DAO pattern tutorial

In this tutorial, you'll practice connecting to a database and retrieving data from it in Java. By the end of this tutorial, you'll have written code that:

- Connects to a PostgreSQL database
- Sends SQL queries to the database
- Creates Java objects that correspond to data returned by queries
- Encapsulates the database interactions using the DAO pattern

Getting started

To get started, follow these steps:

1. Import the project into IntelliJ IDEA.
2. Once the project is open, navigate to the project folder `src/main/java/com/techelevator`.
3. Open the `Tutorial.java` file by double-clicking on the filename.

You'll see the starter code for this project. This project also uses the `PizzaShop` database.

Step One: Configure the database connection

Before the program can connect to the database, you'll need to provide several pieces of information about the connection.

Find the first comment in `Tutorial.java`. You'll add your code after this line:

```
// Step One: Configure the database connection
```

Add three lines to set the URL, username, and password for the connection:

```
dataSource.setUrl("jdbc:postgresql://localhost:5432/PizzaShop");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");
```

Warning: Including credentials like a username and password in your source code is convenient while learning, but it's a major security risk. In the future, you'll learn how to avoid doing this by using environment variables or configuration files.

After adding those lines, run the `main()` method in the `Tutorial` class. You'll see one line of output:

```
Total Sales: $0
```

Step Two: Add SQL for retrieving total sales

Data about the pizza shop's sales is stored in the `sale` table of the database. In the DAO pattern, there are three Java files associated with a database table. For the `sale` table, those three files are:

- `Sale.java`: a class used for objects that each correspond to a row of the `sale` table
- `SaleDao.java`: an interface that specifies the available methods for getting information from the `sale` table
- `JdbcSaleDao.java`: a class that implements the methods specified by the `SaleDao` interface

All the SQL used to retrieve data from the `sale` table is encapsulated in methods of the `JdbcSaleDao` class. Open that class and look at the `getTotalSales()` method. You'll see that it currently consists of only one line of code following the "Step Two" comment:

```
return jdbcTemplate.queryForObject("SELECT 0;", BigDecimal.class);
```

That line uses an instance of the Spring Framework's `JdbcTemplate` class to send a SQL query to the database and return a single `BigDecimal` value. Currently, the query is `SELECT 0;` which means the total returned is always zero.

Replace that query with `SELECT SUM(total) FROM sale;`

Run the `Tutorial` class again, and you'll see that the output has changed to this:

```
Total Sales: $1248.80
```

Step Three: Copy returned values into an object

The `getTotalSales()` method retrieved a single value from the database, but the query sent to the database in the `getSale()` method returns an entire row of data. The `mapRowToSale()` method handles creating a `Sale` object from that row of data. Currently, however, it's returning an empty object.

To set the properties of the object with the values from the row of data, add this code following the "Step Three" comment in `JdbcSaleDao.java`:

```
sale.setSaleId(rowSet.getLong("sale_id"));
sale.setTotal(rowSet.getBigDecimal("total"));
sale.setDelivery(rowSet.getBoolean("is_delivery"));
sale.setCustomerId(rowSet.getLong("customer_id"));
if (rowSet.wasNull()) {
    sale.setCustomerId(null);
}
```

Each line calls a setter and passes it a value retrieved from the `SqlRowSet`. Notice that there are different methods to use for each data type: `getLong()` for `long` values, `getBoolean()` for `boolean` values, and so on.

Also notice that after setting the `customerId` property, an additional call is made to check if the last value retrieved from the database was null. That extra check is necessary because null values are permitted in that column, but the `getLong()` method can't return null.

After adding those lines, return to the `Tutorial` class. Following the "Step Three" comment, add this code to the `run()` method to get a `Sale` object representing the sale with an id of 50, and print it out:

```
Sale sale50 = saleDao.getSale(50);
System.out.println(sale50);
```

Run the `Tutorial` class again, and you'll see that the output has changed to this:

```
Total Sales: $1248.80
Sale 50: $12.99 (carryout)
```

Step Four: Add a new DAO method

In addition to the DAO for the `sale` table, this project also contains a DAO for the `customer` table. It also consists of three files: `Customer.java`, `CustomerDao.java`, and `JdbcCustomerDao.java`. You were able to retrieve a `Sale` object based on its id in the `sale` table using the `getSale()` method of the `SaleDao` interface, but there's currently no way to retrieve a `Customer` object based on its id in the `customer` table.

To add that capability, begin by updating the `CustomerDao` interface with a new method declaration following the "Step Four" comment:

```
Customer getCustomer(long customerId);
```

Remember that interfaces don't normally contain implementations of methods. Interfaces are like contracts, specifying what methods the implementing class—`JdbcCustomerDao` in this case—must have. The interface is an important part of the DAO pattern because it helps make code loosely coupled, which facilitates testing and reuse.

Now that you've added the declaration of `getCustomer()` to the `CustomerDao` interface, implement the method in the `JdbcCustomerDao` class following the "Step Four" comment:

```
@Override
public Customer getCustomer(long customerId) {
    Customer customer = null;
    String sql = "SELECT customer_id, first_name, last_name, street_address, city, 
phone_number, " +
                    "email_address, email_offers " +
                    "FROM customer " +
                    "WHERE customer_id = ?;";
    SqlRowSet results = jdbcTemplate.queryForRowSet(sql, customerId);
```

```
        if (results.next()) {
            customer = mapRowToCustomer(results);
        }
        return customer;
    }
```

Notice that the SQL query used in this method contains a `?`. That `?` is a placeholder for a **parameter**, or a value that's specified when the query is sent to the database. You can see that on the next line where `queryForRowSet()` is called with both the string containing the SQL and the value to use for that parameter, which is `customerId` in this case.

Now that the `CustomerDao` interface offers a `getCustomer()` method, return to the `Tutorial` class and add these lines to the `run()` method to display information about the customer associated with the sale you retrieved data about earlier:

```
Customer customerForSale50 = customerDao.getCustomer(sale50.getCustomerId());
System.out.println("Customer for that sale was " + customerForSale50);
```

Run the `Tutorial` class again, and you'll see that the output looks like this:

```
Total Sales: $1248.80
Sale 50: $12.99 (carryout)
Customer for that sale was Elenore Mamwell
```

Step Five: Call a DAO method that returns a `List`

In addition to returning values or single objects, DAO methods frequently return `List`s of objects. You can find an example of this in the `CustomerDao`, where a `findCustomersByName()` method is declared. As the comments document, this method returns a `List` of `Customer` objects for customers whose first name or last name contains the specified search string.

Switch to the `JdbcCustomerDao` and look at how that method is implemented. It's similar to the method you added in the previous step, with a few important differences to notice:

- The SQL query has two parameters, and so there are two `?` symbols. The values for those are specified in order in the call to `queryForRowSet()`. In this case, the same value is used in both positions, so `search` is passed twice.
- The SQL query uses a `LIKE` for searching, and the `%` symbols are added to the parameter value rather than included in the SQL.
- Since multiple rows of results are expected, a `while` loop is used to iterate through those rows and create a `Customer` object for each.

After reviewing that method, return to the `Tutorial` class and add these lines to the `run()` method to search for all customers with "Ma" in their first or last name:

```
List<Customer> matchingCustomers = customerDao.findCustomersByName("Ma");
System.out.println("All customers with \"Ma\" in their first or last name:");
for (Customer customer : matchingCustomers) {
    System.out.println(customer);
}
```

Run the `Tutorial` class again, and you'll see that the output is now this:

```
Total Sales: $1248.80
Sale 50: $12.99 (carryout)
Customer for that sale was Elenore Mamwell
All customers with "Ma" in their first or last name:
Deanne Mallon
Madge Lampaert
Elenore Mamwell
Margaret Peepall
Raquel Marcome
```

Next steps

Now that you've been introduced to the DAO pattern and how it's used to retrieve information from the database, there are many possibilities for further experimentation.

For example, you could add additional methods to the `SaleDao` or `CustomerDao` interfaces, like a method for retrieving all the sales associated with a particular customer or a method for retrieving all the customers who want to receive email offers.

Another possibility would be to add a new DAO for one of the other tables in the `PizzaShop` database. If you chose the `pizza` table, you'd create a `Pizza` class, a `PizzaDao` interface, and a `JdbcPizzaDao` class.

Finally, DAOs aren't limited to retrieving data. You could add methods to these DAOs that send `INSERT`, `UPDATE`, and `DELETE` SQL commands to the database, to enable creating, updating, and removing records.