```
    }
     }
```

This is a simple domain object despite those visually noisy XML annotations. So let's move to the `BankingTxDAO` interface next.

5.2.2 Defining DAO Interface

The DAO interface corresponding to the domain object of `BankingTx` is exhibited in Listing 5.3. This is a very simple Java interface, and it's very self-explanatory. After you take a quick look at it, let's move to the next section which explains in detail how these interfaces are implemented in JDBC, or specifically the Spring JDBC Framework.

Listing 5.3 BankingTxDao.java

```
package com.perfmath.spring.soba.model.dao;

import java.util.List;
import java.util.Map;
import com.perfmath.spring.soba.model.domain.BankingTx;

public interface BankingTxDao {
    public List<BankingTx> getTransactionList();
    public List<BankingTx> getTransactionList(String accountId);
  public void insert(BankingTx transaction);
  public void update(BankingTx transaction);
  public void delete(String transactionId);
  public BankingTx findByTransactionID(String transactionId);
  public void insertBatch(List<BankingTx> transactions);
  public List<Map<String, Object>> findAll();
  public String getAccountId(String transactionId);
  public int countAll();
}
```

5.3 IMPLEMENTING DAO WITH SPRING JDBC TEMPLATE

This section explains in detail how the `BankingTxDAO` interface defined in the preceding section is implemented with the two major Spring JDBC classes: `JdbcTemplate` and `NamedParameterJdbcTemplate`,against the domain object BankingTx defined in Listing 5.2. There are many variations with Spring JDBC, but I feel that these two classes should be sufficient most of the time. You should refer to *Chapter 13 Data Access with JDBC* of the *Spring Reference Documentation 3.1* if you want to know every detail about this subject. Here, I'd like to present a succinct example to show how it works.

First, let's show how the `JDBCBankingTxDao.java` code is implemented in Listing 5.4. Note the highlighted parts that we explain next following the code listing.

Listing 5.4 JDBCBankingTxDao.java

```
    package com.perfmath.spring.soba.model.dao;

    import java.sql.ResultSet;
    import java.sql.SQLException;
    import java.util.List;
    import java.util.Map;

    import javax.sql.DataSource;

    import
org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
    import
org.springframework.jdbc.core.namedparam.NamedParameterJdbcDaoSupport;
    import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
    import org.springframework.jdbc.core.namedparam.SqlParameterSource;
    import org.springframework.jdbc.core.namedparam.SqlParameterSourceUtils;
    import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
    import org.springframework.jdbc.core.JdbcTemplate;

    import com.perfmath.spring.soba.model.domain.BankingTx;

    public class JdbcBankingTxDao implements BankingTxDao {
        private JdbcTemplate jdbcTemplate;
        private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

        public List<BankingTx> getTransactionList() {
            List<BankingTx> txs = this.jdbcTemplate
                    .query("SELECT TRANSACTION_ID, TRANS_DATE, TYPE, "
                            + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID,
STATUS FROM BANKINGTX",
                            new TransactionMapper());
            return txs;
        }

        public List<BankingTx> getTransactionList(String accountId) {
            List<BankingTx> txs = this.jdbcTemplate
                    .query("SELECT TRANSACTION_ID, TRANS_DATE, TYPE, "
                            + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, "
                            + " ACCOUNT_ID, STATUS FROM BANKINGTX WHERE
ACCOUNT_ID = ?"
                            + " ORDER BY TRANS_DATE DESC", new TransactionMapper(),
                            accountId);
            return txs;
        }
```

```
public void insert(BankingTx transaction) {
    String sql = "INSERT INTO BANKINGTX (TRANSACTION_ID, TRANS_DATE, TYPE,"
        + "INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS) “
        + "VALUES (:transactionId, :transDate, :type, :initiator, "
        + ":description, :amount, :balance, :accountId, :status)";
    SqlParameterSource namedParameters = new
    BeanPropertySqlParameterSource(     transaction);

    int count = this.namedParameterJdbcTemplate
            .update(sql, namedParameters);
}

public void insertBatch(List<BankingTx> transactions) {
    String sql = "INSERT INTO BANKINGTX (TRANSACTION_ID, TRANS_DATE, TYPE,"
        + "INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS ) "
        + "VALUES (:transactionId, :transDate, :type, :initiator, "
        + ":description, :amount, :balance, :accountId, :status)";


    SqlParameterSource[] parameterSource  =
    SqlParameterSourceUtils.createBatch(transactions.toArray());

    int count[] = this.namedParameterJdbcTemplate
            .batchUpdate(sql, parameterSource);  }

public BankingTx findByTransactionID(String transID) {

    String sql = "SELECT TRANSACTION_ID, TRANS_DATE, TYPE, "
        + " INITIATOR, DESCRIPTION, AMOUNT, BALANCE, ACCOUNT_ID, STATUS "
        + " FROM BANKINGTX WHERE TRANSACTION_ID = '" + transID + "'";
    BankingTx trans = this.jdbcTemplate.queryForObject(sql,
            new TransactionMapper());

    return trans;
}

public void update(BankingTx tx) {
}

public void delete(String txId) {
    String sql = "DELETE BANKINGTX WHERE TRANSACTION_ID = ?";

    int count = this.jdbcTemplate.update(sql, txId);
}
```

```
    public List<Map<String, Object>> findAll() {
        String sql = "SELECT * FROM BANKINGTX";

        List<Map<String, Object>> trans = this.jdbcTemplate.queryForList(sql,
                new TransactionMapper());
        return trans;
    }

    public String getAccountId(String transID) {
        String sql = "SELECT ACCOUNT_ID FROM BANKINGTX WHERE TRANSACTION_ID
        = ?";
        String accountId = this.jdbcTemplate.queryForObject(sql, String.class,
                transID);
        return accountId;
    }

    public int countAll() {
        String sql = "SELECT COUNT(*) FROM BANKINGTX";

        int count = this.jdbcTemplate.queryForInt(sql);
        return count;
    }

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(
                dataSource);
    }

    private static class TransactionMapper implements
            ParameterizedRowMapper<BankingTx> {
        public BankingTx mapRow(ResultSet rs, int rowNum) throws SQLException {
            BankingTx tx = new BankingTx();
            tx.setTransactionId(rs.getLong("TRANSACTION_ID"));
            tx.setTransDate(rs.getTimestamp("TRANS_DATE"));
            tx.setType(rs.getString("TYPE"));
            tx.setInitiator(rs.getString("INITIATOR"));
            tx.setDescription(rs.getString("DESCRIPTION"));
            tx.setAmount(rs.getDouble("AMOUNT"));
            tx.setBalance(rs.getDouble("BALANCE"));
            tx.setAccountId(rs.getString("ACCOUNT_ID"));
            tx.setStatus(rs.getString("STATUS"));
            return tx;
        }
    }
}
```

Here is a generic pattern adopted in implementing the JDBCBankingTxDao class show in the above class:

- Add two variables of `jdbcTemplate` and `namedParameterJdbcTemplate` follows:

```
private JdbcTemplate jdbcTemplate;
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

- Add a setDataSource method as shown in the above listing. The `datasource` object is configured externally in `soba-services.xml` and you don't have to worry about it here.
- Define a `TransactionMapper` class as shown in the above listing.

The rule of thumb is that you use the SQL operations from the `JdbcTemplate` class with SQLs that do not use named parameters, but you need to use `NamedParameterJdbctemplate` class with SQLs that use named parameters. Otherwise, if use something like `this.jdbcTemplate.update (sql, namedParameters)` (for example, with the insert method), it would compile fine, but you would get the following error when it is actually executed, for example, with `createTransaction` calls in SOBA:

```
java.sql.SQLException: Invalid argument value: java.io.NotSerializableException
com.mysql.jdbc.SQLError.createSQLException(SQLError.java:1055)
com.mysql.jdbc.SQLError.createSQLException(SQLError.java:956)
com.mysql.jdbc.SQLError.createSQLException(SQLError.java:926)
com.mysql.jdbc.PreparedStatement.setSerializableObject(PreparedStatement.java:4278)
com.mysql.jdbc.PreparedStatement.setObject(PreparedStatement.java:3922)
org.apache.commons.dbcp.DelegatingPreparedStatement.setObject(DelegatingPreparedStatement.java:255)
org.springframework.jdbc.core.StatementCreatorUtils.setValue(StatementCreatorUtils.java:351)
org.springframework.jdbc.core.StatementCreatorUtils.setParameterValueInternal(StatementCreatorUtils.java:21
6)
org.springframework.jdbc.core.StatementCreatorUtils.setParameterValue(StatementCreatorUtils.java:144)
org.springframework.jdbc.core.ArgPreparedStatementSetter.doSetValue(ArgPreparedStatementSetter.java:65)
org.springframework.jdbc.core.ArgPreparedStatementSetter.setValues(ArgPreparedStatementSetter.java:46)
org.springframework.jdbc.core.JdbcTemplate$2.doInPreparedStatement(JdbcTemplate.java:816)
org.springframework.jdbc.core.JdbcTemplate$2.doInPreparedStatement(JdbcTemplate.java:1)
org.springframework.jdbc.core.JdbcTemplate.execute(JdbcTemplate.java:587)
org.springframework.jdbc.core.JdbcTemplate.update(JdbcTemplate.java:812)
org.springframework.jdbc.core.JdbcTemplate.update(JdbcTemplate.java:868)
org.springframework.jdbc.core.JdbcTemplate.update(JdbcTemplate.java:876)
```

Therefore, it's important to know when to use `JdbcTemplate` and when to use `NamedParameterJdbcTemplate`, as explained next.

5.3.1 JdbcTemplate.query

This method is for returning a result set of all or partial rows in your database from the table corresponding to your domain object. It is used to implement the two `getTransactionList` methods, one for all transactions, and the other for a specific account. Note that in the second method, a question mark "?" is used in the SQL query to hold the `ACCOUNT_ID` variable. If you have more variables, then you can use more question marks, but you need to make sure the sequence of the values provided after the second argument `new TransactionMapper()` matches. The return type is a Java List in this case.

5.3.2 JdbcTemplate.queryForObject

This method is applied to `findByTransactionID` and `getAccountId`. In the first case, the returned object is a `BankingTx` object, and in the second case, the returned object is a String representing an account id.

5.3.3 JdbcTemplate.queryForList

This method is applied to the method `findAll`, which returns all banking transactions as a Java List. Note that you need to specify `new TransactionMapper ()` for the second argument of this `queryForList` call, which used to be `BankingTx.class` with the deprecated `getSimpleJdbcTemplate` method.

5.3.4 JdbcTemplate.queryForInt

This method is applied to countAll, which returns the number of rows of the domain object in the database.

5.3.5 NamedParameterJdbcTemplate.update

This method covers all SQL statements of `INSERT`, `UPDATE` and `DELETE`. JDBC does not have a method for each of these SQL operations. Instead, the update method applies to all these cases, and the concrete operation is resolved internally.

Note in the `insert` method, you see items preceded with ":" in the SQL statement instead of question marks. These are called named parameters, which is a feature provided with the Spring JDBC Framework. It is used in conjunction with SqlParameterSource, which automatically matches the passed-in object with the named parameters. Once again, if you use this.jdbcTemplate instead of namedParameterJdbcTemplate, you would get the *java.sql.SQLException: Invalid argument value: java.io.NotSerializableException* error as shown previously.

5.3.6 NamedParameterJdbcTemplate.batchUpdate

This method enables batch operations, for example, inserting/deleting/updating multiple rows in one batch to the database. Note that you need to create a `SqlParameterSource` array with `SqlParameterSourceUtils.createBatch` for the second argument of the `batchUpdate` method. Array processing is a common practice to improve the performance and scalability of a database-centric enterprise application.

Next, let's see how these JDBC DAOs are used by service beans to query or persist the model from and to the database.

5.4 SERVICE BEANS