBeans has provided all the method signatures based on the `EmployeeDAO` interface declarations.

   e. Your next task is to add a constructor and fill in the bodies of the methods.

4. Add a private instance variable, `con`, to hold a `Connection` object instance.

5. Write a package-level constructor for the class. The constructor for this class will create an instance of a `Connection` object that the methods of this class can reuse during the lifetime of the application. Be sure to catch a `SQLException`.

6. Write a method body for the `add` method. The `add` method creates a new record in the database from the `Employee` object passed in as a parameter. Recall that the SQL command to create a new record in the database is: `INSERT INTO <table> VALUES (...)`.

   Note: Use single quotes for the strings and the date.

   a. Rethrow any `SQLException` caught as a `DAOException`.

7. Write a method body for the `findById` method. This method is used by the `update` and `delete` methods and is used to locate a single record to display. Recall that the SQL command to read a single record is: `SELECT * FROM <table> WHERE <pk>=<value>`.

   a. Rethrow any `SQLException` caught as a `DAOException`.

8. Write a method body for the `update` method. The `update` method updates an existing record in the database from the `Employee` object passed in as a parameter. Recall that the SQL command to create a new record in the database is: `UPDATE <table> SET COLUMNNAME=<value>, COLUMNNAME=<value>, ... WHERE <pk>=<value>`.

   Note: Be sure to add single quotes around string and date values.

   a. Rethrow any `SQLException` caught as a `DAOException`.

9. Write a method body for the `delete` method. The `delete` method tests to see whether an employee exists in the database by using the `findById` method, and then deletes the record if it exists. Recall that the SQL command to delete a record from the database is: `DELETE FROM <table> WHERE <pk>=<value>`.

a. Rethrow any `SQLException` caught as a `DAOException`.

10. Write the method body for the `getAllEmployees` method. This method returns an array of `Employee` records. The SQL query to return all records is quite simple: `SELECT * FROM <table>`.

    a. Rethrow any `SQLException` caught as a `DAOException`.

11. Write the method body for the `close` method. This method is defined by the `AutoCloseable` interface. This method should explicitly close the `Connection` object that you created in the constructor.

    a. Rather than rethrow the `SQLException`, simply report it.

12. Save the class. Fix any missing imports and compilation errors if you have not already done so.

13. Update the `EmployeeDAOFactory` to return an instance of your new `EmployeeDAOJDBCImpl`.

```
return new EmployeeDAOJDBCImpl();
```

14. Add the JDBC Derby driver class to the project, but adding the `derbyclient.jar` file to the Libraries for the project.

    a. Right-click the Libraries folder in the project and select Add Jar/Folder.

    b. Browse to `D:\Program Files\Java\jdk1.7.0\db\lib`. `(or your other folder)`

    c. Select `derbyclient.jar`.

    d. Absolute Path should be checked.

    e. Click Open.

15. Save the updated class, and if you have no errors, compile and run the project. This application has an interactive feature that allows you to query the database and read one or all of the records, find an employee by ID, and update and delete an employee record.

Practice 13-2: Detailed Level: Using the Data Access Object Pattern

Overview

In this practice, you will take the existing Employee DAO Memory application and refactor the code to use JDBC instead. The solution from the “Exceptions and Assertions” lesson has been renamed to `EmployeeDAOJDBC`. You will need to create an `EmployeeDAOJDBCImpl` class to replace the `EmployeeDAOMemoryImpl` class, and modify the `EmployeeDAOFactory` to return an instance of your new implementation class instead of the Memory version.

You will not have to alter the other classes. This example illustrates how a well designed Data Access Object application can use an alternative persistence class without significant change.

Tasks

1. Open and examine the `EmployeeDAOJDBC` project in the `D:\labs\13\practices` folder. `(or your other folder)`

   a. In the `com.example.test` package, you see the class `EmployeeTestInteractive`. This class contains the main method and provides a console-based user interface. Through this UI, you will be able to create new records, read all the records, update a record, and delete a record from the Employee database. Note how the main method creates an instance of a Data Access Object (DAO).

   b. In the `com.example.model` package, look at the `Employee` class. This class is a Plain Old Java Object (POJO) that encapsulates all the data from a single employee record and row in the Employee table. Note that there are no set methods in this class, only get methods. Once an Employee object is created, it cannot be changed. It is immutable.

   c. Expand the `com.example.dao` package. Look at the `EmployeeDAO` class and you see the methods that an implementation of this interface is expected to implement. Each of these methods throws a `DAOException`. Note that this interface extends AutoCloseable; therefore, you will need to provide a `close()` method to satisfy the contract with AutoCloseable.

   d. Look at the `EmployeeDAOFactory`. You see that this class has one method, `getFactory()`, that returns an instance of an `EmployeeDAOMemoryImpl`.

   e. Look at the `EmployeeDAOMemoryImpl` class. This class is the workhorse of the DAO pattern. This is the class that the `EmployeeDAOJDBCFactory` returns as an instance from the `createEmployeeDAO` method. This is the class that you will replace with a JDBC implementation.

2. Create a new class, `EmployeeDAOJDBCImpl`, in the `com.example.dao` package.

   a. Right-click on the `com.example.dao` package and choose New Java Class

   b. Enter `EmployeeDAOJDBCImpl` as the class name and click Finish.

   c. Change this class to implement `EmployeeDAO`.

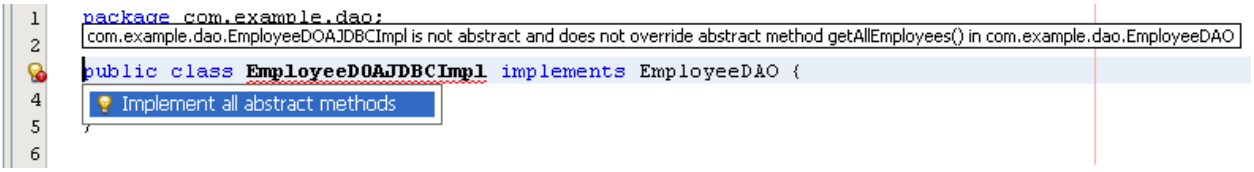
   d. Note that causes an error.

3. Implement the method signatures defined by `EmployeeDAO`.

   a. Click anywhere in the line that is showing an error (a light bulb with a red dot on it):

   ```
   public class EmployeeDAOJDBCImpl implements EmployeeDAO {
   ```

   b. Press the Alt-Enter key combination to show the suggestions for fixing the error in this class. You should see the following:

   Note: Your line numbers may differ from the picture shown.

   c. The class must implement all the methods in the interface because this is a concrete class (not abstract). You can have NetBeans provide all of the method bodies by pressing the Enter key to accept the suggestion "Implement all abstract methods."

   d. You will notice that the error in the file goes away immediately, and NetBeans has provided all the method signatures based on the `EmployeeDAO` interface declarations.

   e. Your next task is to add a constructor and fill in the bodies of the methods.

4. Add a private instance variable, `con`, to hold a `Connection` object instance.

   ```
   private Connection con = null;
   ```

5. Write a constructor for the class. The constructor for this class will create an instance of a `Connection` object that the methods of this class can reuse during the lifetime of the application.

   a. Write the constructor to use package-level access. This will enable only classes within the package to create an instance of this class (like the `EmployeeDAOFactory`).

   ```
   EmployeeDAOJDBCImpl() {
   ```

   b. Open the connection using the JDBC URL, name, and password from the `SimpleJDBCExample` application:

   ```
   String url = "jdbc:derby://localhost:1527/EmployeeDB";
   String username = "public";
   String password = "tiger";
   ```

   c. In a try block (not a `try`-with-resources, because you want to keep this connection open until you exit the application) create an instance of a `Connection` object and catch any exception. If the connection cannot be made, exit the application.

   Note: Ideally you would rather indicate to the user that the connection could not be made and retry the connection a number of times before exiting.

   ```
   try {
      con = DriverManager.getConnection(url, username, password);
   } catch (SQLException se) {
      System.out.println("Error obtaining connection with the
   database: " + se);
      System.exit(-1);
   }
   ```

6. Write a method body for the `add` method. The `add` method creates a new record in the database from the `Employee` object passed in as a parameter. Recall that the SQL command to create a new record in the database is: `INSERT INTO <table> VALUES (...)`.

   a. Delete the boiler-plate code created by NetBeans for the `add` method.

   b. Look at the other methods in the class. They each begin by creating an instance of a `Statement` object in a `try`-with-resources statement:

```
try (Statement stmt = con.createStatement()) {

}
```

   c. Inside the try block, create a query to insert the values passed in the Employee instance to the database. Your query string should look something like this:

```
String query = "INSERT INTO EMPLOYEE VALUES (" + emp.getId()
               + ", '" + emp.getFirstName() + "', "
               + "'" + emp.getLastName() + "', "
               + "'" +
       new java.sql.Date(emp.getBirthDate().getTime()) + "',"
               + emp.getSalary() + ")";
```

   Note the use of single quotes for the strings and the date.

   d. Since you are not expecting a result from the query, the appropriate `Statement` class method to use is `updateQuery`. Make sure to test to see whether the statement executed properly by looking at the integer result of the method. For example:

```
if (stmt.executeUpdate(query) != 1) {
    throw new DAOException("Error adding employee");
}
```

   e. At the end of the try block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example:

```
catch (SQLException se) {
    throw new DAOException("Error adding employee in DAO", se);
}
```

7. Write a method body for the `findById` method. This method is used by the `update` and `delete` methods and is used to locate a single record to display. Recall that the SQL command to read a single record is: "`SELECT * FROM <table> WHERE <pk>=<value>`".

   a. Delete the boiler-plate code created by NetBeans for the `findById` method.

   b. Create an instance of a `Statement` object in a `try`-with-resources block:

```
try (Statement stmt = con.createStatement()) {

}
```

   c.

Inside the try block, write a query statement to include the integer id passed in as an argument to the method and execute the query, returning a `ResultSet` instance:

```
String query = "SELECT * FROM EMPLOYEE WHERE ID=" + id;
ResultSet rs = stmt.executeQuery(query);
```

d. Test the `ResultSet` instance for null using the `next()` method and return the result as a new `Employee` object:

```
if (!rs.next()) {
    return null;
}
return (new Employee(rs.getInt("ID"),
                     rs.getString("FIRSTNAME"),
                     rs.getString("LASTNAME"),
                     rs.getDate("BIRTHDATE"),
                     rs.getFloat("SALARY")));
```

e. At the end of the try block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example

```
catch (SQLException se) {
    throw new DAOException("Error finding employee in DAO", se);
}
```

8. Write a method body for the `update` method. The `update` method updates an existing record in the database from the `Employee` object passed in as a parameter. Recall that the SQL command to create a new record in the database is: "`UPDATE <table> SET COLUMNNAME=<value>, COLUMNNAME=<value>, ... WHERE <pk>=<value>`".

   Note: Be sure to add single quotes around string and date values.

   a. Delete the boiler-plate code created by NetBeans for the `update` method.

   b. Create an instance of a `Statement` object in a `try`-with-resources block:

```
try (Statement stmt = con.createStatement()) {

}
```

   c. Inside the try block, create the SQL `UPDATE` query from the Employee object passed in:

```
String query = "UPDATE EMPLOYEE "
    + "SET FIRSTNAME='" + emp.getFirstName() + "',"
    + "LASTNAME='" + emp.getLastName() + "',"
    + "BIRTHDATE='" + new
java.sql.Date(emp.getBirthDate().getTime()) + "',"
    + "SALARY=" + emp.getSalary()
    + "WHERE ID=" + emp.getId();
```

   d. You may want to test to see that the update was successful by evaluating the return value of the `executeUpdate` method:

```
if (stmt.executeUpdate(query) != 1) {
    throw new DAOException("Error updating employee");
}
```

e. At the end of the try block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example

```
catch (SQLException se) {
    throw new DAOException("Error updating employee in DAO",
se);
}
```

9. Write a method body for the `delete` method. The `delete` method tests to see whether an employee exists in the database by using the `findById` method, and then deletes the record if it exists. Recall that the SQL command to delete a record from the database is: "`DELETE FROM <table> WHERE <pk>=<value>`".

   a. Delete the boiler-plate code created by NetBeans for the `delete` method.

   b. Call the `findById` method with the `id` passed in as a parameter and if the record returned is null, throw a new `DAOException`.

```
Employee emp = findById(id);
if (emp == null) {
    throw new DAOException("Employee id: " + id + " does not
exist to delete.");
}
```

   c. Create an instance of a `Statement` object in a `try`-with-resources block:

```
try (Statement stmt = con.createStatement()) {


}
```

   d. Inside the try block, create the SQL DELETE query and test the result returned to make sure a single record was altered – throw a new `DAOException` if not:

```
String query = "DELETE FROM EMPLOYEE WHERE ID=" + id;
    if (stmt.executeUpdate(query) != 1) {
        throw new DAOException("Error deleting employee");
}
```

   e. At the end of the try block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example:

```
catch (SQLException se) {
    throw new DAOException("Error deleting employee in DAO",
se);
}
```

10. Write the method body for the `getAllEmployees` method. This method returns an array of `Employee` records. The SQL query to return all records is quite simple: "`SELECT * FROM <table>`".
    a. Delete the boiler-plate code created by NetBeans for the `getAllEmployees` method.
    b. Create an instance of a `Statement` object in a `try`-with-resources block:

```
try (Statement stmt = con.createStatement()) {


}
```

    c. Inside the try block, create and execute the query to return all the employee records:

```
String query = "SELECT * FROM EMPLOYEE";
ResultSet rs = stmt.executeQuery(query);
```

    d. The easiest way to create an array of employees to return is to use a Collection object, `ArrayList`, and then convert the `ArrayList` object to an array. Iterate through the `ResultSet` and add each record to the `ArrayList`. In the return statement, use the `toArray` method to convert the collection to an array:

```
ArrayList<Employee> emps = new ArrayList<>();
while (rs.next()) {
    emps.add(new Employee(rs.getInt("ID"),
                          rs.getString("FIRSTNAME"),
                          rs.getString("LASTNAME"),
                          rs.getDate("BIRTHDATE"),
                          rs.getFloat("SALARY")));
}
return emps.toArray(new Employee[0]);
```

    e. At the end of the try block, catch any `SQLException` thrown, and wrap them in the `DAOException` to be handled by the calling application. For example:

```
catch (SQLException se) {
    throw new DAOException("Error getting all employees in DAO",
se);
}
```

11. Write the method body for the `close` method. This method is defined by the `AutoCloseable` interface. This method should explicitly close the `Connection` object that you created in the constructor.
    a. Delete the boiler-plate code created by NetBeans for the `close` method.
    b. In a try block (you must use a try block, because `Connection.close` throws an exception that must be caught or rethrown), call the `close` method on the `Connection` object instance, `con`. Rather than rethrow the exception, simply report it.

```
try {
    con.close();
} catch (SQLException se) {
    System.out.println ("Exception closing Connection: " + se);
}
```

12. Save the class. Fix any missing imports and compilation errors if you have not already.
13. Update the `EmployeeDAOFactory` to return an instance of your new `EmployeeDAOJDBCImpl`.

    ```
    return new EmployeeDAOJDBCImpl();
    ```

14. Add the JDBC Derby driver class to the project, by adding the `derbyclient.jar` file to the Libraries for the project.
    a. Right-click the Libraries folder in the project and select Add Jar/Folder.
    b. Browse to `D:\Program Files\Java\jdk1.7.0\db\lib`. `(or your other folder)`
    c. Select `derbyclient.jar`.
    d. Absolute Path should be checked.
    e. Click Open.
15. Save the updated class, and if you have no errors, compile and run the project. This application has an interactive feature that allows you to query the database and read one or all of the records, find an employee by ID, and update and delete an employee record.